

Fine-grained concurrency with separation logic

Kalpesh Kapoor

Kamal Lodaya

Uday Reddy

March 9, 2010

Abstract

Reasoning about concurrent programs involves representing the information that concurrent processes manipulate disjoint portions of memory. In sophisticated applications, the division of memory between processes is not static. Through operations, processes can exchange the implied ownership of memory cells. In addition, processes can also share ownership of cells in a controlled fashion as long as they perform operations that do not interfere, e.g., they can concurrently read shared cells. Thus the traditional paradigm of distributed computing based on locations is replaced by a paradigm of concurrent computing which is more tightly based on program structure.

Concurrent Separation Logic with Permissions, developed by O’Hearn, Bornat et al, is able to represent sophisticated transfer of ownership and permissions between processes. We demonstrate how these ideas can be used to reason about fine-grained concurrent programs which do not employ explicit synchronization operations to control interference but cooperatively manipulate memory cells so that interference is avoided. Reasoning about such programs is challenging and appropriate logical tools are necessary to carry out the reasoning in a reliable fashion. We argue that Concurrent Separation Logic with Permissions provides such tools. We illustrate the logical techniques by presenting the proof of a concurrent garbage collector originally studied by Dijkstra et al.

1 Introduction

Reasoning about concurrent programs, where multiple processes (or **threads**) are executed in parallel, is widely acknowledged to be one of the most challenging aspects of computer programming. The early work on the problem, carried out by Dijkstra, Hoare, Brinch Hansen and others, emphasized the need to keep the concurrent threads of control as independent as possible, working with separate areas of storage. When shared areas of storage need to be manipulated, for example for the purpose of communication between threads, synchronisation protocols are used to ensure that a single thread is accessing a shared region at any given time. A variety of synchronisation mechanisms, such as atomic statements [17], semaphores [8], conditional critical regions [14] and monitors [5, 15], have been developed to ensure mutually exclusive access to shared storage. During a period of such exclusive access (called a “critical section”), a thread is expected to carry out a well-defined operation on the shared storage to its conclusion. The amount of activity carried out during a critical section is referred to as the *granularity* of concurrency. Coarse granularity, where large operations are carried out in critical sections, is easier to reason about, but it is bad for performance. When one process is executing a critical section, the other processes are blocked from proceeding. In this paper, we address the issues in dealing with fine granularity. Here the atomic operations are small, being at the level of individual machine instructions. Hence, they achieve high performance. Correspondingly, they pose considerable challenges in reasoning.

A second aspect we are interested in is the use of *dynamic storage* (also referred to as the “heap” storage). Computer programs normally work by setting and modifying storage variables during their execution which might be thought of –superficially– as *variable symbols* as in algebra or symbolic logic. To reason about program behaviour, one uses some form of before-and-after specifications. A standard

form is that of Hoare Logic [13], which uses specifications of the form $\{P\} C \{Q\}$ where C is a command, and P and Q are formulas in classical logic (referred to as “assertions”).¹ The assertion P describes a hypothetical state of the program variables before the command execution and Q describes the state obtained after the command execution. A crucial point is that the program variables occurring in the command C are treated in the assertions as if they were ordinary logical variables. Since the storage manipulated by a program is in one-to-one correspondence with the program variables occurring in it, it means that the storage manipulated by the command is more or less fixed. This is a significant limitation of this formalism.

The use of dynamic storage involves the allocation of new storage cells in the course of program execution. These cells are referred to in the program by their unique identifying *addresses* (also referred to as “pointers”). In contrast to the variable symbols mentioned in the preceding paragraph, addresses are part of *data*. Commands can store them in variables or other heap cells and compute with them before deciding to read or write the storage cells that they point to. This means that the structure of the storage manipulated by a program fragment is not fixed in advance and not easily predictable. Program reasoning must deal with the structure of storage, in addition to dealing with the state of such storage. Due to the difficulty of dealing with this issue cleanly and reliably, most theoretical treatments of concurrent programming have completely avoided dynamic storage. (For example, the widely used text book on concurrent programming by Andrews [1] makes no mention of dynamic storage at all.)

A breakthrough in reasoning about dynamic storage was made by Reynolds [27], who used before-and-after specifications $\{P\} C \{Q\}$ where P and Q are formulas in a resource-sensitive logic called the *Logic of Bunched Implications* (BI). The logic BI, formulated by Pym and O’Hearn [20, 25], is a form of substructural logic — in fact, a bunched logic [26] — representing a symmetric combination of the BCI relevant logic, on the one hand, and intuitionistic or classical logic on the other.² BI differs from other forms of relevant logics in having a rich class of models that incorporate a notion of “resource”.

Let us agree that a resource is some form of an entity which has identity and permanence, and which cannot be freely created, destroyed or duplicated. Storage cells themselves form an excellent example of such “resource”. The connectives of the BCI fragment of BI (called the “multiplicative” connectives) allow us to navigate in the plane of resources, whereas those of the classical fragment (called the “additive” connectives) allow us to stay within a context of resources and reason about it in the traditional fashion. For example, the multiplicative conjunction $P \star Q$ is true for a *combination* of two separate collections of resources if the two collections satisfy P and Q respectively. In contrast, the additive conjunction $P \wedge Q$ is true for a collection of resources if both P and Q are true for that *same* collection. The unit for the multiplicative conjunction, written as “**emp**”, is true for the empty collection of resources and nothing else. In contrast, the additive unit, written as “**true**” is true for any collection of resources. A comprehensive proof-theoretic and model-theoretic study of the logic BI can be found in [25].

Reynolds used BI’s \star connective to make assertions about separate parts of the heap storage. These ideas were further developed by Ishtiaq and O’Hearn [16] by adding the requirement of tight specifications. O’Hearn also developed a form of the logic for dealing with concurrent programs [19] with a soundness proof provided by Brookes [6]. Bornat et al [4] enriched the framework by adding a notion of permissions. The logic resulting from all these developments may be termed “Concurrent Separation Logic with permissions” and forms the basis of our study.

Our objective is to test the efficacy of these techniques by applying them to a substantial problem of program proof. The algorithm chosen for this task is that of a concurrent garbage collector due to Dijkstra et al [10]. This is perhaps one of the first challenging concurrent algorithms whose correctness proof was attempted, and has an interesting history. At the time of this proof attempt, virtually no

¹From a logical point of view, Hoare Logic can be thought of as a modal logic, studied by Pratt [24]. A specification $\{P\} C \{Q\}$ is interpreted as a modal formula $P \supset [C]Q$ where the modal operator $[C]$ means “after the execution of C .”

²The development of this logic owes some inspiration to Girard’s Linear Logic [11]. However, its structure and model-theoretic import are quite different.

formal proof techniques were known for fine-grained concurrent programs. The authors used a form of informal reasoning that is ambitious in its scope. An early version of the algorithm [9] had a fault which was discovered after the version was submitted for publication. The final published proof is still too informal to carry full conviction. In the interim period, a formal proof technique for fine-grained concurrent programs was developed by Owicki and Gries [22] and Gries was able to prove the correctness of the algorithm using this technique [12]. This course of events is often used in the field of concurrency to illustrate the challenges underlying concurrent programming. (See, e.g., [7].) Since the publication of this algorithm, many researchers have given alternative proofs and algorithms. For example Ben-Ari, in [3], gave an algorithm that uses two colours and has less complexity. Flaws in his correctness proof were found when checking the proof mechanically [29]. Torp-Smith et al [30] have applied (sequential) Separation Logic to prove the correctness of a copying garbage collector.

Through our proof attempt using Concurrent Separation Logic, we wish to demonstrate how the novel techniques of program logic can be used to reason about concurrent algorithms more reliably. In fact, we claim that the Separation Logic proof exhibits structure which makes it almost impossible to ignore the flaw that was present in the original version of the algorithm. Gries [12] also made a claim that a “careful application” of the Owicki-Gries technique can avoid such errors in reasoning. Clearly, Gries’s proof is a vast improvement in clarity and formality over the previous informal proof. However, it is not a closed chapter. Prensa Nieto et al [18] make the point that a complete pencil and paper proof using this technique is very tedious. For this reason, many of the interference-freeness checks are “usually omitted.” In the Separation Logic approach, on the other hand, these issues are rather at the forefront. It is not even possible to formulate a resource invariant without a clear understanding of how the permissions are distributed among the various components. In this sense, we argue that the Separation Logic techniques provide the right set of logical tools for this problem.

1.1 Concurrent Separation Logic

As mentioned above, Reynolds [27] used the logic of BI to formulate assertions about heap storage. Ishtiaq and O’Hearn [16] made the additional formulation that a specification $\{P\}C\{Q\}$ should be regarded as valid only if C is able to execute *without faults* starting from any heap satisfying P . In particular, C cannot read and write any heap cells that are not in the domain of P . Such specifications are called “tight specifications.” The logic of programs resulting from the two developments is termed Separation Logic and described by Reynolds [28]. This logic admits an elegant proof rule for parallel composition of commands $C_1 \parallel C_2$:

$$\frac{\{P_1\} C_1 \{Q_1\} \quad \{P_2\} C_2 \{Q_2\}}{\{P_1 \star P_2\} C_1 \parallel C_2 \{Q_1 \star Q_2\}}$$

To see why such a rule is sound, consider a heap store that satisfies $P_1 \star P_2$. By the definition of \star , the heap store can be split into two separate halves (with disjoint collections of cells), satisfying P_1 and P_2 respectively. By the tightness property of specifications, we know that C_1 runs without any faults starting from the part that satisfies P_1 . In particular, it does not access any cells in the other part. C_2 does its work similarly, in its part. So, C_1 and C_2 are able to run in parallel, independently and without interference. We think of the portion of the heap store manipulated by each parallel process, and delineated by the corresponding pre-condition in its specification, as being “owned” by that process. The other processes cannot interfere with the storage owned by a process in this fashion. Upon termination of both the processes, we obtain a heap store that satisfies $Q_1 \star Q_2$. (The reader might contemplate how one might go about formulating a rule for parallel composition without the \star connective and the tightness requirement.)

