Noname manuscript No.
(will be inserted by the editor)

Fine-grained concurrency with Separation Logic

**Kalpesh Kapoor · Kamal Lodaya ·
Uday S. Reddy**

Received: date / Accepted: date

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, and extended by Lamport to handle multiple user processes.

K. Kapoor
Department of Mathematics, Indian Institute of Technology Guwahati,
Guwahati 781 039, India E-mail: kalpesh@iitg.ernet.in

K. Lodaya
The Institute of Mathematical Sciences,
C.I.T. Campus, Chennai 600 113, India
E-mail: kamal@imsc.res.in

U. S. Reddy
School of Computer Sciences, University of Birmingham,
Edgbaston, Birmingham B15 2TT, United Kingdom
E-mail: u.s.reddy@cs.bham.ac.uk

Keywords Resource logics · Separation Logic · Program correctness · Concurrent programs · Garbage collection · Heap storage

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 reason for the difficulty is that the different processes act on the same storage, causing interference. The assumptions made in one process regarding the state of storage can be invalidated by the actions done in another process. 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 [19], semaphores [9], conditional critical regions [15] and monitors [6, 16], 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 [14], 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 direct 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

[1] From a logical point of view, Hoare Logic can be thought of as a modal logic, studied by Pratt [26]. A specification $\{P\}\,C\,\{Q\}$ is interpreted as a modal formula $P \Rightarrow [C]Q$ where the modal operator $[C]$ means "after the execution of C."

their unique identifying *addresses* (also referred to as "pointers") and accessed via "indirect addressing." 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.) In contrast, practical applications of concurrency often share only dynamic storage, and routinely exchange such storage between processes and data structures.

A breakthrough in reasoning about dynamic storage was made by Reynolds [29], 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 [22,27], is a form of substructural logic — in fact, a bunched logic [28] — 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 [27].

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 [17] by adding the requirement of "tight specifications" which mention precisely the heap storage being used by a program fragment and no more, leading to local reasoning for heap storage. O'Hearn also developed a form of the logic for dealing with concurrent programs [21] with a soundness proof provided by Brookes [7]. 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.

[2] The development of this logic owes some inspiration to Girard's Linear Logic [12]. However, its structure and model-theoretic import are quite different.

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 [11], who also included its correctness proof. 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 in 1975 [10], virtually no 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 [10] 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 [24] and Gries was able to prove the correctness of the algorithm using this technique [13]. This course of events is often used in the field of concurrency to illustrate the challenges underlying concurrent programming. (See, e.g., [8].) Since the publication of this algorithm, many researchers have given alternative proofs and algorithms for concurrent garbage collection. 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 [31].

Separation Logic has been viewed as a good technique for addressing the correctness of garbage collection algorithms because it gives a tight handle on the storage accessed by a program [32]. So far, only sequential garbage collection algorithms have been treated in this way. 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 [13] 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 [20] make the point that a complete pencil and paper proof using this technique is very tedious. For this reason, many of the interference-freedom 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 reasoning about concurrent algorithms that exhibit a high degree of cooperation between processes.

2 Background

2.1 Concurrent Separation Logic

The basic Separation Logic, described in [30], incorporates Reynolds's insights for the use of separating conjunction and Ishtiaq and O'Hearn's [17] additional formulation of "tight specifications". In this logic, a specification $\{P\}C\{Q\}$ is valid only if C is able to execute *without faults* starting from any heap satisfying P. In

particular, C cannot read or write any heap cells that are not guaranteed to exist by P.

This logic admits an elegant proof rule for parallel composition of commands $C_1 \parallel C_2$:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 \star P_2\}\, C_1 \parallel C_2\, \{Q_1 \star Q_2\}} \qquad \begin{array}{l}\text{if } C_i \text{ does not modify any variable} \\ \text{in } FV(P_j, C_j, Q_j) \text{ for } i \neq j\end{array}$$

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 partitions (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 partition that satisfies P_1. In particular, it does not access any cells in the other partition. C_2 does its work similarly, in its partition. 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 [21] 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 "borrow" 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. The storage returned to the shared resource can be different from what was initially borrowed. Transfer of storage cells can take place between the resource and the "owned" storage of the process during the atomic operation. 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

[3] The use of "resource" for the packets of shared storage is inherited from Hoare [15]. It is a more specialised notion than the general logical notion of resource mentioned earlier in the Introduction.

by the proof rule below:

$$\textit{ATOMIC} \quad \frac{\{P \star R\}\, A\, \{Q \star R\}}{R \vdash \{P\}\, \langle A \rangle\, \{Q\}} \qquad \text{if no other process modifies variables in } FV(P, Q)$$

The parallel composition rule can now be generalized to

$$\frac{R \vdash \{P_1\}\, C_1\, \{Q_1\} \quad R \vdash \{P_2\}\, C_2\, \{Q_2\}}{R \vdash \{P_1 \star P_2\}\, C_1 \parallel C_2\, \{Q_1 \star Q_2\}} \qquad \text{if } C_i \text{ does not modify any variable in } FV(P_j, Q_j) \text{ for } i \neq j$$

The two parallel processes C_1 and C_2 are not independent as they were in the previous rule because they "interfere" via the shared resource. However, it is still possible to use a very similar proof rule because the interference is structured and mediated via the invariant of the shared resource.

The formal semantics of a proof system with more generous judgments $\Gamma \vdash \{P\}C\{Q\}$ as well as its soundness were described in [7]. 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.

Treatment of variables

Even though our main interest is in how to share heap storage between parallel processes, they also need to share variables (represented by variable symbols) to some extent. The conditions associated with sharing variables turn out to be a bit of nuisance and a full treatment is given in Appendix A.2. Here we indicate a brief outline of the state of affairs.

The rule for parallel composition in the previous section allows the processes in a composition $C_1 \parallel C_2$ to share variables that they only read. It does not allow one process to modify variables that are used in the other process (either for reading or writing). Resources can be used to allow other forms of sharing [15,24]. A resource is specified using a declaration of the form

$$\textbf{resource}\ r(X)\ \textbf{in}\ C$$

where r is a name given to the resource and X is a set of variables that are "protected" by the resource. In addition, a resource has an associated resource invariant R as indicated in the previous section. In any atomic block $\langle A \rangle$ occurring in the body C, the protected variables of the resource can be used for both reading and writing. The protected variables *cannot* be used outside atomic blocks.

2.2 Access Permissions

The basic Concurrent Separation Logic treats each heap location as a basic 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 heap locations but particular *access permissions*

on them. Boyland [5] and Bornat et al [4] proposed two forms of permissions suitable for this purpose, and we use a simplified form of their systems. (Our system can be thought of as an instance of "counting permissions" as well as "fractional permissions.")

A full permission on a heap 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 $\overline{\rho}$, both of which allow only reading of the location. Formally, we have a partial commutative semigroup $\{\rho, \overline{\rho}, 1\}$ with its binary operation defined by $\rho \star \overline{\rho} = 1$, undefined for all other cases. (Note that this is not a group structure because 1 is not the unit of $\star$.) A basic assertion in our logic is of the form $l \overset{p}{\longmapsto} x$ which is thought of as an agent possessing a p permission for a heap location l, which holds the value x. The semigroup operation means that, at the assertion level $l \overset{\rho}{\longmapsto} x \star l \overset{\overline{\rho}}{\longmapsto} x \equiv l \overset{1}{\longmapsto} x$.

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, say C_1 has the full permission for l then neither the shared resource nor the other process can have any access to l. The process C_1 can perform reading and writing and use facts about l in its local assertions.
- Suppose the two processes have ρ and $\overline{\rho}$ permissions for the location respectively. Then the shared resource does not have any permissions, and the two processes are restricted to reading the location.
- The third, interesting case, arises when the shared resource is given ρ permission for the location and the complement $\overline{\rho}$ 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 ρ permission from the resource. It can make local assertions about the location l using its $\overline{\rho}$ permission. For instance, it is possible to conclude a specification of the form:

  $$l \overset{\rho}{\longmapsto} \_ \vdash \{l \overset{\overline{\rho}}{\longmapsto} x\} \langle [l] := 23 \rangle \{l \overset{\overline{\rho}}{\longmapsto} 23\}$$

  using the ATOMIC rule and the fact that $l \overset{\rho}{\longmapsto} \_ \star l \overset{\overline{\rho}}{\longmapsto} x \equiv l \overset{1}{\longmapsto} x$. (We use _ as a short-hand notation for a don't-care value, which can be formalised as an existentially quantified variable). 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.

2.3 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 [21]. 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:

$$R \;\equiv\; (x = 0 \wedge \mathbf{emp}) \vee (x = 1 \wedge l \overset{\rho}{\longmapsto} \_)$$

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:

$$R \vdash \{x = 1 \wedge \mathbf{emp}\}\, \langle x := 0 \rangle\, \{x = 0 \wedge l \overset{\rho}{\longmapsto} \_\}$$

If the process already had $\overline{\rho}$ permission for l, then it is able to upgrade it to a full permission through this command:

$$R \vdash \{x = 1 \wedge l \overset{\overline{\rho}}{\longmapsto} \_\}\, \langle x := 0 \rangle\, \{x = 0 \wedge l \overset{1}{\longmapsto} \_\}$$

If, on the other hand, the process C_2 had $\overline{\rho}$ 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:

$$R \vdash \{x = 0 \wedge l \overset{\rho}{\longmapsto} \_\}\, \langle x := 1 \rangle\, \{x = 1 \wedge \mathbf{emp}\}$$

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 5.3 and 7.

3 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 [11] 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 =def begin
        const ROOT, FREE, ENDFREE, NIL: [0..N];
        var i: [0..N+1];
        [NIL.left] := NIL; [NIL.right] := NIL;
        for i := 0 to N do [i.colour] := white od;
        mutator || collector;
      end

mutator =def begin
             var k, j, f, e, m: [0..N];
             do true ⇒ k := some non-NIL node reachable from ROOT;
                       j := some node reachable from ROOT or NIL;
                       if true ⇒ modify left edge(k, j):
                                   addleft(k, j)
                       [] true ⇒ modify right edge(k, j):
                                   addright(k, j)
                       [] true ⇒ get new left edge(k):
                                   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)
                       [] true ⇒ get new right edge(k):
                                   – symmetric to the above
                       fi
             od
           end

collector =def begin
             var i, j: [0..N+1]; c: (white, gray, black);
             do true ⇒ mark; sweep od
           end

mark =def atleastgrey(ROOT); atleastgrey(FREE); atleastgrey(ENDFREE);
          atleastgrey(NIL); i := 0;
          do i ⩽ N ⇒ c := [i.colour];
                     if c ≠ gray ⇒ i := i+1
                     [] c = gray ⇒
                          restart run on gray node:
                            j := i.left; atleastgrey(j);
                            j := i.right; atleastgrey(j);
                            [i.colour]:= black;
                            i := 0
                     fi
          od