Concurrent programming also requires that there be some form of communication between the concurrent processes. In the framework of fine-grained concurrency, such communication is achieved

by executing atomic operations on shared storage. (An atomic operation is an operation carried out by a process without interruption and interference from other processes.) O’Hearn’s proposal in the formulation of Concurrent Separation Logic [19] is to view the shared storage as being made up of one or more *shared resources* which are separate from the storage directly “owned” by the processes.³ The properties of the resources are expressed through *resource invariant* assertions. We use judgments of the form $R \vdash \{P\} C \{Q\}$ to mean that a process C has the before-and-after specification $\{P\} C \{Q\}$ in the context of a resource with resource invariant R . Through an atomic operation, written in the form $\langle A \rangle$ where A is a command, a process can “grab” the storage of a shared resource and temporarily make it a part of the owned state of the process for the duration of A . After the completion of A , the storage is returned back to the shared resource. This form of grabbing is best thought of as “borrowing”. It is worth emphasizing that these ideas of borrowing and returning are not actually computations; they represent our logical view of how the storage is being managed. The shared resource is expected to satisfy the resource invariant at *all times* except when it is borrowed by atomic operations. So, an atomic operation $\langle A \rangle$ can assume that the resource invariant is true when it begins execution and reestablish it again upon the completion of A . All this can be expressed succinctly by the proof rule below:

$$\frac{\{P \star R\} A \{Q \star R\}}{R \vdash \{P\} \langle A \rangle \{Q\}}$$

The formal semantics of a proof system with more generous judgments $\Gamma \vdash \{P\} C \{Q\}$ as well as its soundness were described in [6]. In most of the paper we are concerned only with one resource and one resource invariant, which we write as RI . A brief review of the formalism for the purposes of this article is given in Appendix A.

1.2 Access Permissions

The basic Concurrent Separation Logic treats each storage location as an atomic resource. So, every location would be owned by either one of the processes or one of the resources. However, in concurrent programming, it is also necessary to allow two processes to access shared locations in a controlled fashion, for simultaneous read access as well as other forms of controlled sharing. This can be achieved by treating as resources, not entire locations but particular *access permissions* on them. Bornat et al [4] proposed two forms of permissions suitable for this purpose, and we use a simplified form of their “counting permissions.”

A full permission on a storage location is denoted as “1” and it allows both reading and writing of the location. A full permission can be split into two permissions, denoted ρ and $\bar{\rho}$, both of which allow only reading of the location. Formally, we have a partial commutative semigroup with its binary operation defined by $\rho \star \bar{\rho} = 1$. A basic assertion in our logic would be of the form $l \vdash^{\rho} x$ which is thought of as an agent possessing a ρ permission for a storage location l , which holds the value x .⁴

We indicate how this set up of permissions can be employed for program reasoning in the context of two processes C_1 and C_2 with a shared resource r , and a location l used in both of them.

- If one process has the full permission for l then neither the shared resource nor the other process can have any access to l . The owning process of l can perform reading and writing and use facts about l in its local assertions.
- Suppose the two processes have ρ and $\bar{\rho}$ permissions for the location respectively. Then the shared resource does not have any permissions, and the two processes are restricted to reading the location.

³The use of “resource” for the packets of shared storage is inherited from Hoare [14]. It is a more specialised notion than the general logical notion of resource mentioned earlier in the Introduction.

⁴While this can be thought of as a deontic attitude of an agent, our concerns lie elsewhere.

- The third, interesting case, arises when the shared resource is given ρ permission for the location and the complement $\bar{\rho}$ permission is given to one of the processes, say C_1 . In this case, C_1 can read the location in the normal course of affairs, but it can also *write* to the location in atomic operations $\langle C_1 \rangle$ by borrowing the permission from the resource. It can make local assertions about the location l using its $\bar{\rho}$ permission. For instance, it is possible to conclude a specification of the form:

$$\{l \vdash_{\bar{\rho}} x\} \langle [l] := 23 \rangle \{l \vdash_{\bar{\rho}} 23\}$$

which is a bit counter-intuitive because we are able to make changes to l and reason about these changes even though the process has only a read permission! On the other hand, the process C_2 can only read the location l by borrowing the ρ permission from the shared resource. Since it has no permissions of its own for l , it cannot make any assertions about l .

- Finally, suppose the shared resource is given the full permission for l . Then both the processes can borrow this permission to read as well as write to l . However, lacking their own permissions for l , they cannot make any assertions about l . Their knowledge about the values read from l are limited to whatever properties are guaranteed by the resource invariant.

Such refined control over the access permissions to shared locations comes in very handy in ensuring that correct reasoning is carried out about concurrent execution behaviour.

1.3 Permission transfer

The permissions associated with the processes and shared resources are not static. They can vary during program execution, governed by the resource invariants which control what permissions are owned by the shared resources [19]. Consider again the situation of two processes C_1 and C_2 interacting via a shared resource r . Suppose the resource invariant for r is:

$$(x = 0 \wedge \mathbf{emp}) \vee (x = 1 \wedge l \vdash_{\rho} _)$$

We use $_$ as a short-hand notation for a don't-care value (which can be formalised as an existentially quantified variable). If the process C_1 does an atomic action such as $\langle x := 0 \rangle$, it sends the resource from a state where it might have ρ permission for l to a state where it has no permission for l . However, permissions are being treated as resources themselves. So, they cannot simply disappear. The effect is that the permission gets retrieved by the process C_1 . In other words, we have the following before-and-after specification for the atomic command:

$$\{x = 1 \wedge \mathbf{emp}\} \langle x := 0 \rangle \{x = 0 \wedge l \vdash_{\rho} _ \}$$

If the process already had $\bar{\rho}$ permission for l , then it is able to upgrade it to a full permission through this command. If, on the other hand, the process C_2 had $\bar{\rho}$ permission for l , then this operation takes away the ability of C_2 to make any further changes to l .

Changing x from 0 to 1 has the opposite effect of transferring ρ permission from the process to the resource:

$$\{x = 0 \wedge l \vdash_{\rho} _ \} \langle x := 1 \rangle \{x = 1 \wedge \mathbf{emp}\}$$

Thus, treating permissions as a resource provides a rich mechanism of dynamics of permission transfer, which comes in useful for reasoning about the behaviour of concurrent algorithms. Variants of this form of reasoning appear several times in this paper, in particular see Sections 4.3 and 6.

2 The DLMSS garbage collection algorithm

Most general purpose programming languages provide some mechanism to allocate objects dynamically, that is, at run time. This is facilitated by making use of free storage cells, often referred to as a **heap**, which are made available to the program through their addresses (“pointers”) which are in turn stored by the program in other storage cells. If the program erases pointers to a cell in all its stored places, then the cell is no longer accessible. Such a cell is called **garbage**. It can be reclaimed and reused when the program asks for more free storage.

The process of reclaiming unusable cells is called **garbage collection**. Languages like Lisp and Java provide automatic garbage collection, where the execution environment identifies and reuses space used by inaccessible cells.

The DLMSS paper [10] proposed a concurrent algorithm for automatic garbage collection, where the garbage collector runs concurrently with the user program (the **mutator**). The mutator can request for a “pointer” location to be loaded with the address of such a “new” storage cell at run time, as well as modify existing pointers, perhaps making some cell garbage while doing so. We put down the algorithm and explain it below.

```

gc  $\stackrel{def}{=} \text{begin}$ 
  const ROOT, FREE, ENDFREE, NIL: [0..N];
  var i: [0..N+1];
  for i := 0 to N do [i.colour] := white od;
  mutator || collector;
end

mutator  $\stackrel{def}{=} \text{begin}$ 
  var k, j, f, e, m: [0..N];
  do true  $\Rightarrow$  modify left edge(k, j):
    addleft(k, j);
  □ true  $\Rightarrow$  modify right edge(k, j):
    addright(k, j);
  □ true  $\Rightarrow$  get new left edge(k):
    f := [FREE.left]; e := [ENDFREE.left];
    do f = e  $\Rightarrow$  e := [ENDFREE.left] od;
    m := [f.left]; addleft(k, f); addleft(FREE, m); addleft(f, NIL);
  □ true  $\Rightarrow$  get new right edge(k): – symmetric to above
  od
end

collector  $\stackrel{def}{=} \text{begin}$ 
  var i, j: [0..N+1]; c: (white, gray, black);
  do true  $\Rightarrow$  mark; sweep
  od
end

mark  $\stackrel{def}{=} \text{atleastgrey}(\text{ROOT}); \text{atleastgrey}(\text{FREE}); \text{atleastgrey}(\text{ENDFREE});$ 
  atleastgrey(NIL); i := 0;
  do i  $\leq$  N  $\Rightarrow$  c := [i.colour];
    if c  $\neq$  gray  $\Rightarrow$  i := i+1
    □ c = gray  $\Rightarrow$ 
      restart run on gray node:
      j := i.left; atleastgrey(j);
      j := i.right; atleastgrey(j);
      [i.colour] := black;
  od

```

```

                                i := 0
                                fi
                                od;

sweep  $\stackrel{def}{=} \text{for } i := 0 \text{ until } N \text{ do}$ 
  c := [i.colour];
  if c = white  $\Rightarrow$ 
    collect white node(i):
      e := [ENDFREE.left]; [e.left] := i;
      [i.left] := NIL; [i.right] := NIL; [ENDFREE.left] := i
    □ c = black  $\Rightarrow$  [i.colour] := white
    □ c = gray  $\Rightarrow$  skip
  fi;
  i := i+1;
od

addleft(k, j)  $\stackrel{def}{=} \text{begin } [k.left] := j; \text{ atleastgrey}(j) \text{ end}$ 

atleastgrey(j)  $\stackrel{def}{=} \langle c := [j.colour]; \text{ if } c = \text{white} \Rightarrow [j.colour] := \text{gray} \square c \neq \text{white} \Rightarrow \text{fi} \rangle$ 

```

The storage potentially available to the mutator is represented as a collection of *nodes* with addresses ranging from 0 to N . (This is called the “memory”.) Each node has, in addition to whatever data is stored in it (which we completely ignore), three fields for storing a left pointer and a right pointer and a *colour*. The data used by the mutator forms a *binary data graph* within the memory using the left and right pointers, with a *ROOT* node. Every node of the data graph is reachable from *ROOT* by a path of nodes following the left or right pointers.

The collector maintains a *free list* of nodes, with a start node *FREE* and an end node *ENDFREE*. Here we see the first separation property which we will use in the proof: the data graph and the free list do not overlap.

When the mutator needs more storage, it takes a node from the free list and links it into the data graph. We call this the mutator’s **get** action. In addition, the mutator can **modify** a node’s left or right pointer to point to some other node in the data graph, or perform a **delete** action by setting the pointer to a null value. We will assume a special node called *NIL*, whose left and right nodes are always set to point to itself. Hence giving a null value to a pointer is modelled by modifying it to point to *NIL*.

When a pointer is modified, the node pointed to before the modification can become inaccessible from the data root. Such a node is called *garbage*. The separation property we mentioned above can be extended: the data graph, the free list and the garbage nodes are disjoint.

The DLMSS collector is of the “mark and sweep” type, that is, it has a *marking* phase which marks all the nodes reachable from the data root or the start of the free list, and a *sweeping* phase which puts all unmarked nodes onto the free list. The colour field of each node represents the mark: **black** means a node is marked and **white** means it is unmarked.

The basic idea behind the marking phase is that it begins by marking *ROOT* and *FREE*, and then keeps running through the memory marking the successors of marked nodes. When no marked node has an unmarked successor, all the unmarked nodes are garbage.

The sweeping phase runs through the memory, adding the nodes left unmarked by the previous marking phase to the free list and unmarking marked nodes. Note that the sweeping phase works on the garbage and the mutator works on the data graph, hence we can use separation. The movement of garbage nodes to the free list by the collector and their later reuse by the mutator constitutes an *ownership transfer* which can be modelled well in Separation Logic [19].

The DLMSS collector works all the time, *concurrently* interleaving its work with the mutator’s actions. To facilitate this, Dijkstra et al. introduced a **gray** colour intermediate between “marked”

and “unmarked”. The *ROOT* and the *FREE* nodes are first coloured gray. The marking phase makes repeated runs through the memory; when it finds a gray node, its successors are coloured gray (if they were unmarked), the node is marked by colouring it black, and a *new run* is started.

Hence, progressing from the *ROOT* and *FREE* nodes respectively, the data graph and the free list are progressively coloured black with a gray frontier while marking is in progress, and then they are unmarked (white).

2.1 Proving the DLMSS algorithm

The algorithm presented above is a rather challenging concurrent program to prove correct. The authors of [10] describe various difficulties they encountered in proving correctness. An informal proof is presented which is quite persuasive, but no indication is given as to how it could be formalized. Our formalization of the proof brings up several issues which did not receive full treatment in the original proof. Gries [12] outlined a slightly different, but formal, proof using Owicki and Gries proof system [22].

The use of resource invariants is similar to that of *global invariants*, first considered by Ashcroft [2]. A global invariant must be preserved by all processes at all times, except inside atomic actions. In return, all the actions can assume that the global invariant is true in their initial states. The proof of [10] is based on global invariants as well, even though this fact is not explicitly stated and their proof-outlines often use local assertions that deal with shared storage (in violation of the proof method).

The global invariant method can seem almost impossible at first because, unlike in Hoare Logic proof-outlines, the same assertion must hold at every program point! However, information specific to program points can be incorporated in the global invariants by adding auxiliary variables to the program and making the conditions of the global invariant depend on the values of such auxiliary variables.

The critical ideas used in the correctness proofs, right from the 1970s [10, 12], are summarized in the appendix B.

2.2 Auxiliary variables

To formalize the proof in Concurrent Separation Logic, we need to augment the algorithm with a number of auxiliary variables. These variables are added only for the purpose of the proof. They do not affect the external behavior of the program, hence they can be safely deleted after the proof is completed [21]. In order to be able to use these auxiliary variables in local assertions, in accordance with the *ATOMIC* rule given in Appendix A, we ensure that each of them is updated in at most one process. Shown below is the proof outline of the top-level algorithm along with the auxiliary variables needed:

```
gc  $\stackrel{def}{=}$ 
  const ROOT, FREE, ENDFREE, NIL: [0..N];
  var i: [0..N+1];
  auxvar scanned[0..N]: bool; tested[0..N]: bool updated by collector;
        in_marking: bool updated by collector;
        lgray, rgray: [0..N] updated by collector;
        add: [0..N] updated by mutator;
  {cells1[0..N]}
  for i := 0 to N do [i.colour] := white; tested[i] := false; scanned[i] := false od;
  in_marking := false;
  add, lgray, rgray := NIL, NIL, NIL;
  {RI  $\star$  mutI  $\star$  (colI  $\wedge$   $\neg$ in_marking  $\wedge$   $\forall i \in [0..N] : \neg$ tested[i]  $\wedge$   $\neg$ scanned[i])}
  resource r(scanned, tested, in_marking, lgray, rgray, add) in
  begin
```

```

    {mutI ★ (colI ∧ ¬in_marking ∧ ∀i ∈ [0..N] : ¬tested[i] ∧ ¬scanned[i])}
    mutator || collector
    {false ★ false}
end

```

The auxiliary boolean arrays *scanned* and *tested* keep track of which nodes have been scanned (during the marking phase) and which nodes have been tested and found to be part of the data graph. Initially, all nodes are deemed to be unscanned and untested. The boolean variable *in_marking* keeps track of the phase of the collector (true in the marking phase and false in the sweeping phase).

The two processes mutator and collector have invariants *mutI* and *colI* associated with them, which are detailed in the proofs of these processes. The address variables *add*, *lgray*, *rgray* are also explained there.

The assertion outline described above can be proved using the rules of concurrent separation logic [19] described in Appendix A. Associated with the shared storage is the resource invariant *RI*, a predicate over the (shared) states of the processes. We define the resource invariant for our proof in the next section.

Hence we are left to prove:

$$\begin{aligned}
 RI \vdash \{mutI\} \text{ mutator } \{false\} \\
 RI \vdash \{colI \wedge \neg in_marking \wedge \forall i \in [0..N] : \neg tested[i] \wedge \neg scanned[i]\} \text{ collector } \{false\}
 \end{aligned}$$

The postconditions are *false* because the processes do not terminate. The main point of the proof is to show that the mutator and the collector cooperate correctly.

3 Storage, permissions and colours

We use the auxiliary predicates defined in Table 1. Each heap node in the DLMS algorithm consists of three fields: a left pointer, a right pointer (the *link* fields) and a colour. We denote the three fields by *i.left*, *i.right* and *i.colour*. The notation $i \xrightarrow{\rho} (j, k, c)$ defines ρ *permission for a heap cell* as consisting of a read permission for all its fields, but *full* permission for the colour field whenever the variable *tested*[*i*] is false. The $\bar{\rho}$ *permission for a heap cell* is defined as just $\bar{\rho}$ permission for the link fields (and no access to the colour field). The *F permission for a heap cell* is defined in a similar way to the ρ permission. Note that:

$$i \xrightarrow{\rho} (j, k, c) \star i \xrightarrow{\bar{\rho}} (j, k, c) \iff i \xrightarrow{F} (j, k, c)$$

The predicate *cell^p* asserts a permission *p* for a cell and *cells^p* similarly asserts permission *p* for a finite set of cells. The predicate *listseg^p(j, k, V)* asserts, through a recursive definition which has a unique solution, permission *p* for a “linked list” segment of cells starting at a cell *j* and ending at an address *k*, with *V* being the finite set of all the cells making up the segment. (Such a “linked list” is built using the left links for pointing to the successor nodes and the right links set to *NIL*.)

The *edge* and *path* predicates state that there is an edge (respectively a path) between the two given nodes within the heap formed using the links (without passing through the nodes in *X*). The predicate *reachGraph^p* defines permission *p* to a directed graph (the data graph) of nodes reachable from *ROOT* (the set *U*), without passing through nodes in *X*. The reason for the exception set *X* is that while performing an operation, other nodes which we do not think of as being the data graph may temporarily be reachable from *ROOT*.

The predicate *freeList^{pqr}* describes a tripartite free list structure with permissions *p*, *q* and *r* for the three parts respectively. This predicate is admittedly complex, but a justification is provided in the next subsection. The head part of the free list, which has at most one cell, runs between *f* and *g* containing

$i \mapsto^{\rho} (j, k, c)$	$\stackrel{def}{=}$	$i.left \mapsto^{\rho} j \star i.right \mapsto^{\rho} k \star$ $((i.colour \mapsto^{\rho} c \wedge tested[i]) \vee (i.colour \mapsto^1 c \wedge \neg tested[i]))$
$i \mapsto^{\bar{\rho}} (j, k, c)$	$\stackrel{def}{=}$	$i.left \mapsto^{\bar{\rho}} j \star i.right \mapsto^{\bar{\rho}} k$
$i \mapsto^F (j, k, c)$	$\stackrel{def}{=}$	$i.left \mapsto^1 j \star i.right \mapsto^1 k \star$ $((i.colour \mapsto^{\rho} c \wedge tested[i]) \vee (i.colour \mapsto^1 c \wedge \neg tested[i]))$
$cell^p(i)$	$\stackrel{def}{=}$	$\exists j \in [0..N], k \in [0..N], c : i \mapsto^p (j, k, c)$
$cells^p(X)$	$\stackrel{def}{=}$	$\prod_{i \in X} cell^p(i)$
$listseg^p(j, k, V)$	\iff	$(j = k \wedge V = \emptyset \wedge \mathbf{emp}) \vee$ $\exists l : j \in V \wedge (j \mapsto^p (l, NIL, -) \star listseg^p(l, k, V \setminus \{j\}))$
$edge^p(j, k)$	$\stackrel{def}{=}$	$j \xrightarrow{p} (k, -, -) \vee j \xrightarrow{p} (-, k, -)$
$path^p(j, k, X)$	\iff	$j = k \vee \exists l : l \notin X \wedge edge^p(j, l) \wedge path^p(l, k, X)$
$reachGraph^p(U, X)$	$\stackrel{def}{=}$	$cells^p(U) \wedge \forall i (i \in U \equiv path^p(ROOT, i, X))$
$freeList^{pqr}(V)$	$\stackrel{def}{=}$	$\exists f, g, e, V_1, V_2, V_3 :$ $V = \{FREE, ENDFREE\} \uplus V_1 \uplus V_2 \uplus V_3 \wedge$ $(FREE \mapsto^p (f, NIL, -) \star ENDFREE \mapsto^p (e, NIL, -) \star$ $listseg^p(f, g, V_1) \star listseg^q(g, e, V_2) \star listseg^r(e, NIL, V_3)) \wedge$ $ V_1 \leq 1 \wedge (V_1 = 0 \supset V_2 = 0)$
$freeHead^p(f, g, V)$	$\stackrel{def}{=}$	$(FREE \mapsto^p (f, NIL, -) \star listseg^p(f, g, V)) \wedge V \leq 1$

Table 1: Auxiliary predicate definitions

cells V_1 , the middle part runs between g and e containing cells V_2 and the tail part runs between e and NIL containing cells V_3 . The tail part typically contains exactly one cell, but we don't require it as part of the definition. The last line says that the front part is empty only if the middle part is empty. The predicate $freeHead$ describes the head of the free list, which is a list segment of length at most 1.