sweep =def for i := 0 to N do
             c := [i.colour];
             if c = white ⇒
                  collect white node(i):
                    e := [ENDFREE.left]; [e.left] := i;
                    [i.left] := NIL; [i.right] := NIL; [ENDFREE.left] := i
             [] c = black ⇒ [i.colour] := white
             [] c = gray ⇒  skip
             fi;
             i := i+1
           od
```

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

atleastgrey(j) $\stackrel{def}{=}$ ⟨c := [j.colour]; **if** c = white ⇒ [j.colour] := gray [] c ≠ white ⇒ **skip fi**⟩

The algorithm is written using an "Algol-like" notation. (See Appendix A.1 for a brief description.) We label some of the command blocks by bold face labels such as **modify left edge(k,j)** which are later used as abbreviations for the command blocks. The only atomic block in the algorithm is in the definition of atleastgrey(j). It can be implemented at the machine level using instructions like "test and set" or "compare and swap". All the other commands are executed without any synchronisation.

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, 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 beginning at the value of *FREE*.left and an end node at the value of *ENDFREE*.left. 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 *ROOT* and *FREE*, 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 colour gray represents an intermediate marked state.

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.

The DLMSS collector works all the time, *concurrently* interleaving its work with the mutator's actions. To facilitate this, the gray colour, intermediate between

"marked" and "unmarked," is used. It signifies a node that is known to be reachable but whose successors may not yet have been marked. The roots 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 fully marked by colouring it black, and a *new run* is started.

Hence, progressing from the root nodes, the data graph and the free list are progressively coloured black with a gray frontier while marking is in progress. During the sweeping phase, they are unmarked (white).

3.1 Proving the DLMSS algorithm

The algorithm presented above is a rather challenging concurrent program to prove correct. The authors of [11] 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 [13] outlined a slightly different, but formal, proof using Owicki and Gries proof system [24].

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 atomic actions can assume that the global invariant is true in their initial states. The proof of [11] 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 [11,13], are summarized in the Appendix B.

3.2 Augmented algorithm

The DLMSS garbage collection algorithm is remarkable in that it works with virtually no synchronization between the processes except for the atleastgrey operation, which must be carried out atomically.

However, to reason about the algorithm using Concurrent Separation Logic, we must treat each basic command that deals with shared storage as an atomic action. This would allow the use of ATOMIC rule of Section 2.1 to reason about the manipulation of shared storage. For example, whereas the original algorithm says [i.colour] := white to set i's colour, we write it as $\langle$[i.colour] := white$\rangle$ in the augmented algorithm. This makes no operational difference at the machine level because our syntax allows at most one heap location to be read or written in a single command, and this can be done without any additional synchronization mechanisms.

Secondly, our augmented algorithm adds a number of "auxiliary variables" to the algorithm meant for reasoning purposes. The need for such variables in reasoning about concurrent programs has been long recognized [23]. The auxiliary variables do not affect the original data flow or control flow of the algorithm. So, they can be safely deleted upon the completion of the proof, recovering the original algorithm. More precisely, a variable is said to be *auxiliary* for an original program P if it is not used in any control flow tests in **if** and **do** commands and it is not used on the right hand side of an assignment command $x := E$ or $[e] := E$, where x is an original variable of the program P. We specify the auxiliary variables added to the algorithm using **auxvar** declarations. We also annotate each declaration with the process that updates the variables. (Our proof rules for atomic commands have a side condition that the variables mentioned in the local assertions are not modified by other processes. This information is useful for ensuring the side condition.)

```
gc ≝ const ROOT, FREE, ENDFREE, NIL: [0..N];
     var i: [0..N+1];
     auxvar in_marking: bool updated by collector;
            scanned[0..N]: bool; tested[0..N]: bool updated by collector;
            lgray_s, lgray_t, rgray_s, rgray_t, reclaim: [0..N] updated by collector;
            ladd, radd: [0..N] updated by mutator;
            avail: bool updated by mutator;
     [NIL.left] := NIL; [NIL.right] := NIL;
     for i := 0 to N do [i.colour] := white; tested[i] := false; scanned[i] := false od;
     avail := false; ladd, radd := NIL, NIL;
     in_marking := false;
     lgray_s, lgray_t, rgray_s, rgray_t, reclaim := NIL, NIL, NIL, NIL, NIL;
     resource r(scanned, tested, in_marking, lgray_s, lgray_t, rgray_s, rgray_t, reclaim,
                avail, ladd, radd)
     in
         mutator || collector

mutator ≝ begin
            var k, j, f, e, m: [0..N];
            do true ⇒
                k := some non-NIL node reachable from ROOT;
                j := some node reachable from ROOT or NIL;
                if true ⇒ modify left edge(k, j):
                            addleft(k, j)
                [] true ⇒ modify right edge(k, j):
                            addright(k, j)
                [] true ⇒ get new left edge(k):
                            ⟨f := [FREE.left]⟩;
                            ⟨e := [ENDFREE.left]; avail := (f ≠ e)⟩;
                            do f = e ⇒ ⟨e := [ENDFREE.left]; avail := (f ≠ e)⟩ od;
                            ⟨m := [f.left]⟩;
                            addleft(k, f);
                            ⟨addleft(FREE, m); avail := false⟩;
                            addleft(f, NIL)
                [] true ⇒ get new right edge(k):
                            – symmetric to the above
                fi
            od
          end

addleft(p, q) ≝ begin ⟨[p.left] := q; ladd := p⟩; ⟨atleastgrey(q); ladd := NIL⟩ end

atleastgrey(j) ≝ ⟨c := [j.colour]; if c = white ⇒ [j.colour] := gray [] c ≠ white ⇒ skip fi⟩
```

```
collector =def begin
               var i, j: [0..N+1]; c: (white, gray, black);
               do true ⇒ mark; sweep od
             end

mark =def begin
          ⟨atleastgrey(ROOT); tested[ROOT] := true⟩;
          ⟨atleastgrey(FREE); tested[FREE] := true⟩;
          ⟨atleastgrey(ENDFREE); tested[ENDFREE] := true⟩;
          ⟨atleastgrey(NIL); tested[NIL] := true⟩;
          ⟨in_marking := true⟩;
          i := 0;
          do i ⩽ N ⇒
               atomic ⟨c := [i.colour] |
               if  c ≠ gray ⇒ | scanned[i] := true⟩; i := i+1
               ▯   c = gray ⇒ | tested[i] := true⟩;
                                restart run on gray node(i):
                                    ⟨j := i.left; lgray_s := i; lgray_t := j ⟩;
                                    ⟨atleastgrey(j); lgray_t := NIL⟩;
                                    ⟨j := i.right; rgray_s := i; rgray_t := j ⟩;
                                    ⟨atleastgrey(j); lgray_t := NIL⟩;
                                    ⟨[i.colour] := black; lgray_s := NIL; rgray_s := NIL;
                                     for j := 0 to i-1 do scanned[j] := false od⟩;
                                    i := 0;
               fi
          od;
          ⟨in_marking := false⟩
        end