3.1 Distributing the permissions

The permissions for all the heap locations must be split three-way: for the mutator, the collector and the central resource. The mutator and the collector can freely use whatever permissions they own. They can also mention these permissions in their local assertions. When accessing the resource, a process grabs the permissions for the heap cells described by the resource invariant and combines them with its own permissions using the \star connective. The permissions owned by the central resource can only be *borrowed* by the two processes in atomic actions, such permissions *cannot* be mentioned in the local assertions.

It is in general desirable to minimize the locations and permissions held in a resource because their properties must be expressed in the invariant which is not state-specific. However, Concurrent Separation Logic allows permission transfer between processes and the central resource but not directly between processes themselves. So, we are also forced to park certain permissions with the central resource until one of the processes retrieves them.

We have already noted that the memory of the algorithm can be split into three parts: the data graph, the free list and the garbage cells. Let us consider each in turn.

- The data graph's link fields can be modified only by the mutator but the collector needs read access for them to carry out marking.

We give $\bar{\rho}$ permission to these fields to the mutator and retain ρ permission in the central resource. (This allows the mutator to modify the link fields in atomic actions and the collector to read them.)

- The data graph's colour fields are modifiable by both the collector and the mutator. The mutator's modifications are limited to turning white nodes into gray.

White nodes are present in the data graph until they are marked gray during a marking scan. They are subsequently tested by the collector and identified as belonging to the data graph. We retain a full permission for the colour fields of all untested nodes in the central resource (so that both the mutator and the collector can modify them). For tested nodes, ρ permission for the colour fields is retained in the central resource and $\bar{\rho}$ permission is given to the collector.⁵

- The free list consists of at least two parts that are used in different ways. All the nodes up to the end node are used in a way similar to the data graph: their link fields can be modified only by the mutator and the colour fields by both the mutator and the collector. The end node is modified exclusively by the collector.

So, it might appear that we should give $\bar{\rho}$ permission to all but the end node to the mutator and a similar permission to the end node to the collector.

However, the collector extends the free list by adding nodes at the end of the free list and moving the *ENDFREE* pointer forward. This results in an ownership transfer for the erstwhile end node, and this transfer must be made to the central resource.

So, we split the free list into three parts:

1. The head part that contains a list segment of at most one node has its $\bar{\rho}$ permission given to the mutator and ρ permission retained in the central resource.
2. The middle part that contains all the nodes except the first and the last nodes has its full permission deposited in the central resource.
3. The tail part consisting of the end node has its $\bar{\rho}$ permission given to the collector and ρ permission retained in the central resource.

Note that there is ownership transfer: nodes are regularly moving from the tail part to the middle part and from there to the front part.

- The garbage nodes are not modifiable by either the mutator or the collector until they are reclaimed by the collector.

So, the full permission to the garbage nodes is retained in the central resource.

Using these intuitions, we now formally state the permissions given to the three components as assertions *RP*, *mutP* and *colP* respectively.

The permissions given to the central resource are defined as follows:

$$RP(U, V, W) \stackrel{def}{=} cell^F(NIL) \star freeList^{\rho F \rho}(V) \star reachGraph^{\rho}(U, V \cup \{NIL\}) \star cells^F(W)$$

The structure of the definition follows the preceding discussion. The set *W* is that of garbage cells for which the central resource has an *F* permission. It is not hard to see that if any heap satisfies *RP(U, V, W)* then there is precisely one assignment of values to *U*, *V* and *W*. Moreover, for all the cells in $U \cup V \cup W \cup \{NIL\}$, the central resource always holds at least a read permission for the colour field.

⁵One might wonder if it is necessary to treat the untested and tested nodes differently. Can we not retain a full permission for all the nodes in the central resource? Note that the collector needs to reason about the colours of the nodes that it marks. Retaining full permissions in the central resource would inhibit such local reasoning.

The permissions for the collector process include the $\bar{\rho}$ permission for the tail part of the free list and $\bar{\rho}$ permission for the colour fields of all the tested nodes:

$$\begin{aligned} colP &\stackrel{def}{=} \exists e. ENDFREE \vdash^{\bar{\rho}} (e, NIL, -) \star listseg^{\bar{\rho}}(e, NIL, -) \star tested_colours \\ tested_colours &\stackrel{def}{=} \prod_{k \in [0..N]} (\neg tested[k] \wedge \mathbf{emp}) \vee \\ &\quad (\exists c : tested[k] \wedge k.colour \vdash^{\bar{\rho}} c \wedge c \in \{g, b\}) \end{aligned}$$

Permissions for the mutator process include the $\bar{\rho}$ permission for the data graph and the head part of the free list:

$$mutP(U, V, f, g) \stackrel{def}{=} reachGraph^{\bar{\rho}}(U, \{NIL\}) \star freeHead^{\bar{\rho}}(f, g, V)$$

Even though the resource permissions allow the *reachGraph* to store pointers into the free list nodes, our mutator is written so that the *reachGraph* is self-contained. There is no conflict here, because any heap that satisfies *reachGraph*($U, \{NIL\}$) without encroaching on the free list also satisfies *reachGraph*($U, V \cup \{NIL\}$).

3.2 Colour properties and the resource invariant

The global resource invariant is given by

$$RI \stackrel{def}{=} \exists U, V, W, X: RP(U, V, W) \wedge X = \{NIL\} \cup U \cup V \wedge [0..N] = X \cup W \wedge \\ whiteI(X) \wedge grayI(X) \wedge bwI(X) \wedge blackI$$

We have already seen the *RP* predicate in the previous section. Its storage is expected to span all the cells numbered $0..N$. The symbol X denotes the set of all nodes reachable from a root: *ROOT*, *FREE*, *ENDFREE* and *NIL*. The other conjuncts of the invariant maintain several properties of the heap nodes, which are detailed next.

The collector maintains a pair of auxiliary variables *lgray* and *rgray* which record the address of a node whose left child or right child is in the process of being greyed in the marking phase. The notion of a *C*-edge used by Dijkstra et al. [10] (see also Appendix B) is defined using these variables.

$$\begin{aligned} Cedge(k, j) &\stackrel{def}{=} k = lgray \neq NIL \wedge k \xrightarrow{\rho} (j, -, -) \vee \\ &\quad k = rgray \neq NIL \wedge k \xrightarrow{\rho} (-, j, -) \end{aligned}$$

The mutator maintains the variable *add* recording the address of the target node when it is in the middle of an *addleft* or *addright* action.

White invariant: During the marking phase, every white reachable node is reachable from a gray reachable node, but without passing through a *C*-edge. During the sweeping phase, reachable nodes can be white only if their *scanned* flag is set to false.

$$\begin{aligned} whiteI(X) &\stackrel{def}{=} \forall i \in X : i.colour \xrightarrow{\rho} w \supset \\ &\quad (in_marking \supset \exists j : j \in X \wedge gwpath(j, i)) \wedge \\ &\quad (\neg in_marking \supset \neg scanned[i]) \\ gwpath(j, i) &\stackrel{def}{=} \exists k : gwedge(j, k) \wedge \neg Cedge(j, k) \wedge wpath(k, i) \\ gwedge(j, k) &\stackrel{def}{=} j \xrightarrow{\rho} (k, -, g) \vee j \xrightarrow{\rho} (-, k, g) \\ wpath(k, i) &\iff k = i \vee \exists l.wedge(k, l) \wedge \neg Cedge(k, l) \wedge wpath(l, i) \\ wedge(k, l) &\stackrel{def}{=} k \xrightarrow{\rho} (l, -, w) \vee k \xrightarrow{\rho} (-, l, w) \end{aligned}$$

Gray invariant: During the marking phase, as long as there is a gray reachable node, there must be a gray node which is unscanned. This is initially established by making all nodes unscanned.

$$\begin{aligned} grayI(X) \stackrel{def}{=} in_marking \supset (\exists i : i \in X \wedge i.colour \xrightarrow{\rho} g) \supset \\ (\exists j : j \in [0..N] \wedge j.colour \xrightarrow{\rho} g \wedge \neg scanned[j]) \end{aligned}$$

Black-to-white invariant: During the marking phase, there is at most one edge that is a black-to-white edge or a C-edge leading to a white node. Further, the source of this edge is represented by the auxiliary shared variable *add*, which is maintained by the mutator. This is initially established by colouring all the nodes white, and setting the auxiliary variables to *NIL*.

$$\begin{aligned} bwI(X) \stackrel{def}{=} in_marking \supset \\ \forall k, j \in X : (bwedge(k, j) \vee Cwedge(k, j)) \supset k = add \\ bwedge(k, j) \stackrel{def}{=} (k \xrightarrow{\rho} (j, -, b) \vee k \xrightarrow{\rho} (-, j, b)) \wedge j.colour \xrightarrow{\rho} w \\ Cwedge(k, j) \stackrel{def}{=} Cedge(k, j) \wedge j.colour \xrightarrow{\rho} w \end{aligned}$$

Black invariant: Tested nodes can be gray or black and only tested nodes can be black. The first conjunct equivalently says white nodes have to be untested, which is initially established.

$$\begin{aligned} blackI \stackrel{def}{=} (\forall i \in [0..N] : tested[i] \supset i.colour \xrightarrow{\rho} g \vee i.colour \xrightarrow{\rho} b) \wedge \\ (\forall i \in [0..N] : i.colour \xrightarrow{\rho} b \supset tested[i]) \end{aligned}$$

4 The proof

4.1 The mutator process

Consider the following mutator invariant:

$$mutI \stackrel{def}{=} \exists U, V_0, f, g : mutP(U, V_0, f, g) \wedge k \in U \wedge j \in U \cup \{NIL\}$$

In its proof, the assertion $mutI \star RP(U, V, W)$ allows the mutator to update the link fields of the nodes in *reachGraph* and the header of the free list (using the fact that $\rho \star \bar{\rho} = 1$). It can also read the colour fields of all these nodes but it can only update the colour fields of the untested nodes. However, it must do so *without mentioning the colours in its assertions*.