sweep =def begin
            i := 0;
            do i ⩽ N ⇒
                 atomic ⟨c := [i.colour] |
                 if c = white ⇒ | reclaim := i⟩;
                                  collect white node(i):
                                    ⟨scanned[i] := false⟩;
                                    ⟨[i.left] := NIL⟩; ⟨[i.right] := NIL⟩;
                                    ⟨e := [ENDFREE.left]⟩;
                                    ⟨[e.left] := i; reclaim := NIL⟩;
                                    ⟨[ENDFREE.left] := i⟩
                 ▯ c = black ⇒ | skip⟩;
                                  whiten black node(i):
                                    ⟨[i.colour] := white; tested[i] := false; scanned[i] := false⟩
                 ▯ c = gray ⇒  | skip⟩;
                                  skip gray node(i):
                                    ⟨tested[i] := false; scanned[i] := false⟩
                 fi;
                 i := i+1;
             od
          end
```

The auxiliary variables we add have one of two purposes. Some of them are used to capture control flow information so that global invariants can state properties that must hold in specific regions of the code. The others are used to capture an abstract view of the processing done in one process so that it can be used in reasoning about the other process.

– The boolean variable *in_marking* captures control information. It is set to true during the marking phase of the collector and false outside the marking phase.

- The boolean arrays *scanned* and *tested* indexed by heap nodes can be viewed as finite sets of nodes and provide an abstraction of the processing done in the collector. The array *scanned* captures the nodes that have been scanned during the marking phase of the collector. The array *tested* represents the nodes that are tested during the marking phase and found to be at least gray. Both of these arrays are progressively reset to false during the sweeping phase.
- The variables $lgray_s$, $lgray_t$, $rgray_s$, $rgray_t$ mark the source and target nodes of edges that are traversed and coloured in the marking phase. They are set to *NIL* after the colouring is completed.
- The variable *reclaim* is used by the collector to mark an unreachable node that it is about to reclaim. It is reset to *NIL* after the reclamation.
- The variables *ladd* and *radd* mark the source of an edge that has been just added by the mutator. After the add action, the mutator greys the target node, at which point the variables are reset back to *NIL*.
- The variable *avail* represents information about the availability of nodes in the free list. It is an abstraction of the nodes manipulated by the collector, but used inside the mutator.

In addition to the usual forms of commands, we have used atomic conditional branching and (later in Section 7) atomic iteration constructs of the form:

$$\begin{array}{ll}
\textbf{atomic } \langle C_0 | & \textbf{atomic } \langle C_0 | \\
\textbf{if } E_1 \Rightarrow |C_1\rangle; C_1' & \textbf{do } E_1 \Rightarrow |C_1\rangle; C_1' \\
[\!] \ \ldots & [\!] \ \ldots \\
[\!] \ E_n \Rightarrow |C_n\rangle; C_n' & [\!] \ E_n \Rightarrow |C_n\rangle; C_n' \\
\textbf{fi} & \textbf{od}
\end{array}$$

These constructs *atomically* execute a setup operation C_0, the chosen condition test E_i and the atomic part of the corresponding conditional branch C_i. (The remainder of the chosen branch C_i' is executed outside the atomic action.) The semantics and proof rules for the constructs are given in Appendix A. If each $C_i = \textbf{skip}$ and each E_i involves only local variables of the process then the atomic branching construct can be simplified to the following sequences using standard **if** and **do**:

$$\begin{array}{ll}
\langle C_0 \rangle; & \langle C_0 \rangle; \\
\textbf{if } E_1 \Rightarrow C_1' \ [\!] \ \ldots \ [\!] \ E_n \Rightarrow C_n' \ \textbf{fi} & \textbf{do } E_1 \Rightarrow C_1' \ [\!] \ \ldots \ [\!] \ E_n \Rightarrow C_n' \ \textbf{od}
\end{array}$$

The reason is that, since the expressions E_i involve only local variables, they are not affected by other processes and give the same values outside the atomic brackets as they do inside.

It may be verified that the augmented algorithm can be transformed back to the original one after the removal of the auxiliary variables. The code for atleastgrey is the only place in the program where atomicity is required, which matches the granularity of the program we originally considered.

4 Storage, permissions and colours

Assertions in Separation Logic are resource-sensitive. Hence, they must delineate (permissions for) an area of storage in addition to stating properties that must

$$
\begin{array}{rcl}
i \overset{\rho}{\longmapsto} (j,k,c) & \stackrel{def}{=} & i \in [0\,..\,N] \;\wedge \\
& & (i.left \overset{\rho}{\longmapsto} j \star i.right \overset{\rho}{\longmapsto} k \star \\
& & ((i.colour \overset{\rho}{\longmapsto} c \wedge tested[i]) \vee (i.colour \overset{1}{\longmapsto} c \wedge \neg tested[i]))) \\
i \overset{\overline{\rho}}{\longmapsto} (j,k,c) & \stackrel{def}{=} & i.left \overset{\overline{\rho}}{\longmapsto} j \star i.right \overset{\overline{\rho}}{\longmapsto} k \\
i \overset{F}{\longmapsto} (j,k,c) & \stackrel{def}{=} & i \in [0\,..\,N] \;\wedge \\
& & (i.left \overset{1}{\longmapsto} j \star i.right \overset{1}{\longmapsto} k \star \\
& & ((i.colour \overset{\rho}{\longmapsto} c \wedge tested[i]) \vee (i.colour \overset{1}{\longmapsto} c \wedge \neg tested[i]))) \\
i \overset{1}{\longmapsto} (j,k,c) & \stackrel{def}{=} & i.left \overset{1}{\longmapsto} j \star i.right \overset{1}{\longmapsto} k \star i.colour \overset{1}{\longmapsto} c \\
cell^p(i) & \stackrel{def}{=} & \exists j \in [0..N],\, k \in [0..N],\, c : i \overset{p}{\longmapsto} (j,k,c) \\
cells^p(X) & \stackrel{def}{=} & \circledast_{i \in X}\, cell^p(i) \\
listseg^p(j,k,V) & \iff & (j = k \wedge V = \varnothing \wedge \mathbf{emp}) \vee \\
& & (j \neq k \wedge \exists l :\; j \in V \wedge (j \overset{p}{\longmapsto} (l, NIL, \_) \star listseg^p(l,k,V \backslash \{j\}))) \\
edge^p(j,k) & \stackrel{def}{=} & j \overset{p}{\longmapsto} (k,\_,\_) \vee j \overset{p}{\longmapsto} (\_,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 \iff 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 \overset{p}{\longmapsto} (f, NIL, \_) \star ENDFREE \overset{p}{\longmapsto} (e, NIL, \_) \star \\
& & listseg^p(f,g,V_1) \star listseg^q(g,e,V_2) \star listseg^r(e, NIL, V_3)) \wedge \\
& & (avail \wedge |V_1| = 1 \vee \neg avail \wedge |V_1| = 0) \\
freeHead^p(f,g,V) & \stackrel{def}{=} & (FREE \overset{p}{\longmapsto} (f, NIL, \_) \star listseg^p(f,g,V)) \wedge \\
& & (avail \wedge |V| = 1 \vee \neg avail \wedge |V| = 0)
\end{array}
$$

Table 1 Auxiliary predicate definitions

be satisfied by the contents of such storage. Assertions that precisely delineate a set of storage locations and permissions for them are termed "precise" assertions. At the other extreme, assertions that continue to hold when the heap extended with additional locations or permissions are termed "intuitionistic" assertions. (See Appendix A for precise definitions.) Recall that an assertion of the form $P \wedge Q$ holds for a heap iff both P and Q hold for the same heap. We arrange our assertions as conjunctions of the form $P \wedge Q$ where P is a precise assertion delineating locations and permissions and Q is an intuitionistic assertion that states the properties required for these locations, with the result that the whole formula is precise.

We use the auxiliary predicates defined in Table 1. Each heap node in the DLMSS 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 $i.colour$. For readability, we abuse the notations $\longmapsto$ and $\hookrightarrow$ to apply to entire *heap nodes* (in addition to individual fields of the nodes). We also annotate these notations with new *composite permissions* for heap nodes, denoted ρ, $\overline{\rho}$ and F. The notation $i \overset{\rho}{\longmapsto} (j,k,c)$ indicates 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 $\overline{\rho}$ permission for a heap cell $i \overset{\overline{\rho}}{\longmapsto} (j,k,\_)$ is defined as just the $\overline{\rho}$ 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 so that

these definitions satisfy:

$$i \overset{\rho}{\longmapsto} (j,k,c) \star i \overset{\overline{\rho}}{\longmapsto} (j,k,c) \iff i \overset{F}{\longmapsto} (j,k,c)$$

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*.) Note that the set of nodes spanned by the predicate V is used as a parameter to the predicate. The use of such parameters avoids certain anomalies in the use of Separation Logic formulas with permissions. (See, e.g., [4].)

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), but 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 part of the data graph may temporarily be reachable from $ROOT$ and we need to exclude them.

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 free list is viewed as consisting of three segments, running between f to g, g to e and e to *NIL* (with respective sets of nodes V_1, V_2 and V_3). The first segment, called the "head" of the free list, contains at most one cell and its length is controlled by the variable *avail*. The third segment, called the "tail" of the free list, is identified by *ENDFREE*. It is normally of length 1, but this is not required in the definition. The predicate $freeHead$ describes just the head of the free list.

Note that the predicates $edge^p$ and $path^p$ are intuitionistic and all other predicates defined in Table 1 are precise. For instance $reachGraph^p(U,X)$ is precise because $cells^p(U)$ is precise. More interestingly, $\exists U : reachGraph^p(U,X)$ is also precise because U is uniquely determined as the set of all nodes reachable from $ROOT$ without passing through X.

4.1 Distributing the permissions

The permissions for every heap node is split three-way: between the mutator process, the collector process and the central resource. That this split happens differently at different program states is a key idea in the structure of the proof. The mutator and the collector can use whatever permissions they "own" in a state. 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 in order to read or write a heap location. But, 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.* As a specific example, there

are points in the program where the function atleastgrey(p) can be called by both the mutator and the collector to colour the cell p. At such points, 1 permission for p's colour field resides with the central resource, and can be borrowed by either process. But, as a consequence, the colour field of p is absent from both processes' local assertions.

It is in general desirable to minimize the locations and permissions held in the central resource because their properties must be expressed in the resource 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 forced to park permissions with the central resource until one of the processes retrieves them for itself, typically by setting an auxiliary variable.

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 $\overline{\rho}$ 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 greyed 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 $\overline{\rho}$ permission is given to the collector.*[4]
- 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 $\overline{\rho}$ 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 can only 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 $\overline{\rho}$ permission given to the mutator and ρ permission retained in the central resource.*

[4] One might wonder if it is necessary to treat the untested and tested nodes differently. Is it not possible to retain a full permission for all the colour fields 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.

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 $\overline{\rho}$ 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.*