The top level outline of the mutator is simple, since it is a loop over its operations.

```
mutator  $\stackrel{def}{=} \mathbf{var}$  k, j, f, e, m: [0..N];
 $\mathbf{do}$  {mutI  $\wedge$  add = NIL}
  true  $\Rightarrow$  modify left edge(k, j);
   $\square$  true  $\Rightarrow$  modify right edge(k, j);
   $\square$  true  $\Rightarrow$  get new left edge(k);
   $\square$  true  $\Rightarrow$  get new right edge(k);
 $\mathbf{od}$ 
```

4.2 The collector process

The following collector invariant plays a central role in the proof of the collector process:

$$colI \stackrel{def}{=} colP \wedge lgray = NIL \wedge rgray = NIL$$

In the collector's proof, the assertion $colI \star RP(U, V, W)$ allows the process to update the link fields of the end node of the free list, to update the colour fields of tested nodes. It can also update the colour fields of untested nodes (using only RP 's full permission), but without mentioning them in its assertions. Likewise, it can access and update the remaining *garbage* nodes using the RP 's full permission.

```

collector  $\stackrel{def}{=} \mathbf{var}$  i: [0..N+1]; c: (white, gray, black);
  do true  $\Rightarrow$  {colI  $\wedge$   $\neg$ in_marking  $\wedge$   $\forall k \in [0..N] : \neg$ scanned[k]  $\wedge$   $\neg$ tested[k]}
    mark;
    {colI  $\wedge$   $\neg$ in_marking  $\wedge$   $\forall k \in [0..N] :$  scanned[k]}
    sweep
    {colI  $\wedge$   $\neg$ in_marking  $\wedge$   $\forall k \in [0..N] : \neg$ scanned[k]  $\wedge$   $\neg$ tested[k]}
  od

```

4.3 Marking phase

The marking phase of the collector is an initialization followed by a loop over the marking operations. The proof makes use of the **marking invariant**, which is a local loop invariant of the collector process.

$$markI(i) \stackrel{def}{=} colP \wedge in_marking \wedge i \in [0..N+1] \wedge \forall k \in [0..N] : (scanned[k] \equiv k < i)$$

Setting the *in_marking* flag requires us to establish the stronger version of the white invariant, viz., that every white reachable node is gray-reachable. This is achieved by greying all the root nodes initially. We set the *tested* flag of all greyed nodes to true to signify that only the collector has permission to write to the colour fields after that point. In the main loop below, this is after the collector checks that the colour is grey and *knows* that it has the permission. Such epistemic assertions are commonly found in distributed programs of this complexity.

```

mark  $\stackrel{def}{=} \{colI \wedge \neg$ in_marking  $\wedge \forall k \in [0..N] : \neg$ scanned[k]  $\wedge$   $\neg$ tested[k]}
   $\langle$ atleastgrey(ROOT); tested[ROOT] := true  $\rangle$ ;
   $\langle$ atleastgrey(FREE); tested[FREE] := true  $\rangle$ ;
   $\langle$ atleastgrey(ENDFREE); tested[ENDFREE] := true  $\rangle$ ;
   $\langle$ atleastgrey(NIL); tested[NIL] := true  $\rangle$ ;
  in_marking := true;
  i := 0;
  {markI(i)  $\wedge$  lgray = NIL = rgray}
  do i  $\leq$  N  $\Rightarrow$  {markI(i)  $\wedge$  lgray = NIL = rgray}
    with  $\langle$ c := [i.colour] |
      if c  $\neq$  gray  $\Rightarrow$  | scanned[i] := true;
        {markI(i+1)  $\wedge$  lgray = NIL = rgray}
        i := i+1
       $\square$  c = gray  $\Rightarrow$  | tested[i] := true;
        {markI(i)  $\wedge$  lgray = NIL = rgray  $\wedge$  tested[i]  $\wedge$  i.colour  $\xrightarrow{\bar{p}}$  g}
        restart run on gray node(i)
    fi
  od;
  {(colI  $\wedge$   $\forall k \in [0..N] :$  scanned[k]}
  in_marking := false
  {(colI  $\wedge$   $\neg$ in_marking  $\wedge$   $\forall k \in [0..N] :$  scanned[k]}

```

In this program block, we have used atomic conditional branching to test the colours of nodes. As a result of the initialisation $c := [i.colour]$ and the test $c \neq gray$, we conclude that i is non-gray. After

setting $scanned[i]$, the global invariants are restored (especially $grayI$). In the case where the node is gray, $tested[i]$ is set to true and the collector acquires \bar{p} permission over the colour field. A longer sequence of statements, whose proof appears later, is used to blacken the node.

The postcondition of the marking phase asserts that all nodes are scanned. Hence from the gray invariant, we get that there are no reachable gray nodes. From the white invariant, we get that all white nodes are unreachable. This is the basis for the sweeping phase.

4.4 Sweeping phase

The proof uses the sweeping invariant $sweepI(i)$, which is a loop invariant that must hold at the beginning and end of each iteration:

$$sweepI(i) \stackrel{def}{=} colI \wedge \neg in_marking \wedge i \in [0..N + 1] \wedge \\ \forall k \in [0..N] : (k < i \supset \neg scanned[k] \wedge \neg tested[k]) \wedge (k \geq i \supset scanned[k])$$

```

sweep  $\stackrel{def}{=} i := 0;$ 
  { $sweepI(i)$ }
  do  $i \leq N \Rightarrow \{sweepI(i)\}$ 
    with  $\langle c := [i.colour] \mid$ 
      if  $c = white \Rightarrow \{skip\}; \{sweepI(i)\}$ 
        collect white node(i)
       $\square \quad c = black \Rightarrow \{skip\}; \{sweepI(i) \wedge i.colour \xrightarrow{\bar{p}} b\}$ 
        whiten black node(i)
        { $sweepI(i)$ }
       $\square \quad c = gray \Rightarrow \{skip\}; \{sweepI(i) \wedge i.colour \xrightarrow{\bar{p}} g\}$ 
        skip gray node(i);
        { $sweepI(i)$ }
    fi
  od
  { $colI \wedge \neg in\_marking \wedge \forall k \in [0..N] : \neg tested[k] \wedge \neg scanned[k]$ }

```

Again atomic conditional branching is used to test node colours. If i is white, the node is added to the free list by a sequence of statements whose proof follows in the next section. On the other hand, if i is black, from the black invariant $tested[i]$ is true and the collector can assert its colour. This node is whitened, and the proof is in the next section. If i is gray, the sweeping invariant is immediately established and the collector can proceed to examine the next node.

In the sweeping phase, bwI is trivially true. From the black invariant, we have that no black nodes are left at the end of sweeping, hence no black-to-white edges either. This is the basis for the marking phase which repeats after.

5 Example proofs of operations

We illustrate how proofs of the basic operations are performed by proving a couple of key operations. *The proof of each “basic” operation requires this level of detail.*

5.1 Addleft

Table 2 shows the proof outline for addleft, a key operation in the mutator’s modification operations. We do not explain here the other key mutator action, getting a new cell from the free list. However,

addleft(p, q):

$$\begin{aligned} & \{ \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (l, m, -) \wedge p \neq NIL \wedge q \in U \cup \{NIL\} \wedge \text{add} = NIL \} \\ & \langle [p.\text{left}] := q; \text{add} := p; \rangle \\ & \{ \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (q, m, -) \wedge q \in U \cup \{NIL\} \wedge \text{add} = p \neq NIL \} \\ & \langle \text{atleastgrey}(q); \text{add} := NIL \rangle \\ & \{ \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (q, m, -) \wedge q \in U \cup \{NIL\} \wedge \text{add} = NIL \} \end{aligned}$$

get new left node(k):

$$\begin{aligned} & \{ \exists U, V : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \star \text{freeHead}^{\bar{\rho}}(-, -, V) \wedge k \in U \} \\ & f := [\text{FREE}.\text{left}]; e := [\text{ENDFREE}.\text{left}]; \\ & \{ \exists U, V : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \star \text{freeHead}^{\bar{\rho}}(f, -, V) \wedge k \in U \} \\ & \mathbf{do} f = e \Rightarrow e := [\text{ENDFREE}.\text{left}] \mathbf{od}; \\ & \{ \exists U, : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \star \text{FREE} \xrightarrow{\bar{\rho}} (f, NIL, -) \star f \xrightarrow{\bar{\rho}} (-, NIL, -) \wedge k \in U \} \\ & \{ \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \star \text{freeHead}^{\bar{\rho}}(f, -, \{f\}) \wedge k \in U \} \\ & m := [f.\text{left}]; \\ & \{ \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \star \text{freeHead}^{\bar{\rho}}(f, m, \{f\}) \wedge k \in U \} \\ & \text{addleft}(k, f); \\ & \{ \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{f, NIL\}) \star \text{freeHead}^{\bar{\rho}}(f, m, \{f\}) \wedge k \in U \} \\ & \text{addleft}(\text{FREE}, m); \\ & \{ \exists U, V : \text{reachGraph}^{\bar{\rho}}(U, \{m, NIL\}) \star \text{freeHead}^{\bar{\rho}}(m, -, V) \wedge k \in U \} \\ & \text{addleft}(f, NIL); \\ & \{ \exists U, V : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \star \text{freeHead}^{\bar{\rho}}(-, -, V) \wedge k \in U \} \end{aligned}$$

Table 2: Mutation operations

Section 6 considers the more difficult case when more than one mutator process is trying to do this action.

The proof uses the auxiliary shared variable *add*. The purpose of this variable is to ensure that the black-to-white invariant is maintained and there is at most one black-to-white edge.

The first assertion to be proved, in the context of the resource invariant *RI*, is $RI \vdash \{P\} \langle C \rangle \{Q\}$ with

$$\begin{aligned} P & \equiv \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (l, m, -) \wedge q \in U \cup \{NIL\} \wedge \text{add} = NIL \\ C & \equiv [p.\text{left}] := q; \text{add} := p; \\ Q & \equiv \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (q, m, -) \wedge q \in U \cup \{NIL\} \wedge \text{add} = p \neq NIL \end{aligned}$$

That means, we must prove $\{RI \star P\} C \{RI \star Q\}$, or, equivalently $RI \star P \supset RI \star P'$ where

$$P' \equiv \exists U : \text{reachGraph}^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (-, m, -) \wedge q \in U \cup \{NIL\} \wedge p = p \neq NIL$$

$RI \star P$ allows a combined ρ and $\bar{\rho}$ permission, that is, *F* permission to all the nodes in *U* and hence to *p.left*. The local postcondition *P'* follows immediately. We verify that *RI* is re-established in the postcondition.

- For the resource permission, the only node to worry about is *l*, the initial left child of *p*, since the edge from *p* to *l* has been removed.