In addition to the three major regions of storage, there are two transient areas at the borders, which need special treatment controlled through auxiliary variables. The head of the free list, lying at the border of the free list and data graph during allocation of nodes, is controlled by the variable *avail*. During the collection phase, a chosen node of the garbage area lies at the border of garbage cells region and the free list, whose status is controlled by the variable *reclaim*.

Using these intuitions, we now formally state the permissions given to the three components as *precise* 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.

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

$$colP \stackrel{def}{=} \exists e.\, ENDFREE \stackrel{\overline{\rho}}{\longmapsto} (e, NIL, \_) \star e \stackrel{\overline{\rho}}{\longmapsto} (\_,\_,\_) \star tested\_colours$$
$$tested\_colours \stackrel{def}{=} \circledast_{k \in [0..N]} (\neg tested[k] \wedge \mathbf{emp}) \vee (\exists c : tested[k] \wedge k.colour \stackrel{\overline{\rho}}{\longmapsto} c \wedge c \in \{g, b\})$$

Recall that the atomic operations in a process can use the permissions of the process as well as the permissions of the central resources. So, the effective permissions for the collector process are as follows:

Remark 1 The assertion $RP(\_,\_,\_) * colP$ includes F permission for all the garbage cells and the tail of the free list and ρ permission for the rest of the nodes in the free list and the data graph. It also includes 1 permission for the colour fields of all tested nodes.

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

$$mutP(U,V) \stackrel{def}{=} reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(\_,\_,V)$$

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

Remark 2 The assertion $RP(\_,\_,\_) \star mutP(\_,\_)$ includes F permission for all the nodes in the data graph and all nodes but the tail of the free list.

4.2 Colour properties and the resource invariant

The global resource invariant is given by

$$\begin{aligned} RI \stackrel{def}{=} \exists U,V,W,X\colon\ & RP(U,V,W) \wedge X = \{NIL\} \cup U \cup V \wedge \\ & [0..N]\backslash\{i \mid i = reclaim \neq NIL\} = X \cup W \wedge \\ & whiteI(X) \wedge grayI(X) \wedge bwI(X) \wedge blackI \end{aligned}$$

We have already seen the RP predicate in the previous section. Its storage is expected to span all the cells numbered $0,\ldots,N$ except for the cell *reclaim* (whenever it is not *NIL*). The symbol X denotes the set of all nodes reachable from a root: *ROOT*, *FREE*, *ENDFREE* and *NIL*. The other conjuncts of the invariant, *whiteI* etc., are intuitionistic assertions that maintain several properties of the heap nodes, which are detailed in the sequel.

Lemma 1 *RI is precise.*

Proof Suppose $(s,h) \models RI$. Let R stand for the value of $\{i \mid i = reclaim \neq NIL\}$ in the current store. Then $(s,h) \models RP(U,V,W)$ and it can be seen from the structure of RP that $\mathrm{dom}(h) = U \uplus V \uplus W$. So, $\mathrm{dom}(h) = [0\ldots N]\backslash R$. For each node in this domain, the permission for the link fields is determined by RP and that for colour fields is uniquely determined by the value of *tested* in the store. □

Recall, from Sec. 3.2, that the collector maintains a set of auxiliary variables $lgray_s$, $lgray_t$, $rgray_s$ and $rgray_t$ which record the source and target information about the edges being coloured in the marking phase. We define a notion of *C-edge*, similar to the one used by Dijkstra et al. [11] using these variables.

$$\begin{aligned} Cedge(k,j) \stackrel{def}{=}\ & k = lgray_s \neq NIL \wedge k \stackrel{\rho}{\longrightarrow} (j,\_,\_) \wedge (lgray_t \neq NIL \Rightarrow j \neq lgray_t) \vee \\ & k = rgray_s \neq NIL \wedge k \stackrel{\rho}{\longrightarrow} (\_,j,\_) \wedge (rgray_t \neq NIL \Rightarrow j \neq rgray_t) \end{aligned}$$

The idea is that, if $(lgray_s, lgray_t)$ is a putative edge being coloured by the collector, then an edge starting at the same source but a different target is a C-edge. Such an edge might result from the mutator changing the target of the edge without the knowledge of the collector.

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

$$\begin{aligned} whiteI(X) \quad & \stackrel{def}{=} \quad \forall i \in X : i.colour \stackrel{\rho}{\longrightarrow} w \Rightarrow \\ & \qquad (in\_marking \Rightarrow \exists j : j \in X \wedge propath(j,i)) \wedge \\ & \qquad (\neg in\_marking \Rightarrow \neg scanned[i]) \\ propath(j,i) \quad & \stackrel{def}{=} \quad \exists k : j.colour \stackrel{\rho}{\longrightarrow} g \wedge edge(j,k) \wedge \neg Cedge(j,k) \wedge wpath(k,i) \\ wpath(k,i) \quad & \iff \quad k = i \vee \\ & \qquad \exists l : k.colour \stackrel{\rho}{\longrightarrow} w \wedge edge(k,l) \wedge \neg Cedge(k,l) \wedge wpath(l,i) \end{aligned}$$

(The predicate *wpath* asserts the existence of a path through white nodes, while the predicate *propath* captures the notion of a propagation path.)

*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.

$$grayI(X) \stackrel{def}{=} in\_marking \Rightarrow (\exists i : i \in X \wedge i.colour \xhookrightarrow{\rho} g) \Rightarrow (\exists j : j \in [0..N] \wedge j.colour \xhookrightarrow{\rho} g \wedge \neg scanned[j])$$

*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 one of the auxiliary shared variables *ladd* and *radd*, which are maintained by the mutator. This is initially established by colouring all the nodes white, and setting the auxiliary variables to *NIL*.

$$\begin{aligned} bwI(X) &\stackrel{def}{=} in\_marking \Rightarrow \\ &\quad \neg(ladd \neq NIL \wedge radd \neq NIL) \wedge \\ &\quad (\forall k, j \in X : bwedge(k,j) \vee Cwedge(k,j) \Rightarrow add(k,j)) \\ bwedge(k,j) &\stackrel{def}{=} (k \xhookrightarrow{\rho} (j, \_, b) \vee k \xhookrightarrow{\rho} (\_, j, b)) \wedge j.colour \xhookrightarrow{\rho} w \\ Cwedge(k,j) &\stackrel{def}{=} Cedge(k,j) \wedge j.colour \xhookrightarrow{\rho} w \\ add(k,j) &\stackrel{def}{=} (k = ladd \wedge k.left \xhookrightarrow{\rho} j) \vee (k = radd \wedge k.right \xhookrightarrow{\rho} j) \end{aligned}$$

*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.

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

5 The proof

A proof outline for the top-level of the augmented algorithm can be written as follows:

$\{cells^1[0..N]\}$
[NIL.left] := NIL; [NIL.right] := NIL;
for i := 0 **to** N **do** [i.colour] := white; *tested*[i] := false; *scanned*[i] := false **od**;
avail := false; *ladd*, *radd* := NIL, NIL;
in_marking := false;
$lgray_s$, $lgray_t$, $rgray_s$, $rgray_t$, *reclaim* := NIL, NIL, 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*, $lgray_s$, $lgray_t$, $rgray_s$, $rgray_t$, *reclaim*,
avail, *ladd*, *radd*) **in**
begin
$\{mutI\}$ mutator $\{false\}$ ||
$\{colI \wedge \neg in\_marking \wedge (\forall i \in [0..N] : \neg tested[i] \wedge \neg scanned[i])\}$ collector $\{false\}$
end
$\{false\}$

The initial pre-condition asserts 1 permission for all the cells $0, \ldots, N$. After the initialisation steps, the three assertions RI, $mutI$ and $colI$ along with the additional conditions are claimed to hold in separate contexts of the heap permissions. (The assertions $mutI$ and $colI$ are $mutP(\_, \_)$ and $colP$ along with additional conditions for the auxiliary variables, defined in the sequel.) The permissions of $cells^1[0..N]$ are split three-way, along the lines described in Section 4.1.

It is worth checking each of the properties involved in the resource invariant hold after the initialisation steps. The white invariant holds because *in_marking* is false and the *scanned* flag is false for all nodes. The gray invariant holds because there are no gray nodes. The black-to-white invariant holds because there are no black nodes or C-edges. The black invariant holds because there are no tested nodes or black nodes.

The **resource** block and the parallel composition split the assertion into three parts, for the central resource, the mutator and the collector, requiring us to prove:

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

(The post-conditions are *false* because the processes do not terminate.) Proving these statements amounts to proving the *safety* of the collector. Since $mutI$ and $colI$ are disjoint and $mutI$ includes $\overline{\rho}$ permission for the link fields of all the data graph nodes, it means that the collector does its work without altering the data graph. (There is a separate *liveness* property that can be stated to the effect that any unreachable node is eventually put on the free list by the collector. However, our proof technique is *not* meant for addressing liveness.)

Note that the entire processes of mutator and collector are within the scope of the **resource** declaration. Hence, their proof outlines are written in the context of the resource invariant RI. It is only inside atomic blocks inside these processes that the invariant participates in the pre- and post-conditions.

5.1 The mutator process

The proof outline of the mutator is constructed using the following mutator invariant:

$$mutI \stackrel{def}{=} \exists U, V_0 : mutP(U, V_0) \wedge ladd = NIL = radd \wedge \neg avail$$

Using Remark 2, the assertion $mutI \star RI$ has F permission for the data graph and the head part of the free list, allowing the mutator to update the link fields of these nodes in atomic operations. The mutator 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 any updates of the colour fields *without mentioning the colours in its assertions.*

For the mutator, we need to prove the following proof outline in the context of the resource invariant RI:

mutator $\stackrel{def}{=}$ **var** k, j, f, e, m: [0..N];
{$mutI$}
do true ⇒ k := some non-NIL node reachable from ROOT;
j := some node reachable from ROOT or NIL;

$\{\exists U, V_0 : mutP(U, V_0) \wedge ladd = NIL = radd$
$\wedge\ k \in U \wedge j \in U \cup \{NIL\}$
if true ⇒ modify left edge(k, j)
[] true ⇒ modify right edge(k, j)
[] true ⇒ get new left edge(k)
[] true ⇒ get new right edge(k)
fi
od

Here, in the sequel, we use the labels such as "modify left edge(k,j)" as abbreviations for command blocks in the augmented algorithm (cf. Sec. 3.2).

Proof outlines for the operations of the mutator are shown in Table 2. Since the operation addleft is used in both **modify left edge** and **get new left edge**, we prove a more general specification for its definition. The predicate *reach* is defined by:

$$reach(q) \stackrel{def}{=} \exists x : edge^{\overline{\rho}}(x, q) \vee x = NIL$$

So, the pre-condition of addleft says that in the storage delineated by $mutP(U, V)$, there is a node p and some path to reach address q (unless it is *NIL*), and the *ladd* and *radd* variables are *NIL*. The post-condition says very much the same thing except for noting that p's left link has been modified to q. The specification can be strengthened to that of **modify left edge** using the structural rules of Separation Logic. (Cf. Appendix A.2.) A full discussion of the proof of addleft is given in Sec. 6.1.

5.2 The collector process

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

$$colI \stackrel{def}{=} colP \wedge lgray = (NIL, NIL) = rgray \wedge reclaim = NIL$$

For brevity, we treat $lgray = (lgray_s, lgray_t)$ and $rgray = (rgray_s, rgray_t)$ as pairs in our assertions.

As per Remark 1, the assertion $colI \star RI$ allows the collector process to update the link fields of the end node of the free list, and to update the colour fields of tested nodes. It can also update the colour fields of untested nodes (using only RI's 1 permission), but without mentioning them in its assertions. Likewise, it can access and update the remaining *garbage* nodes using the RI's full permission. The following proof outline needs to be proved for the collector process in the context of the resource invariant RI:

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

addleft(p, q):
$\{\exists U, V : mutP(U,V) \wedge p \xrightarrow{\overline{\rho}} (l, m, \_) \wedge reach(q) \wedge ladd = NIL = radd\}$
$\langle$[p.left] := q; *ladd*:= p$\rangle$;
$\{\exists U, V : mutP(U,V) \wedge p \xrightarrow{\overline{\rho}} (q, m, \_) \wedge reach(q) \wedge ladd = p \wedge radd = NIL\}$
$\langle$atleastgrey(q); *ladd*:= NIL$\rangle$
$\{\exists U, V : mutP(U,V) \wedge p \xrightarrow{\overline{\rho}} (q, m, \_) \wedge reach(q) \wedge ladd = NIL = radd\}$
modify left edge(k, j):
$\{\exists U, V_0 : mutP(U, V_0) \wedge ladd = NIL = radd \wedge k \in U \wedge j \in U \cup \{NIL\} \wedge \neg avail\}$
addleft(k, j)
$\{\exists U, V_0 : mutP(U, V_0) \wedge ladd = NIL = radd \wedge k \in U \wedge j \in U \cup \{NIL\} \wedge \neg avail\}$
get new left edge(k):
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(\_, \_, \varnothing) \wedge k \in U \wedge \neg avail\}$
$\langle$f := [FREE.left]$\rangle$;
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, \_, \varnothing) \wedge k \in U \wedge \neg avail\}$
$\langle$e := [ENDFREE.left]; *avail* := (f $\neq$ e)$\rangle$;
$\{\exists U, V : reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, \_, V) \wedge k \in U \wedge avail = (f \neq e)\}$
do f = e $\Rightarrow$ $\langle$e := [ENDFREE.left]; *avail* := (f $\neq$ e)$\rangle$ **od**;
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, \_, \{f\}) \wedge k \in U \wedge avail\}$
$\langle$m := [f.left]$\rangle$;
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, m, \{f\}) \wedge k \in U \wedge avail\}$
addleft(k, f);
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{f, NIL\}) \star freeHead^{\overline{\rho}}(f, m, \{f\}) \wedge k \in U \wedge avail\}$
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{f, NIL\}) \star FREE \xmapsto{\overline{\rho}} (f, NIL, \_) \star f \xmapsto{\overline{\rho}} (m, NIL, \_) \wedge avail\}$
$\langle$addleft(FREE, m); *avail* := false$\rangle$;
$\{\exists U : (reachGraph^{\overline{\rho}}(U, \{m, NIL\}) \wedge f \xrightarrow{\overline{\rho}} (m, NIL, \_)) \star FREE \xmapsto{\overline{\rho}} (m, NIL, \_) \wedge \neg avail\}$
$\{\exists U : (reachGraph^{\overline{\rho}}(U, \{m, NIL\}) \wedge f \xrightarrow{\overline{\rho}} (m, NIL, \_)) \star freeHead^{\overline{\rho}}(m, \_, \varnothing) \wedge \neg avail\}$
addleft(f, NIL);
$\{\exists U : reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(\_, \_, \varnothing) \wedge \neg avail\}$

Table 2 Mutation operations

At the end of the marking phase, all nodes are scanned. Hence, by gray invariant, we can conclude that there are no reachable gray nodes. Then the white invariant implies that there no reachable white nodes. This enables the sweeping phase to reclaim all the white nodes. At the end of the sweeping phase, the loop invariant is re-established. (Our proof outlines do not show that all the white nodes have been reclaimed because we are only proving the safety of the garbage collector.)

5.3 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.

$$\begin{aligned} markI'(i) &\stackrel{def}{=} colP \wedge in\_marking \wedge reclaim = NIL \wedge \\ &\qquad i \in [0..N+1] \wedge \forall k \in [0..N] : (scanned[k] \iff k < i) \\ markI(i) &\stackrel{def}{=} markI(i) \wedge lgray = (NIL, NIL) = rgray \end{aligned}$$

Setting the *in_marking* flag requires us to establishes the stronger version of the white invariant, viz., that every white reachable node is reachable via a propagation path. This is achieved by greying all the root nodes initially. The assertion "k is a root" in the proof outline below means that k is one of $ROOT$, $FREE$, $ENDFREE$, and NIL. Setting the *tested* flag of all greyed nodes to true allows the collector to retrieve the $\overline{\rho}$ permission for their colour fields from the central resource and prohibits the mutator from further modifying the colours. The retrieved permissions are absorbed into $colP$.

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$\rangle$;
$\{colI \wedge \neg in\_marking \wedge (\forall k \in [0..N] : \neg scanned[k] \wedge (k \text{ is a root} \Rightarrow tested[k]))\}$
$\langle in\_marking := \text{true}\rangle$;
$\{colI \wedge in\_marking \wedge (\forall k \in [0..N] : \neg scanned[k])\}$
$\{markI(0) \wedge lgray = (NIL, NIL) = rgray\}$
i := 0;
$\{markI(i)\}$
do i $\leqslant$ N $\Rightarrow$
$\{markI(i) \wedge i \leqslant N\}$
atomic $\langle$c := [i.colour] |
if c $\neq$ gray $\Rightarrow$ | $scanned$[i] := true$\rangle$;
$\{markI(i) \wedge i \leqslant N \wedge scanned[i]\}$
$\{markI(i+1) \wedge i \leqslant N\}$
i := i+1
[] c = gray $\Rightarrow$ | $tested$[i] := true$\rangle$;
$\{markI(i) \wedge i \leqslant N \wedge \wedge tested[i] \wedge i.colour \xhookrightarrow{\overline{\rho}} g\}$
restart run on gray node(i)
fi
od;
$\{markI(N+1)\}$
$\{colI \wedge in\_marking \wedge \forall k \in [0..N] : scanned[k]\}$
$\langle in\_marking := \text{false}\rangle$
$\{colI \wedge \neg in\_marking \wedge \forall k \in [0..N] : scanned[k]\}$

In this program block, we have used atomic conditional branching to test the colours of nodes. So, the resource invariant is assumed at the beginning of every atomic sequence and needs to be re-established at the end of the sequence. Consider the sequence

$\langle$c := [i.colour] | c $\neq$ gray $\Rightarrow$ | scanned[i] := true$\rangle$

As a result of the initialisation $c := [i.colour]$ and the test $c \neq gray$, we conclude that i is non-gray. Since the collector does not have any permission for the colour fields of untested nodes, this information about the colour will be lost at the end of the atomic sequence. So we set $scanned[i]$ to true, note that the the resource invariants are restored (especially $grayI$), and assert that $scanned[i]$ is true. In the case where the node is gray, $tested[i]$ is set to true and the collector acquires

$\overline{\rho}$ permission over the colour field. So, it is possible to assert the fact about the colour in the post-condition. A longer sequence of statements, whose proof appears in Table 3, is used to blacken the node and restart the scan.

Restart run on gray node(i):

$\{markI'(i) \wedge i \leqslant N \wedge tested[i] \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} g \wedge lgray = (NIL, NIL) = rgray\}$
$\langle$j := [i.left]; $lgray_s$:= i; $lgray_t$:= j$\rangle$;

$\{markI'(i) \wedge i \leqslant N \wedge tested[i] \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} g \wedge lgray = (i, j) \wedge rgray = (NIL, NIL)\}$
$\langle$atleastgrey(j); $lgray_t$:= NIL$\rangle$;

$\{markI'(i) \wedge i \leqslant N \wedge tested[i] \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} g \wedge lgray = (i, NIL) \wedge rgray = (NIL, NIL)\}$
$\langle$j := [i.right]; $rgray_s$:= i; $rgray_t$:= j$\rangle$;
$\langle$atleastgrey(j); $rgray_t$:= NIL$\rangle$;

$\{markI'(i) \wedge i \leqslant N \wedge tested[i] \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} g \wedge lgray = (i, NIL) = rgray\}$
$\langle$[i.colour] := black; $lgray_s$:= NIL; $rgray_s$:= NIL;
for j := 0 **to** i-1 **do** *scanned*[j] := false **od**$\rangle$;

$\{markI'(0) \wedge i \leqslant N \wedge tested[i] \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} b \wedge lgray = (NIL, NIL) = rgray\}$
i := 0;
$\{markI'(i) \wedge lgray = (NIL, NIL) = rgray\}$

Table 3 Marking phase operations

The post-condition 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.

5.4 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:

$$\begin{aligned} sweepI(i) \overset{def}{=}\ & colI \wedge \neg in\_marking \wedge lgray = (NIL, NIL) = rgray \wedge \\ & i \in [0..N+1] \wedge \\ & \forall k \in [0..N] : (k < i \Rightarrow \neg scanned[k] \wedge \neg tested[k]) \wedge (k \geqslant i \Rightarrow scanned[k]) \end{aligned}$$

At stage i, the *scanned* and *tested* flags are expected to have been set to false for all the nodes $0, \ldots, (i-1)$.

sweep $\overset{def}{=}$ $\{sweepI(0) \wedge reclaim = NIL\}$
i := 0;
$\{sweepI(i) \wedge reclaim = NIL\}$
do i $\leqslant$ N $\Rightarrow$
$\{sweepI(i) \wedge reclaim = NIL \wedge i \leqslant N\}$
atomic $\langle$c := [i.colour] |
if c = white $\Rightarrow$ | $reclaim$:= i$\rangle$;
$\{i \overset{1}{\longmapsto} (\_, \_, w) \star sweepI(i) \wedge reclaim = i \wedge i \leqslant N\}$
collect white node(i)

$\{sweepI(i) \wedge reclaim = NIL\}$
[] c = black ⇒ $\langle$ **skip** $\rangle$;
$\{sweepI(i) \wedge reclaim = NIL \wedge i \leqslant N \wedge i.colour \xleftarrow{\overline{\rho}} b\}$
whiten black node(i)
$\{sweepI(i) \wedge reclaim = NIL\}$
[] c = gray ⇒ $\langle$ *scanned*[i] := false; *tested*[i] := false$\rangle$;
$\{sweepI(i+1) \wedge reclaim = NIL \wedge i \leqslant N\}$
i := i+1
$\{sweepI(i) \wedge reclaim = NIL\}$
fi
od
$\{sweepI(N+1) \wedge reclaim = NIL\}$
$\{colI \wedge \neg in\_marking \wedge \forall k \in [0..N] : \neg tested[k] \wedge \neg scanned[k]\}$

Again atomic conditional branching is used to test node colours. If i is white, it is a garbage cell and the central resource has 1 permission for it. In this case, the variable *reclaim* is set to i inside the atomic sequence, which removes the 1 permission from the central resource and releases it to the collector. So, we assert the permission in the post-condition of the atomic sequence. Next, the node is added to the free list by a sequence of statements whose proof is shown in Table 4.

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 also shown in Table 4. If i is gray, which is somewhat of an unusual occurrence, we set the *scanned* and *tested* flags to false, which immediately extends the sweeping invariant to the node i.

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.

6 Example proofs of operations

Tables 2, 3 and 4 show the proof outlines of the detailed operations of the mutator and collector. Considerable reasoning is involved in proving that these proof outlines are valid. We illustrate the reasoning by giving detailed proofs of the operations addleft used in the mutator, and the "restart run on gray node" action of the collector.

6.1 Addleft

Table 2 shows the proof outline for addleft, which constitutes the entire operation of "modify left edge" and also used in "get new left edge" several times. We prove the validity of this proof outline.

The proof uses the auxiliary shared variable *ladd*. 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.

Collect white node(i):

$\{sweepI(i) \star i \overset{1}{\longmapsto} (\_, \_, w) \wedge i \leqslant N \wedge reclaim = i)\}$
$\langle scanned[\mathrm{i}] := \mathrm{false}\rangle$;
$\{sweepI(i+1) \star i \overset{1}{\longmapsto} (\_, \_, w) \wedge i \leqslant N \wedge reclaim = i\}$
$\langle[\mathrm{i.left}] := \mathrm{NIL}\rangle$; $\langle[\mathrm{i.right}] := \mathrm{NIL}\rangle$;
$\{sweepI(i+1) \star i \overset{1}{\longmapsto} (NIL, NIL, w) \wedge i \leqslant N \wedge reclaim = i\}$
$\langle \mathrm{e} := [\mathrm{ENDFREE.left}]\rangle$;
$\{sweepI(i+1) \star i \overset{1}{\longmapsto} (NIL, NIL, w) \wedge i \leqslant N \wedge reclaim = i \wedge ENDFREE \overset{\overline{\rho}}{\longrightarrow} (e, NIL, \_)\}$
$\langle[\mathrm{e.left}] := \mathrm{i};\ reclaim := \mathrm{NIL}\rangle$;
$\{sweepI(i+1) \wedge i \leqslant N \wedge reclaim = NIL$
$\quad \wedge ENDFREE \overset{\overline{\rho}}{\longrightarrow} (e, NIL, \_) \wedge e \overset{\overline{\rho}}{\longrightarrow} (i, NIL, \_) \wedge i \overset{\overline{\rho}}{\longrightarrow} (NIL, NIL, \_)\}$
$\langle[\mathrm{ENDFREE.left}] := \mathrm{i}\rangle$;
$\{sweepI(i+1) \wedge i \leqslant N \wedge reclaim = NIL \wedge ENDFREE \overset{\overline{\rho}}{\longrightarrow} (i, NIL, \_) \wedge i \overset{\overline{\rho}}{\longrightarrow} (NIL, NIL, \_)\}$
$\{sweepI(i+1) \wedge reclaim = NIL\}$
i := i+1
$\{sweepI(i) \wedge reclaim = NIL\}$

Whiten black node(i):

$\{sweepI(i) \wedge reclaim = NIL \wedge i \leqslant N \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} b\}$