- If l is reachable from $ROOT$ then the postcondition retains the $\bar{\rho}$ permission for it as part of $reachGraph^{\bar{\rho}}(U, \{NIL\})$. A read permission is left with the invariant, as required.
- If l is unreachable from $ROOT$ then the postcondition has no permission for l any more. The invariant is left with the F permission for l , which is again as required because l has been moved to the unreachable part of the heap (W).
- For the white invariant, if we are in the marking phase, we need that every white reachable node is reachable from a gray reachable node without using a C -edge. Since the edge from p to l has been removed, we must consider the case where l is a white node. (Outside the marking phase, this is not an issue and the white invariant is automatically preserved.)
 - If l continues to be reachable from $ROOT$, say via another edge (h, l) then, by $bwI \wedge (add = NIL) \wedge in_marking$ we infer that h is not black in the pre-state. It must be either gray or, if white, reachable without a C -edge from a gray node. Since h is not altered in the command, l continues to be reachable without a C -edge from a gray node in the post-state.
 - If l ceases to be reachable from $ROOT$ then $whiteI$ is not affected.
- The gray invariant $grayI$ is unaffected by the command.
- Since $add = NIL$ initially, inside the marking phase, bwI implies there is no black-to-white edge or C -edge to a white node. In the post-state there is a potential black-to-white edge, or C -edge, from p to q . However, bwI is maintained because add has been set to p .

Notice that, if l becomes unreachable, the node l silently moves in the resource invariant from $reachGraph$ into garbage. This means a *permission transfer*: the mutator retains no permissions on it in the postcondition and the invariant takes on F permission. We will see below that this will enable the collector to later sweep this node into the free list.

The next judgment to be proved, in the context of the resource invariant RI , is $RI \vdash \{Q\} \langle C' \rangle \{Q'\}$ with

$$\begin{aligned}
C' &\equiv \text{atleastgrey}(q); \text{add} := NIL \\
Q' &\equiv \exists U : reachGraph^{\bar{\rho}}(U, \{NIL\}) \wedge p \xrightarrow{\bar{\rho}} (q, m, _) \wedge q \in U \cup \{NIL\} \wedge add = NIL
\end{aligned}$$

First of all, the mutator together with RI has F permission on node q , either using its $\bar{\rho}$ and RI 's ρ , or using RI 's F . If $tested[q]$ is false, then write permission is available. If $tested[q]$ is true, the node cannot be white (using the black invariant) and in this case the read permission is enough for the execution of *atleastgrey*.

Again a little thought shows that the local conditions are easy to establish and it is re-establishing the resource invariant which requires careful argument. This time it is the gray invariant which is in danger in the postcondition if the node q was coloured white before the update, and happened to be scanned. By the white invariant, q was reachable from a gray node m , and hence by the gray invariant, there was a gray unscanned node $g \neq q$. This node is unaffected and so the gray invariant holds. The other parts of the resource invariant are easily seen to be maintained.

This proof of **adleft** is a key step in the correctness of the algorithm. It is in fact surprising that the sequence of operations

$$[p.\text{left}] := q; \text{atleastgrey}(q);$$

can possibly work because the first update operation potentially creates a black-to-white edge from p to q and one would wonder if the collector might reclaim q at this stage. It seems safer to use the opposite order of the operations:

$$\text{atleastgrey}(q); [p.\text{left}] := q;$$

Restart run on gray node(i):

```

{markI(i) ∧ i.colour  $\xrightarrow{\bar{p}}$  g}
⟨j := i.left; lgray := i⟩; atleastgrey(j);
⟨j := i.right; rgray := i⟩; atleastgrey(j);
⟨[i.colour] := black; lgray := NIL; rgray := NIL;
  for j := 0 until i-1 do scanned[j] := false od;⟩
{markI(0)}
i := 0;
{markI(i)}

```

Table 3: Marking phase operations

In fact, the early version of the DLMSS algorithm given in [9] had this sequence of operations. However, Stenning and Woodger found an error trace that showed that this version of the algorithm allowed reachable nodes to be garbage collected, invalidating the original correctness proof.

We argue that our approach of using Separation Logic with permissions makes it almost impossible to commit such an error. In order for the greying action to have any effect, one must be able to assert that q is grey as a result of the action:

$$\text{atleastgrey}(q); \{mutI \wedge q.colour \xrightarrow{p} g\} [p.left] := q;$$

However, the way we have set up the permissions, the central resource holds the full permission for all the colour fields of untested nodes. So, it is not possible to refer to the colour fields in local assertions. This might lead one to try to change the set-up of permissions so that the mutator has at least a read permission for the colour fields. But this would imply that the collector is unable to change the colour fields, so both marking and sweeping become impossible.

5.2 Restart run

Next we look at the proof outline of the action of the collector when it encounters a gray node during the marking phase. This is shown in Table 3.

Unlike the proof of the mutator, the collector has no direct permissions to the heap except for the cell *ENDFREE*. All its actions are performed by borrowing permissions from the resource invariant in atomic operations. The local invariant $markI(i)$ is easily maintained, but each step has to maintain *RI*.

Observe that for the node $l = i.left$ either $tested[l]$ is false and the collector can grey it using the full permission of the invariant, or $tested[l]$ is true and the collector can grey it putting together its \bar{p} permission with the invariant's ρ permission.

Since node i is gray and remains unscanned, the gray invariant is not violated. The white invariant holds since if a white node were gray-reachable using a path through (i, l) , it is gray-reachable from l and does not have to pass through a *C*-edge. If $i.left$ is white, then $(i, i.left)$ is a *C*-edge, so the black-to-white invariant holds. Hence P_2 holds under *RI* and $markI(i)$.

We cannot assert after greying $i.left$ that it is not white, since the mutator may modify the left pointer after the greying, perhaps to a white node, leading to $add = lgray = i$ holding. This is why the *C*-edges were introduced in [10].

The proof for greying the right child is symmetric. So let us come to the proof of the blackening step. First of all, *RI* together with the local permission $i.colour \xrightarrow{\bar{p}} g$, allows write access to $i.colour$.

The local postconditions hold, so we have to show that RI is re-established. The black invariant holds as $tested[i]$ is true. The gray invariant holds since all nodes are unscanned. The white invariant holds since if a white node was gray-reachable using a path without C -edges (and hence without the edges from i to its children), such a path is unaffected.

The black-to-white invariant holds in the post-state because if (i, l) or (i, r) is a black-to-white edge, it would have been a C -edge in the pre-state and hence $add = i$. After blackening i , the edge is a black-to-white edge with $add = i$ and hence the black-to-white invariant holds.

We do not give the detailed proof of the sweeping phase's actions but just indicate them by the proof outlines in Table 4.

Skip gray node(i):

$$\{sweepI(i) \wedge i.colour \xrightarrow{\bar{p}} g\}$$

$$tested[i] := false; scanned[i] := false; i := i+1;$$

$$\{sweepI(i)\}$$

Whiten black node(i):

$$\{sweepI(i) \wedge i.colour \xrightarrow{\bar{p}} b\}$$

$$\langle [i.colour] := white; tested[i] := false; scanned[i] := false \rangle;$$

$$\{sweepI(i+1)\}$$

$$i := i+1$$

$$\{sweepI(i)\}$$

Collect white node(i):

$$\{sweepI(i) \wedge tested[i] \wedge ENDFREE \xrightarrow{1} (-, NIL, -)\}$$

$$scanned[i] := false; tested[i] := false;$$

$$\{sweepI(i+1) \wedge ENDFREE \xrightarrow{1} (-, NIL, -)\}$$

$$[ENDFREE.left] := i;$$

$$\{sweepI(i+1) \wedge ENDFREE \xrightarrow{1} (i, NIL, -)\}$$

$$ENDFREE := i;$$

$$\{sweepI(i+1) \wedge ENDFREE \xrightarrow{1} (-, -, -)\}$$

$$[ENDFREE.left] := NIL; [ENDFREE.right] := NIL;$$

$$\{sweepI(i+1) \wedge ENDFREE \xrightarrow{1} (NIL, NIL, -)\}$$

$$i := i+1$$

$$\{sweepI(i)\}$$

Table 4: Sweeping phase operations

6 Multiple Mutators

One advantage of using a modular proof method such as ours is that it allows the components to be modified with relatively minor adaptations to the correctness proof. To illustrate how this works, we consider the modification of replacing the single mutator in our algorithm by multiple mutator processes

$mutator_1, \dots, mutator_n$, which have identical program code in our abstract treatment. Since our interest is in demonstrating how to adapt the proof, we will assume that the mutators are independent, i.e., they manipulate disjoint data graphs, each of which is reachable from a distinct root node $ROOT_i$. However, the free list is unique. So, there is potential contention among the mutators in acquiring new nodes from the free list. The procedure for acquiring new nodes needs to be more elaborate to resolve the contention.

We first consider the issue of distributing permission resources across the mutators. The permissions used for the composition of the multiple mutators are exactly the same as those used for the single mutator in the original algorithm. But since the free list is unique, its “head” needs to be shared by all the mutators. We envisage that the permission for the free list head is deposited in a “local” shared resource, separate from the central resource, so that each mutator can grab the permission to it in critical sections. This leads to a scheme of permissions such as the following:

$$\begin{aligned} mutP(U, V, f, G) &\stackrel{def}{=} (\exists U_1, \dots, U_n : U = \bigcup_{i=1}^n U_i \wedge \prod_{i=1}^n mutP_i(U_i)) \star LP(V, f, g) \\ mutP_i(U) &\stackrel{def}{=} reachGraph_i^{\bar{p}}(U, \{NIL\}) \end{aligned}$$

Here, LP stands for the permissions deposited with the local resource and $mutP_i$ stands for the permissions held by the i 'th mutator. The predicate $reachGraph_i$ denotes reachability from $ROOT_i$.

For managing the shared access to the free list head, we use an additional control variable called get with the possible values $0, 1, \dots, n$. If the value is 0, the permission to the free list head is deposited with the local resource. If it is some $i \in 1 \dots n$, then the permission is deemed to be with the mutator i . Each mutator follows a protocol whereby it waits until $get = 0$ and then atomically sets get to its own index i . To reason about the status of get in each mutator, we need an auxiliary variable in each mutator, denoted acq_i , indicating that the mutator i has acquired the permission for the free list head.