$\langle[\mathrm{i.colour}] := \mathrm{white};\ scanned[\mathrm{i}] := \mathrm{false};\ tested[\mathrm{i}] := \mathrm{false}\rangle$;
$\{sweepI(i+1) \wedge reclaim = NIL \wedge i \leqslant N\}$
i := i+1
$\{sweepI(i) \wedge reclaim = NIL\}$

Table 4 Sweeping phase operations

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

$$P \equiv \exists U, V : mutP(U, V) \wedge p \overset{\overline{\rho}}{\longrightarrow} (l, m, \_) \wedge q \in (U \cup V \cup \{NIL\}) \wedge ladd = NIL = radd$$
$$C \equiv ([\mathrm{p.left}] := \mathrm{q};\ ladd := \mathrm{p})$$
$$Q \equiv \exists U, V : mutP(U, V) \wedge p \overset{\overline{\rho}}{\longrightarrow} (q, m, \_) \wedge q \in (U \cup V \cup \{NIL\}) \wedge radd = p \wedge radd = NIL$$

That means, we must prove $\{RI \star P\}C\{RI \star Q\}$ using sequential Separation Logic with permissions.

Note that $mutP$ is a precise assertion whose domain includes all the nodes reachable from $ROOT$ except NIL and the head node of the free list. The sets of these nodes are captured in the logical variable U and V respectively. The node p must be included among these nodes. The assertion P has $\overline{\rho}$ permission to the nodes in $U \cup V$ and RI has ρ permission to them. So, $RI \star P$ has F permission. Thus, it is permissible for C to alter $p.left$.

We verify that RI is re-established in the post-condition.

- For the resource permission RP, note that the only node whose status is changed is l, the initial left child of p, since the edge from p to l has been removed.
  - If l is reachable via some other path then the post-condition retains the $\overline{\rho}$ permission for it as part of $mutP(U, V)$. A read permission is left with the resource invariant, as required.

  - If l becomes unreachable then the post-condition has no permission for l any more. The resource 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 via a propagation path. 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, say via another edge (h, l) then, by $bwI \wedge (ladd = NIL = radd) \wedge in\_marking$ we infer that h is not black in the initial state. It must be either gray or, if white, reachable via a propagation path. Since h is not altered in the command, l continues to be reachable via propagation path in the final state.
  - If l ceases to be reachable in the final state then the white invariant is not affected.
- The gray invariant $grayI$ is unaffected by the command.
- Since $ladd = NIL = radd$ 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 $ladd$ 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 post-condition and the resource invariant takes on F permission. This enables the collector to later sweep this node into the free list.)

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

$$Q \equiv \exists U, V : mutP(U,V) \wedge p \xrightarrow{\overline{\rho}} (q, m, \_) \wedge q \in (U \cup V \cup \{NIL\}) \wedge ladd = p \wedge radd = NIL$$
$$C' \equiv (\text{atleastgrey(q)};\ ladd := \text{NIL})$$
$$R \equiv \exists U, V : mutP(U,V) \wedge p \xrightarrow{\overline{\rho}} (q, m, \_) \wedge q \in (U \cup V \cup \{NIL\}) \wedge ladd = NIL = radd$$

As before, $RI \star Q$ has F permission for node q, either by combining the $\overline{\rho}$ and ρ permissions from the two conjuncts or using RI's F. This gives the command a 1 permission for the colour field of q in case $tested[q]$ is false. If, on the other hand, $tested[q]$ is true, the node cannot be white (using the black invariant) and, in this case, the ρ permission available in RI is enough for the execution of $atleastgrey$.

The local post-condition R is easily established because $ladd$ has been set to NIL. Re-establishing the resource invariant requires a careful argument.

- The resource permission RP is unaffected by the command.
- The white invariant is preserved. If q is initially white and any propagation path passed through q, then the path can be replaced by the suffix that just begins at q, because q is gray in the final state.
- The gray invariant is affected if the node q was white before the greying action, and happened to be scanned. However, by the white invariant, q is reachable from a gray node g in the initial state, and hence by the gray invariant, there is a gray unscanned node $g' \neq q$. The node g' is unaffected by the greying of q and so the gray invariant continues to hold in the final state.

– As for the black-to-white invariant, the edge (p, q) in the initial state is a potential black-to-white edge or a C-edge to a white node, since $ladd = p$. Since it is the only edge of this kind, greying q and setting $ladd = NIL$ restores the invariant. □

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

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

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:

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

In fact, the early version of the DLMSS algorithm given in [10] had this sequence of operations and most computer scientists seem to believe, at first sight, that it is safe. However, it is faulty. 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.[5]

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 be useful, one must be able to assert that q is gray as a result of the greying action:

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

However, as argued in Sec. 4.1, the mutator cannot have any permissions for the colour fields of nodes. So, it is not possible to refer to the colour fields in the local assertions of the mutator. The techniques of Concurrent Separation Logic provide a formal framework to help one avoid such serious pitfalls in reasoning.

6.2 Restart run

We look at the proof outline of the action "Restart run on gray node" carried out by the collector when it encounters a gray node during the marking phase. This is shown in Table 3.

Unlike the mutator, the collector has no direct permissions to the link fields of nodes other than the tail of the free list. It has $\overline{\rho}$ permission to the colour fields of tested nodes. All its actions are performed by borrowing permissions from the resource in atomic operations.

Proof The first proof segment to be proved is $RI \vdash \{P_1\}\langle C_1\rangle\{P_2\}$ with

$$P_1 \equiv markI(i) \wedge tested[i] \wedge i.colour \xrightarrow{\overline{\rho}} g \wedge lgray = (NIL, NIL) = rgray$$
$$C_1 \equiv (\text{j := [i.left]; } lgray_s := \text{i; } lgray_t := \text{j})$$
$$P_2 \equiv markI(i) \wedge tested[i] \wedge i.colour \xrightarrow{\overline{\rho}} g \wedge lgray = (i, j) \wedge rgray = (NIL, NIL)$$

[5] The reason that the seemingly unsafe order of operations above works correctly is very subtle. The node q is a reachable node before assigning it as the left successor of p and, so, it has an independent propagation path as per the white invariant.

$RI \star markI(i)$ has at least a read permission for the link fields and, so, it is permissible to read the left link of i. The local post condition P_2 clearly holds in the final state and the resource invariant is not affected.

The second proof segment to be proved is $RI \vdash \{P_2\}\, C_2\, \{P_3\}$ with

$$C_2 \equiv (\text{atleastgrey(j)};\ lgray_t := \text{NIL})$$
$$P_2 \equiv markI(i) \wedge tested[i] \wedge i.colour \xhookrightarrow{\overline{\rho}} g \wedge lgray = (i, NIL) \wedge rgray = (NIL, NIL)$$

$RI \star markI(i)$ has 1 permission for the colour fields of reachable nodes and, so, it is permissible to grey i. The local post-condition is immediate. So, it remains to show that RI is re-established in the final state.

If l is the initial left successor of i (i.e., $i.left \hookrightarrow l$) then either $l = j = lgray_t$ or, if the mutator has modified the target of the edge, $l \neq j$ and (i, l) is a C-edge. If $l = j$, the white invariant requires that all reachable nodes should have a propagation path that does not pass through C-edges. The operation atleastgrey(j) makes this possible. If there was a propagation path in the initial state that passed through (i, j) then, in the final state, it can be replaced by the suffix of the path beginning at j, since j is now gray. Hence the white invariant is preserved. If, on the other hand, $l \neq j$, then the white invariant is not affected because propagation paths do not pass through the C-edge (i, l).

Since the node i is gray and remains unscanned, the gray invariant is preserved. The black-to-white invariant is preserved because, even though a new C-edge (i, j) is potentially created by C_2, its target is non-white. If the left successor of i is some other node l then (i, l) would have been a C-edge already in the initial state, and would have satisfied the black-to-white invariant.

(We cannot assert after greying j that it is not white, since we can't establish that j is tested. In fact, j may not be the left successor of i any more. The mutator may modify the left pointer after the greying, perhaps to a white node, leading to $ladd = lgray_s = i$ holding. This is why the notion of C-edges was introduced in [11]. The black-to-white invariant captures the maximum information that can be assumed at this point.)

The proof for greying the right child is symmetric. So let us come to the proof of the blackening step. We must show $RI \vdash \{P_5\}\, \langle C_5 \rangle\, \{P_6\}$ with

$$P_5 \equiv markI(i) \wedge tested[i] \wedge i.colour \xhookrightarrow{\overline{\rho}} g \wedge lgray = (i, NIL) = rgray$$
$$C_5 \equiv ([\text{i.colour}] := \text{black};\ lgray_s := \text{NIL};\ rgray_s := \text{NIL};$$
$$\quad \textbf{for}\ \text{j} := 0\ \textbf{to}\ \text{i-1}\ \textbf{do}\ scanned[\text{j}] := \text{false}\ \textbf{od})$$
$$P_6 \equiv markI(0) \wedge tested[i] \wedge i.colour \xhookrightarrow{\overline{\rho}} b \wedge lgray = (NIL, NIL) = rgray$$

$RI \star markI(i)$ allows write access to $i.colour$, just as in the case of C_2 above. The local post-condition $markI(0)$ holds in the final state because all the *scanned* flags have been set to false. It remains to show that RI is re-established.

Let l and r stand for the initial left and right successors of i respectively. Since $lgray_s = i = rgray_s$, both (i, l) and (i, r) are C-edges in the initial state. The black-to-white invariant says that there is at most one edge that is a black-to-white edge or a C-edge to a white node and, further, the source of this edge is one of *ladd* and *radd*. So, at most one of l and r is white and, if one of them (say l) is white, then $ladd = i$. Since i is black in the final state, (i, l) has turned into a black-to-white edge with $ladd = i$, as allowed by the black-to-white invariant. Setting $lgray_s$

and $rgray_s$ to NIL removes the C-edges, but does not affect the black-to-white invariant. If both l and r are non-white, the black-to-white invariant is preserved trivially.

The black invariant is preserved as $tested[i]$ is true. The gray invariant holds in the final state since all nodes are unscanned. (The nodes that have been freshly greyed in this operation might have been previously scanned. So, those scans are now obsolete and a fresh scan is warranted.) The white invariant is preserved since no edges are changed or turned into C-edges in this step.

The last step of the operation is a straightforward assignment to a local variable and does not affect the resource invariant. □

7 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 $mutator_1, \ldots, mutator_n$ [18], 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, that is, 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 total 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:

$$\begin{aligned} mutP(U,V) &\stackrel{def}{=} (\exists U_1,\ldots,U_n : U = \textstyle\bigcup_{i=1}^{n} U_i \ \wedge\ \prod_{i=1}^{n} mutP_i(U_i)) \star LP(V,\_,\_) \\ mutP_i(U) &\stackrel{def}{=} reachGraph_i^{\overline{\rho}}(U,\{NIL\}) \end{aligned}$$