Hence, the local resource permission is defined by:

$$LP(V, f, g) \stackrel{def}{=} (\forall i : 1 \leq i \leq n. acq_i \iff get = i) \wedge ((get = 0 \wedge freeHead^{\bar{p}}(f, g, V)) \vee (1 \leq get \leq n \wedge \mathbf{emp}))$$

The central resource invariant remains essentially the same, except that it has to account for the fact that each mutator can be adding edges to its data graph. So we use a separate auxiliary variable add_i in each mutator $_i$ for recording the node to which it is adding an edge, and modify the black-to-white invariant:

$$\begin{aligned} bwI(X) &\stackrel{def}{=} in_marking \supset \\ &\quad \forall k, j \in X : (bwedge(k, j) \vee Cwedge(k, j)) \supset \exists i. k = add_i \end{aligned}$$

We now have a parallel structure of n mutators, with the i 'th mutator process maintaining the invariant $mutI_i$.

mutator $\stackrel{def}{=} \begin{array}{l} \mathbf{var} \text{ get: } [0..n] \mathbf{ updated by } \text{ each mutator}_i; \\ \mathbf{var} \text{ acq}_1 : \mathbf{ bool updated by mutator}_1; \\ \dots \\ \mathbf{var} \text{ acq}_n : \mathbf{ bool updated by mutator}_n; \\ \mathbf{resource} \text{ l(get, acq}_1, \dots, \text{acq}_n) \mathbf{ in} \\ \quad \{mutI_1 \star \dots \star mutI_n\} \\ \quad \text{mutator}_1 \parallel \dots \parallel \text{mutator}_n \\ \quad \{false \star \dots \star false\} \end{array}$

$$mutI_i \stackrel{def}{=} \exists U : mutP_i(U) \wedge k \in U \wedge j \in U \cup \{NIL\}$$

The definitions of the operations “modify left edge” and “modify right edge” as well as their correctness proofs remain exactly the same as in the single mutator case. For the operation “get new left edge,” we offer the following candidate algorithm shown in Table 5 along with the assertion annotations. This

```

get new left edge(k) in mutatori:
  {mutIi ∧ addi = NIL ∧ ¬acqi}
  do get ≠ i ⇒
    {mutIi ∧ addi = NIL ∧ ¬acqi}
    ⟨ if get = 0 ⇒ get := i; acqi := true
      □ get ≠ 0 ⇒ skip
    fi
  od;
  {(mutIi ∧ addi = NIL ∧ acqi) ★ freeHeadp(_, _, _)}
  f := [FREE.left];
  e := [ENDFREE.left];
  do f = e ⇒ e := [ENDFREE.left] od;
  m := [f.left];
  addleft(k, f);
  addleft(FREE, m);
  addleft(f, NIL)
  {(mutIi ∧ addi = NIL ∧ acqi) ★ freeHeadp(_, _, _)}
  ⟨get := 0; acqi := false⟩
  {mutIi ∧ addi = NIL ∧ ¬acqi}

```

Table 5: Get new left operation for mutator i in case of multiple mutators

is a coarse-grained solution for resolving the contention between the mutators. A mutator busy-waits until the *get* flag turns 0 and grabs the free list head by setting the flag to its own index. At this point, the mutator has access to the free list head and the usual procedure for detaching the first free node is used. By setting the *get* flag to 0 at the end, the free list head is returned to the local resource.

7 Conclusion

Separation Logic was initially conceived as a logic to conveniently reason about spatial separation of program components. However, it is slowly emerging that the notion of separation can be stretched by inventing novel kinds of components. O’Hearn [19] made the first break by treating resources and critical sections as components through which shared data can be manipulated. Still, critical sections represent a powerful barrier demarcating the separation of components. In this work, we have made an attempt to break the barrier by treating an example with fine-grained concurrency where race conditions arise in a natural (albeit controlled) way. In work done concurrently with ours, Parkinson et al [23] make another attempt at breaking the barrier by treating non-blocking algorithms.

The moral to be extracted from our exercise is that permissions play a crucial role in reasoning about such fine-grained concurrent programs. The notion of “separation of storage” gives way to one of “separation of permissions”. By controlling the permissions held by the invariant via suitable control variables, it becomes possible for processes to exchange permissions with the invariant in a sophisticated manner.

We found the exercise of proving this algorithm quite challenging. This is not surprising, given the history of the challenges posed by this algorithm. We have learnt much from the previous attempts to prove its correctness [10, 12], but our methods in turn posed their own challenges. The main difference from the proof of Gries is that our proof is based on global invariants, which is more modular than the former but less flexible in the treatment of interference between processes. This is exhibited in the number of auxiliary variables that we needed to introduce (4 scalar variables and 2 arrays) compared to the one scalar variable required in Gries’s proof. On balance, the invariant-based proof is modular and, hence, less work is involved in checking for interference between processes.

In a recent development, Vafeiadis and Parkinson [31] have found a way to combine Separation Logic and rely/guarantee reasoning (which is a modular alternative to Owicki-Gries interference handling). This should pave the way for using separation concepts along with reasoning about interference whereas, in our approach, all sharing had to be mediated by the central resource.

References

- [1] G.R. Andrews. *Concurrent programming: Principles and practice*. Addison Wesley, Menlo Park, 1991.
- [2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, Feb 1975.
- [3] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.
- [4] R. Bornat, C. Calcagno, P.W. O’Hearn, and M. Parkinson. Permission accounting in separation logic. In *Symposium on Principles of Programming Languages*, pages 59–70. ACM Press, 2005.
- [5] P. Brinch Hansen. *Operating system principles*. Prentice-Hall, Englewood Cliffs, 1973.
- [6] S.D. Brookes. A semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375(1-3):227–270, Apr 2007.
- [7] W.-P. de Roeper. *Concurrency verification: Introduction to compositional and noncompositional methods*. Cambridge University Press, Cambridge, 2001.
- [8] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.
- [9] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. Technical Report EWD496B, University of Texas, Jun 1975.
- [10] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.
- [11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [12] D. Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.
- [13] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

- [14] C.A.R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and R.H. Perrott, editors, *Operating systems techniques*, pages 61–71. Academic Press, 1972.
- [15] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–558, October 1974.
- [16] S.S. Ishtiaq and P.W. O’Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.
- [17] L. Lamport. The “Hoare Logic” of concurrent programs. *Acta Informatica*, 14:21–37, 1980.
- [18] L. Prensa Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielson and B. Rován, editors, *MFCS*, volume 1893 of *Springer Lecture Notes in Computer Science*, pages 619–628, 2000.
- [19] P.W. O’Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.
- [20] P.W. O’Hearn and D.J. Pym. The logic of bunched implications. *Bulletin Symbolic Logic*, 5(2):215–244, June 1999.
- [21] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.
- [22] S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
- [23] M. Parkinson, R. Bornat, and P.W. O’Hearn. Modular verification of a non-blocking stack. In *Principles of Programming Languages*, pages 297–302. ACM, 2007.
- [24] V.R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Symp. Found. Comp. Sci.*, pages 109–121. IEEE, 1976.
- [25] D.J. Pym. *The Semantics and Proof Theory of the Logic of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.
- [26] S. Read. *Relevant logic: A philosophical examination of inference*. Basil Blackwell, 1988.
- [27] J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davis, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*. Palgrave, Houndsmill, Hampshire, 2000.
- [28] J.C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
- [29] D.M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects Computing*, 6(4):359–390, 1994.
- [30] N. Torp-Smith, L. Birkedal, and J.C. Reynolds. Local reasoning about a copying garbage collector. *ACM Transactions on Programming Languages and Systems*, 30(4):1–58, Jul 2008.
- [31] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and Separation Logic. In *CONCUR 2007*, volume 4703 of *Springer Lecture Notes in Computer Science*, pages 256–271, 2007.

Appendix

A Review of Separation Logic

Separation Logic, formulated by Reynolds, O’Hearn and colleagues [28], is a programming logic similar to Hoare Logic with the main difference being that the assertions are written in a resource-sensitive logic that is tailored to reasoning about heap storage.

A *heap* is taken to be a partial map from location addresses (L) to values (V). A partial operation \star is defined on heaps by:

$$h_1 \star h_2 = \begin{cases} h_1 \cup h_2, & \text{if } \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset, \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Variables are assigned values using maps called *stores*. Assertions are then interpreted in contexts (s, h) consisting of a store and a heap, as follows:

$$\begin{aligned} (s, h) \models P \star Q &\iff \exists h_1, h_2. h = h_1 \star h_2 \wedge (s, h_1) \models P \wedge (s, h_2) \models Q \\ (s, h) \models \mathbf{emp} &\iff h \text{ is the empty heap} \\ (s, h) \models P \wedge Q &\iff (s, h) \models P \wedge (s, h) \models Q \\ (s, h) \models \mathbf{true} &\iff \text{always} \end{aligned}$$

The idea is that, whenever $h = h_1 \star h_2$, the heap h can be split into two disjoint partitions h_1 and h_2 , each of which can be operated upon independently by a concurrent process without interference with the other. An assertion of the form $P \star Q$ allows this fact to be recorded at the level of assertions. Note that \star and \mathbf{emp} are modal connectives, whereas \wedge and \mathbf{true} are classical connectives. Other classical connectives \vee , \mathbf{false} , \Rightarrow , \forall and \exists are also available in a similar fashion. In addition, we use an iterated form of the \star connective: if $X = \{x_1, \dots, x_n\}$ is a finite set, $\prod_{i \in X} P(i)$ means $P(x_1) \star \dots \star P(x_n)$.

For our application, we need a version of Separation Logic with *permissions* [4]. In this version, heaps are partial maps $L \rightarrow V \times P$, with P denoting the set of permissions, where P is equipped with a partial commutative semigroup structure whose operation is also denoted \star . Now, $h_1 \star h_2$ is defined iff h_1 and h_2 associate the same value for all locations $l \in \text{dom}(h_1) \cap \text{dom}(h_2)$, and

$$(h_1 \star h_2)(l) = \begin{cases} (v, p_1 \star p_2), & \text{if } h_1(l) = (v, p_1) \text{ and } h_2(l) = (v, p_2) \\ (v, p_1), & \text{if } h_1(l) = (v, p_1) \text{ and } h_2(l) \text{ undefined} \\ (v, p_2), & \text{if } h_2(l) = (v, p_2) \text{ and } h_1(l) \text{ undefined} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

In our application, we use the permission algebra $P = \{\rho, \bar{\rho}, 1\}$ with $\rho \star \bar{\rho} = 1$.

With this being the general structure of the assertion logic, we only need one non-standard atomic formula $E_1 \xrightarrow{p} E_2$, with the interpretation:

$$(s, h) \models E_1 \xrightarrow{p} E_2 \iff \text{dom}(h) = \{\llbracket E_1 \rrbracket s\} \wedge h(\llbracket E_1 \rrbracket s) = (\llbracket E_2 \rrbracket s, p)$$

The notation $i \xrightarrow{p} j$, inherited from Reynolds [28], means $(i \xrightarrow{p} j) \star \mathbf{true}$. All the other atomic formulas are insensitive to the heap, for example

$$(s, h) \models E_1 = E_2 \iff \llbracket E_1 \rrbracket s = \llbracket E_2 \rrbracket s$$

The programming logic of Separation Logic is similar to Hoare Logic except that a “tight” interpretation of the specifications is employed. We use the notation $(s, h) \xrightarrow{C} (s', h')$ to mean that the execution of a command C can transform an initial state (s, h) to the state (s', h') . It is also possible for execution starting from state (s, h) to lead to an error. This happens if C attempts to read or write to a heap location that is not defined in h . A specification $\{P\}C\{Q\}$ is *valid* iff, for all stores s and heaps h such that $(s, h) \models P$:

1. $(s, h) \not\stackrel{C}{\rightsquigarrow} \text{error}$, and
2. whenever $(s', h'). (s, h) \stackrel{C}{\rightsquigarrow} (s', h'), (s', h') \models Q$.

This means that, starting from any state satisfying the pre-condition P , the command C must execute without the possibility of error and, upon termination, results in a state satisfying Q . The precondition P must mention and assert “ownership” of all the heap locations and their permissions required for the command C to run. If that is not the case, then condition 1 of validity would be violated.

A.1 Programming language notation

Before we give the proof rules we use, we will quickly run through the programming syntax we use. The first four forms below are *atomic commands*: for fine-grained concurrency, no explicit synchronization in the form of critical sections needs to be used. Rather, we can depend on the atomicity of the hardware instructions to ensure mutual exclusion at a fine-grained level.

$$\begin{aligned}
C ::= & x := E \mid x := [E] \mid [E] := E' \mid \langle C \rangle \\
& \mid \mathbf{skip} \mid C_1; \dots; C_n \\
& \mid \mathbf{if} E_1 \Rightarrow C_1 \square \dots \square E_n \Rightarrow C_n \mathbf{fi} \\
& \mid \mathbf{do} E_1 \Rightarrow C_1 \square \dots \square E_n \Rightarrow C_n \mathbf{od} \\
& \mid \mathbf{with} \langle C_0 \mid \mathbf{if} E_1 \Rightarrow \langle C_1 \rangle; C'_1 \square \dots \square E_n \Rightarrow \langle C_n \rangle; C'_n \mathbf{fi} \\
& \mid \mathbf{with} \langle C_0 \mid \mathbf{do} E_1 \Rightarrow \langle C_1 \rangle; C'_1 \square \dots \square E_n \Rightarrow \langle C_n \rangle; C'_n \mathbf{od} \\
& \mid \mathbf{resource} r(X) \mathbf{in} C_1 \parallel \dots \parallel C_n
\end{aligned}$$

We are using the guarded command notation popularized by Dijkstra.

The **if** represents a choice between alternatives depending on the conditions $E_i, i \in \{1, \dots, n\}$ (and more than one of them might hold). If E_i holds, execution can continue with the command C_i . If none of the conditions holds, the command leads to error.

The **do** is a repeated iteration of the alternatives among the C_i , when none of the conditions E_i holds, the command terminates.

The **with** commands are a strengthening of Dijkstra’s guarded commands by adding atomic conditional branching, which is new in our syntax. Their semantics is that the initial setup command C_0 , the conditional test E_i and the corresponding initial steps of the chosen branch C_i are done atomically. After this the commands C'_i are done, but not atomically.

The **resource** command declares a resource for use within a parallel composition command.

A.2 Proof rules

The adaptation of Concurrent Separation Logic [19] to our setting involves the following modifications:

1. *All* the shared variables of concurrent processes are deemed to belong to a single implicit resource with an associated resource invariant denoted RI . We often refer to it as the “central resource.”
2. All atomic commands play the role of critical sections by grabbing the resource (consisting of *all* the shared variables) and preserving the resource invariant.
3. We also treat all individual reads and writes of variables and heap locations as implicit atomic operations without explicitly enclosing them in atomic brackets.

$$\text{ATOMIC} \quad \frac{\{P \star RI\} A \{Q \star RI\}}{RI \vdash \{P\} \langle A \rangle \{Q\}}$$

The side condition of this rule is that no other process modifies variables free in P or Q .

The proof rules for ordinary commands are traditional, except that we need two versions of each rule, one version for ordinary specifications $\{P\}C\{Q\}$ to be used inside atomic brackets, and another version for judgments of the form $RI \vdash \{P\}C\{Q\}$ to be used outside the atomic brackets. We just give the former version below. The other version can be easily inferred by generalisation.

The axioms for reading and writing heap locations are as follows:

$$\begin{array}{l} \{E \xrightarrow{p} E'\} \quad x := [E] \quad \{E \xrightarrow{p} E' \wedge x = E'\} \\ \{E \xrightarrow{1} _ \} \quad [E] := E' \quad \{E \xrightarrow{1} E'\} \end{array}$$

Note that any permission p is enough to read a heap cell (at address E), but a 1 permission is needed to write to it.

skip, sequential composition, **if** and **do** are standard.

$$\begin{array}{c} \{P\}\mathbf{skip}\{P\} \\ \\ \frac{\{P\}C_1\{Q\}, \quad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}} \\ \\ \frac{\{P \wedge E_i\}C_i\{Q\} \quad (i = 1, n)}{\{P\} \mathbf{if} E_1 \Rightarrow C_1 \square \dots \square E_n \Rightarrow C_n \mathbf{fi} \{Q\}} \\ \\ \frac{\{P \wedge E_i\}C_i\{P\} \quad (i = 1, n)}{\{P\} \mathbf{do} E_1 \Rightarrow C_1 \square \dots \square E_n \Rightarrow C_n \mathbf{od} \{P \wedge \neg E_1 \wedge \dots \wedge \neg E_n\}} \end{array}$$

For the atomic conditional commands we need new proof rules. Their side conditions are similar to those of the atomic command above.

$$\begin{array}{c} \frac{\{RI \star P\}C_0\{P'\} \quad \{P' \wedge E_i\}C_i\{RI \star Q_i\} \quad RI \vdash \{Q_i\}C'_i\{Q\} \quad (i = 1, n)}{RI \vdash \{P\} \mathbf{with} \langle C_0 \mid \mathbf{if} E_1 \Rightarrow |C_1\rangle; C'_1 \square \dots \square E_n \Rightarrow |C_n\rangle; C'_n \mathbf{fi} \{Q\}} \\ \\ \frac{\{RI \star P\}C_0\{P'\} \quad \{P' \wedge E_i\}C_i\{RI \star Q_i\} \quad RI \vdash \{Q_i\}C'_i\{P\} \quad (i = 1, n)}{RI \vdash \{P\} \mathbf{with} \langle C_0 \mid \mathbf{do} E_1 \Rightarrow |C_1\rangle; C'_1 \square \dots \square E_n \Rightarrow |C_n\rangle; C'_n \mathbf{od} \{P \wedge \neg E_1 \wedge \dots \wedge \neg E_n\}} \end{array}$$

For parallel composition of processes in the context of a shared resource, we have the rule below. This rule has the side condition that C_i should not modify any free variable of P_j or Q_j , for $j \neq i$.

$$PAR \frac{RI \vdash \{P_i\}C_i\{Q_i\} \quad (i = 1, n)}{\{RI \star P_1 \star \dots \star P_n\} \mathbf{resource} r(X) \mathbf{in} C_1 \parallel \dots \parallel C_n \{Q_1 \star \dots \star Q_n\}}$$

Finally, the tight interpretation of specifications as well as the resource-sensitive nature of assertions make possible the all-important *frame rule*:

$$FRAME \frac{\{P\}C\{Q\}}{\{P \star R\}C\{Q \star R\}}$$

with the side condition that C should not modify any free variable of R .

B The invariants used in proving the DLMSS algorithm

The algorithm presented in [10] is a rather challenging concurrent program to prove correct. We summarize the critical ideas used in the correctness proofs, right from the 1970s [10, 12].

The **black-to-white invariant** (corresponding to P1, P3 and P3a in [10]) says that there are no black-to-white edges in the graph (because all paths from black nodes to white nodes are mediated by gray nodes). Unfortunately, this invariant can be violated by the mutator actions “modify left edge(k , j)” and “modify right edge(k , j).” If k and j are the addresses of a black node and white node respectively, then the modification introduces a black-to-white edge. Even though the algorithm greys the new target node in:

modify left edge(k , j): [k.left] := j ; atleastgrey(j);

the invariant can still be falsified in between the two steps of this operation. Hence it is necessary to weaken the invariant to say that there is *at most one* black-to-white edge in the graph, and this can occur precisely when the mutator is in the middle of such a **modify** operation.

Gries’s proof [12] makes do with this weaker version of the invariant because of the way Owicki-Gries interference freedom works, but this version is not actually a global invariant. So, Dijkstra et al. define a further variation. They define a C -edge to be a gray-to-white edge which occurs in the marking phase in the midst of greying the (original) children of a node. A C -edge can turn into a black-to-white edge in the course of marking. The black-to-white invariant above is now strengthened to say that there is at most one edge that is either a black-to-white edge or a C -edge leading to a white node.

The **white invariant** (corresponding to P2 of [10]) is a consequence of the original black-to-white invariant which does not need to be weakened to account for the mutator actions: every white node is reachable from a gray node by a path consisting only of white nodes. This pretty picture of the collector can be potentially spoiled by the mutator, which can get and modify nodes, changing the data graph and the free list. Hence it can violate the collector’s invariant. However, the trick mentioned above of greying the target of every new edge created by the mutator is adequate for ensuring that the mutator maintains the white invariant.

A **gray invariant** is also used in the proof of the marking phase: if there is a gray node among those already processed by the collector during its run, then (this gray node could only have been coloured gray by the mutator and because of the white invariant it follows that) there is a gray node among those not yet processed by the collector during its run. This gray invariant is preserved by the marking actions of the collector. So the updates of the mutator and the collector’s marking phase preserve the conjunction of the white and gray invariants.