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 $0, 1, \ldots, n$. If the value is 0, the permission to the free list head is deposited with the local resource. If it is some $i \in 1 \ldots n$, 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, to indicate that the mutator i has acquired the permission for the free list head.

Hence, the local resource permission is defined by:

$$LP(V,f,g) \stackrel{def}{=} (\forall i \in [1..n] : acq_i \iff get = i) \wedge \Big((get = 0 \wedge freeHead^{\overline{\rho}}(f,g,V)) \vee (1 \leqslant get \leqslant n \wedge \mathbf{emp})\Big)$$

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 set of auxiliary variables $ladd_i$ and $radd_i$ in each mutator$_i$ for recording the node to which it is adding an edge, and modify the black-to-white invariant as follows:

$$\begin{aligned} bwI(X) \stackrel{def}{=}\ & in\_marking \Rightarrow \\ & \forall k,j \in X : (bwedge(k,j) \vee Cwedge(k,j)) \Rightarrow \\ & \qquad \exists i.\, (k = ladd_i \wedge k.left \xhookrightarrow{\rho} j) \vee (k = radd_i \wedge k.right \xhookrightarrow{\rho} j) \end{aligned}$$

We now have a parallel structure of n mutators, with the i'th mutator process maintaining the invariant $mutI_i \stackrel{def}{=} \exists U : mutP_i(U) \wedge ladd_i = NIL = radd_i$.

mutator $\stackrel{def}{=}$
var get: [0..n] **updated by** mutator$_1$,. . . ,mutator$_n$;
auxvar acq_1 : bool **updated by** mutator$_1$;
. . .
auxvar acq_n : bool **updated by** mutator$_n$;
get := 0; acq_1 := false; . . . acq_n := false;
resource l(get, acq_1, . . . , acq_n) **in**
$\{mutI_1 \star \cdots \star mutI_n\}$
mutator$_1$ || . . . || mutator$_n$
$\{false \star \cdots \star false\}$

where

$$mutI_i \stackrel{def}{=} \exists U : mutP_i(U) \wedge ladd_i = NIL = radd_i$$

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 algorithm shown in Table 5 along with the assertion annotations. This 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. We use a simple version of the atomic iterative command for this purpose (see Appendix A.1 for the full syntax). Its proof rule (Appendix A.2) allows us to derive the local assertion acq_i from the exit condition which tests the shared variable *get*, and in fact *get* is never mentioned in the local assertions of $mutator_i$, satisfying the side conditions of the atomic iteration and the parallel composition with the other mutators. Following Lamport [18], we note that this is only the second instance, after that in atleastgrey, where an atomic command (after removal of auxiliary variables) includes a test and set of a shared variable, and thus needs hardware support.

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.

get new left edge(k) in mutator$_i$:
$\{mutI_i \wedge \neg acq_i\}$
$\{(\neg acq_i \wedge (mutI_i \star \mathbf{emp})) \vee (acq_i \wedge (mutI_i \star freeHead^{\overline{\rho}}(\_,\_,\_)))\}$
atomic do $get = 0 \Rightarrow$ | get := i; acq_i := true$\rangle$
$[\![get \notin \{0, i\} \Rightarrow$ | **skip**$\rangle$ **od**;
$\{acq_i \wedge (mutI_i \star freeHead^{\overline{\rho}}(\_,\_,\_))\}$
$\langle$f := [FREE.left]$\rangle$;
$\langle$e := [ENDFREE.left]; $avail$:= (f $\neq$ e)$\rangle$;
do f = e $\Rightarrow \langle$e := [ENDFREE.left]; $avail$:= (f $\neq$ e)$\rangle$ **od**;
$\langle$m := [f.left]$\rangle$;
addleft(k, f);
$\langle$addleft(FREE, m); $avail$:= false$\rangle$;
addleft(f, NIL);
$\{acq_i \wedge (mutI_i \star freeHead^{\overline{\rho}}(\_,\_,\_))\}$
$\langle$get := 0; acq_i := false $\rangle$
$\{mutI_i \wedge \neg acq_i\}$

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

8 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 [21] 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 [25] 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 [11,13], 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. Our proof extended to handle multiple mutators, as was informally done by Lamport [18], without changing a single assertion inside the collector process, a slight addition to the resource invariant and a modification of the mutator code for interacting with the free list. Ours are the first global invariant proofs (rather than ones using interference-freedom) of these two concurrent garbage collection programs within a formal proof system.

In a recent development, Vafeiadis and Parkinson [33] 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.

Acknowledgements The second author thanks the School of Computer Science, University of Birmingham, for hosting a visit during May-June 2003, which laid the ground for this work. The authors would like to thank the referees for their extensive and detailed reports. The authors are also grateful to the editor, Hans van Ditmarsch, for his extraordinary patience.

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 Lamguages 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. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th Intern. Symp.*, volume 2694 of *Springer Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.
6. P. Brinch Hansen. *Operating system principles.* Prentice-Hall, Englewood Cliffs, 1973.
7. S.D. Brookes. A semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375(1-3):227–270, Apr 2007.
8. W.-P. de Roever. *Concurrency verification: Introduction to compositional and noncompositional methods.* Cambridge University Press, Cambdridge, 2001.
9. E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.
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. Technical Report EWD496B, University of Texas, Jun 1975.
11. 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.
12. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
13. D. Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.
14. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
15. 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.
16. C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–558, October 1974.
17. 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.
18. L. Lamport. Garbage Collection with Multiple Processes: An Exercise in Parallelism. In *Parallel Processing*, pages 50–54, 1976.
19. L. Lamport. The "Hoare Logic" of concurrent programs. *Acta Informatica*, 14:21–37, 1980.
20. L. Prensa Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielson and B. Rovan, editors, *MFCS*, volume 1893 of *Springer Lecture Notes in Computer Science*, pages 619–628, 2000.
21. P.W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.

22. P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin Symbolic Logic*, 5(2):215–244, June 1999.
23. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.
24. S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
25. 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.
26. V.R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Symp. Found. Comp. Sci.*, pages 109–121. IEEE, 1976.
27. 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.
28. S. Read. *Relevant logic: A philosophical examination of inference*. Basil Blackwell, 1988.
29. 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.
30. J.C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
31. D.M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects Computing*, 6(4):359–390, 1994.
32. N. Torp-Smith, L. Birkedal, and J.C. Reynolds. Local reasoning about a copying garbage collector. *ACM Transactions on Programming Lamguages and Systems*, 30(4):1–58, Jul 2008.
33. 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 [30], 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) with $L \subseteq V$. A partial operation $\star$ is defined on heaps by:

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

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

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

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 **emp** are "modal" connectives (interpreting their subformulas in contexts different from the current context), whereas $\wedge$ and **true** are classical connectives. Other classical connectives $\vee$, **false**, $\Rightarrow$, $\forall$ and $\exists$ are also available in a similar fashion. In addition, we use an iterated form of the $\star$ connective: if $X = \{x_1, \ldots, x_n\}$ is a finite set, $\circledast_{i \in X} P(i)$ means $P(x_1) \star \cdots \star P(x_n)$.

For our application, we need a version of Separation Logic with *permissions* [4]. In this version, heaps are partial maps $L \rightharpoonup V \times P$, with P denoting the set of permissions, where P is equipped with a partial cancellative commutative semigroup structure whose operation is also

denoted $\star$. Now, $h_1 \star h_2$ is defined iff $(h_1 \star h_2)(l)$ is defined for all $l \in \mathrm{dom}(h_1) \cup \mathrm{dom}(h_2)$ as per the following rule:

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

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

Expressions occurring in Separation Logic formulas are normal mathematical expressions over the variable symbols that are assigned values in stores. The valuation of an expression E in store s is denoted $[\![E]\!]s$. Note that the values of expressions *do not* depend on the heap. Normal atomic formulas are likewise insensitive to heap. For example: All the other atomic formulas are insensitive to the heap, for example

$$(s, h) \models E_1 = E_2 \iff [\![E_1]\!]s = [\![E_2]\!]s$$

We only need one non-standard atomic formula $E_1 \overset{p}{\longmapsto} E_2$ which is heap-dependent. Its interpretation is:

$$(s, h) \models E_1 \overset{p}{\longmapsto} E_2 \iff \mathrm{dom}(h) = \{[\![E_1]\!]s\} \wedge h([\![E_1]\!]s) = ([\![E_2]\!]s,\, p)$$

The notation $E_1 \overset{p}{\hookrightarrow} E_2$, inherited from Reynolds [30], means $(E_1 \overset{p}{\longmapsto} E_2) \star \mathit{true}$. Whereas $E_1 \overset{p}{\longmapsto} E_2$ implies that the heap contains *exactly* one location with the address E_1, $E_1 \overset{p}{\hookrightarrow} E_2$ implies that the heap contains *at least* one location with address E_1.

The definition of $\star$ on heaps induces a notion of a *subheap*

$$h_1 \leqslant h_2 \iff \exists h' : h_1 \star h' = h_2$$

An assertion P is said to be *precise* if, whenever $(s, h) \models P$, there is no subheap $h' \leqslant h$ such that $(s, h') \models P$. Examples of precise assertions include **emp**, $E_1 \overset{p}{\longmapsto} E_2$, $P \star Q$ whenever both P and Q are precise, and $P \wedge Q$ whenever either P or Q is precise. $P \vee Q$ is in general not precise. An assertion P is said to be *intuitionistic* if, whenever $(s, h) \models P$, for all $h' \geqslant h$, $(s, h') \models P$. Examples of intuitionistic assertions include true, $E_1 \overset{p}{\hookrightarrow} E_2$, $P \star Q$ whenever P or Q is intuitionistic, and $P \wedge Q$ whenever both P and Q are intuitionistic. An assertion P is said to be *pure* if it is heap-independent, i.e., $(s, h) \models P$ implies $(s, h') \models P$ for all heaps h'.

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

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, result 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.

In Concurrent Separation Logic, we deal with parallel execution of commands. The parallel executions are interleaved respecting the atomic brackets used inside the commands, and they access shared resources for which suitable invariants are expected to be preserved. A specification $r_1(X_1) : R_1, \ldots, r_n(X_n) : R_n \vdash \{P\}C\{Q\}$ is valid if, starting in a state where the shared resources $r_1, \ldots, r_n$ satisfy their respective invariants $R_1, \ldots, R_n$ and the local state satisfies P, any execution of the command C in parallel with an environment that preserves the resource invariants proceeds without giving an error or a race condition and, if it terminates, ends in a final state where the shared resources satisfy their respective invariants and the local state of the process satisfies Q [7].

A.1 Programming language notation

The syntax of commands used in our programming notation is as follows:

$$\begin{array}{rl} C ::= & x := E \mid x := [E] \mid [E] := E' \mid \langle C \rangle \\ & \mid \textbf{skip} \mid C_1; \ldots; C_n \mid C_1 \parallel C_2 \\ & \mid \textbf{if } E_1 \Rightarrow C_1 \,[]\, \cdots \,[]\, E_n \Rightarrow C_n \textbf{ fi} \\ & \mid \textbf{do } E_1 \Rightarrow C_1 \,[]\, \cdots \,[]\, E_n \Rightarrow C_n \textbf{od} \\ & \mid \textbf{resource } r(X) \textbf{ in } C \end{array}$$

The command $x := E$ means the variable x should be modified (in the store) to have the value of E, whereas $x := [E]$ means that it should be modified to have the contents of the heap location with address E. (Note that $[E]$ is not regarded as an "expression", despite its appearance. This is because expressions in Separation Logic are required to be heap-independent.) The command $[E] := E'$ means the heap location with address E should be modified to have the value of E'. Note that each basic command involves at most one read/write operation on the heap memory.

The **if** and **do** commands represent the *guarded command* notations popularized by Dijkstra. The **if** command represents a nondeterministic choice between alternatives depending on the conditions E_i (more than one of which might hold). If E_i holds, execution can continue with the command C_i. If none of the conditions holds, the command leads to an error. The **do** command is a repeated iteration of the alternatives C_i, governed by the enabling conditions E_i. If none of the E_i holds, the command terminates.

The **resource** command declares a resource r with protected variables X for use within a command that presumably uses parallel composition. All the variables listed in X must be used only inside atomic brackets in C.

We also use, by "syntactic sugar," a more elaborate version of the conditional command with atomic branching:

$$\begin{array}{l} \textbf{atomic } \langle C_0 | \textbf{ if } E_1 \Rightarrow |B_1\rangle; C_1 \,[]\, \cdots \,[]\, E_n \Rightarrow |B_n\rangle; C_n \textbf{ fi} \\ \quad \stackrel{def}{=} \textbf{ begin var } b : \text{integer}; \\ \qquad \langle C_0; \textbf{ if } E_1 \Rightarrow B_1;\ b := 1 \,[]\, \cdots \,[]\, E_n \Rightarrow B_n;\ b := n \textbf{ fi}\rangle; \\ \qquad \textbf{if } b = 1 \Rightarrow C_1 \,[]\, \cdots \,[]\, b = n \Rightarrow C_n \textbf{ fi} \\ \quad \textbf{end} \end{array}$$

where b is a fresh variable. Its semantics is that the initial setup command C_0, the conditional test E_i and the corresponding initial steps of the chosen branch B_i are done atomically. After this, the command C_i is executed, but outside the atomic section.

There is also a corresponding version of the **do** command with atomic branching:

$$\begin{array}{l} \textbf{atomic } \langle C_0' | \textbf{ do } E_1 \Rightarrow |B_1\rangle; C_1; \langle D_1 | \,[]\, \cdots \,[]\, E_n \Rightarrow |B_n\rangle; C_n; \langle D_n | \textbf{ od} \\ \quad \stackrel{def}{=} \textbf{ begin var } b : \text{integer}; \\ \qquad \langle C_0;\ test\rangle; \\ \qquad \textbf{do } b = 1 \Rightarrow C_1; \langle D_1; test\rangle \,[]\, \cdots \,[]\, b = n \Rightarrow C_n; \langle D_n; test\rangle \textbf{ od} \\ \quad \textbf{end} \\ \text{where} \\ \quad test \stackrel{def}{=} \textbf{if } E_1 \Rightarrow B_1;\ b := 1 \,[]\, \cdots \,[]\, E_n \Rightarrow B_n;\ b := n \,[]\, \textstyle\bigwedge_{i=1}^{n} \neg E_i \Rightarrow b := 0 \textbf{ fi} \end{array}$$

The first iteration of the loop is similar to atomic-if. The setup command C_0 is executed atomically with the chosen test E_i and the corresponding B_i. For subsequent iterations, the trailing command D_i of the chosen branch in the current iteration and the chosen test E_j and the corresponding B_j of the next iteration are done atomically. If the C_i and D_i are omitted, they are taken to be **skip**.

A.2 Proof rules

Our programming language is a revised version of O'Hearn's Concurrent Separation Logic [21] adapted to the use of fine-grained concurrency and extended to allow nested atomic sections

and multiple resources. We borrow the notation of Brookes [7] to formalize some of these aspects.[6]

A judgement in the logic is of the form $\Gamma \vdash \{P\}C\{Q\}$ with Γ being a *resource context* of the form $r_1(X_1) : R_1, \ldots, r_n(X_n) : R_n$, where r_i are resource names, X_i are lists of "protected" variables of the resources and R_i are their respective resource invariants, satisfying the condition $X_i \cap X_j = \varnothing$ for $i \neq j$. We use $\circledast\Gamma$ to denote $R_1 \star \cdots \star R_n$. The free variables of Γ are $FV(\Gamma) = X_1 \cup \ldots \cup X_n$. (The rules ensure that the free variables of the resource invariants are included among these.)

If there is a single central resource with a resource invariant R, as it is in the majority of this paper, we abbreviate the judgement to $R \vdash \{P\}\, C\, \{Q\}$.

If P is a formula, the notation $FV(P)$ denotes the set of free variables of P. If C is a command, $FV(C)$ denotes the set of variables that occur outside atomic brackets in C. We also use the FV notation with multiple arguments, e.g., $FV(\Gamma, P, C, Q)$, in which case we mean the union of the individual sets of free variables.

The structural rules of the logic are the following:

$$\text{CONSEQ}\quad \frac{P' \Rightarrow P \quad \Gamma \vdash \{P\}\, C\, \{Q\} \quad Q \Rightarrow Q'}{\Gamma \vdash \{P'\}\, C\, \{Q'\}}$$

$$\text{EXIST}\quad \frac{\Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{\exists x\colon P\}\, C\, \{\exists x\colon Q\}} \qquad \text{if } x \text{ is not in } FV(\Gamma, C)$$

$$\text{SUBST}\quad \frac{\Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P[E/x]\}\, C\, \{Q[E/x]\}} \qquad \text{if } x \text{ is not in } FV(C)$$

$$\text{INV}\quad \frac{\Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P \wedge R\}\, C\, \{Q \wedge R\}} \qquad \text{if } R \text{ is pure and } C \text{ does not modify any variable in } FV(R)$$

$$\text{FRAME}\quad \frac{\Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P \star R\}\, C\, \{Q \star R\}} \qquad \text{if } C \text{ does not modify any variable in } FV(R)$$

$$\text{AUXILIARY}\quad \frac{\Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P\}\, C\backslash X\, \{Q\}} \qquad \text{if } X \text{ is auxiliary for } C \text{ and } X \cap FV(P, Q) = \varnothing$$

A set of variables X is said to be *auxiliary* for C if every free occurrence of a variable from X in C is in an assignment that only affects the variables in X. The command $C\backslash X$ is obtained by deleting all assignments to variables in X.

The proof rules for the sequential part of the programming language are straightforward adaptations of the sequential Separation Logic [30]:

$$\frac{}{\Gamma \vdash \{x = m \wedge \mathbf{emp}\}\, x := E\, \{x = E[m/x] \wedge \mathbf{emp}\}}$$

$$\frac{}{\Gamma \vdash \{x = m \wedge E \overset{p}{\longmapsto} E'\}\, x := [E]\, \{x = E' \wedge E[m/x] \overset{p}{\longmapsto} E'\}} \qquad \text{if } x \text{ not in } FV(E')$$

$$\frac{}{\Gamma \vdash \{E \overset{1}{\longmapsto} \_\}\, [E] := E'\, \{E \overset{1}{\longmapsto} E'\}}$$

$$\frac{}{\Gamma \vdash \{P\}\, \mathbf{skip}\, \{P\}} \qquad \frac{\Gamma \vdash \{P\}\, C_1\, \{Q\} \quad \Gamma \vdash \{Q\}\, C_2\, \{R\}}{\Gamma \vdash \{P\}\, C_1; C_2\, \{R\}}$$

$$\frac{P \Rightarrow E_1 \vee \cdots \vee E_n \quad \Gamma \vdash \{P \wedge E_i\}\, C_i\, \{Q\} \quad (i = 1, \ldots, n)}{\Gamma \vdash \{P\}\, \mathbf{if}\ E_1 \Rightarrow C_1 \,[]\, \cdots \,[]\, E_n \Rightarrow C_n\ \mathbf{fi}\, \{Q\}}$$

$$\frac{\Gamma \vdash \{P \wedge E_i\}\, C_i\, \{P\} \quad (i = 1, \ldots, n)}{\Gamma \vdash \{P\}\, \mathbf{do}\ E_1 \Rightarrow C_1 \,[]\, \cdots \,[]\, E_n \Rightarrow C_n\ \mathbf{od}\, \{P \wedge (\neg E_1 \wedge \cdots \wedge \neg E_n)\}}$$

[6] Brookes's framework has subtle differences from that of O'Hearn. Our logic follows the O'Hearn system despite the use of Brookes's notation.

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.

The proof rules dealing with concurrent constructs are as follows:

$$\textit{RESOURCE}\ \frac{\Gamma,\, r(X) : R \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{R \star P\}\ \textbf{resource}\ r(X)\ \textbf{in}\ C\, \{R \star Q\}}\quad \text{if } R \text{ is precise and } FV(R) \subseteq FV(\Gamma) \cup X$$

$$\textit{PAR}\ \frac{\Gamma \vdash \{P_1\}\, C_1\, \{Q_1\} \quad \Gamma \vdash \{P_2\}\, C_2\, \{Q_2\}}{\Gamma \vdash \{P_1 \star P_2\}\, C_1 \parallel C_2\, \{Q_1 \star Q_2\}}\quad \text{if } C_i \text{ does not modify variables in } FV(P_j, C_j, Q_j) \text{ (for } i \neq j)$$

$$\textit{ATOMIC}\ \frac{\vdash \{P \star (\circledast\Gamma)\}\, C\, \{Q \star (\circledast\Gamma)\}}{\Gamma \vdash \{P\}\, \langle C \rangle\, \{Q\}}\quad \text{if no other process modifies variables in } FV(P, Q)$$

The *RESOURCE* rule requires a resource invariant R to be extricated from both the pre-condition and post-condition of the **resource** command which can be used as part of the resource context for the body C. The rule *ATOMIC* allows all the resource invariants in the context to be retrieved in the pre-condition and post-condition of an atomic block. The side condition for the rule, which is informally stated, requires that the local pre-condition and post-condition should not have free variables that are modified by "other processes". It is possible to formalize the informal statement by using more sophisticated judgements that track variables used in the assertions and commands, but the simple rule should suffice for our purposes. It is worth noting that *all* the resources in the resource context are absorbed in an atomic block. This is in contrast the conditional critical regions treated in [7,21] which absorb a single resource named in the critical region. Likewise, all the protected variables of Γ can occur in the atomic block, but they cannot occur in commands outside atomic blocks.

The following proof rules can be derived for the atomic-branching commands from their sequential definition. Their side conditions are similar to those of the *ATOMIC* rule above. Note that the **atomic** iteration allows a loop invariant L.

$$\frac{\vdash \{P \star (\circledast\Gamma)\}\, C_0\, \{P' \wedge (E_1 \vee \cdots \vee E_n)\} \qquad \vdash \{P' \wedge E_i\}\, B_i\, \{Q_i \star (\circledast\Gamma)\} \quad \Gamma \vdash \{Q_i\}\, C_i\, \{R\} \quad (i = 1, \ldots, n)}{\Gamma \vdash \{P\}\ \textbf{atomic}\ \langle C_0 |\ \textbf{if}\ E_1 \Rightarrow |B_1\rangle; C_1 \,[]\, \cdots \,[]\, E_n \Rightarrow |B_n\rangle; C_n\ \textbf{fi}\ \{R\}}$$

$$\frac{\vdash \{P \star (\circledast\Gamma)\}\, C_0\, \{L\} \quad L \wedge \neg E_1 \wedge \cdots \wedge \neg E_n \Rightarrow R \star (\circledast\Gamma) \qquad \vdash \{L \wedge E_i\}\, B_i\, \{Q_i \star (\circledast\Gamma)\} \quad \Gamma \vdash \{Q_i\}\, C_i\, \{R_i\} \quad \vdash \{R_i \star (\circledast\Gamma)\}\, D_i\, \{L\} \quad (i = 1, \ldots, n)}{\Gamma \vdash \{P\}\ \textbf{atomic}\ \langle C_0 |\ \textbf{do}\ E_1 \Rightarrow |B_1\rangle; C_1; \langle D_1 |\ []\ \cdots\ []\ E_n \Rightarrow |B_n\rangle; C_n; \langle D_n |\ \textbf{od}\ \{R\}}$$

B The invariants used in proving the DLMSS algorithm

The algorithm presented in [11] is a rather challenging concurrent program to prove correct. We summarize the critical ideas used in the correctness proofs, right from the 1970s [11,13].

The black-to-white invariant (corresponding to P1, P3 and P3a in [11]) 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 [13] 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 [11]) 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.

C Proving the operations

We gather all the resource and local invariants into Table 6 for easy reference.

$$
\begin{array}{l}
RI \stackrel{def}{=} \exists U, V, W, X\colon RP(U,V,W) \wedge X = \{NIL\} \cup U \cup V \wedge [0..N] = X \cup W \wedge \\
\qquad\qquad whiteI(X) \wedge grayI(X) \wedge bwI(X) \wedge blackI \\
RP(U,V,W) \stackrel{def}{=} \\
\quad cell^F(NIL) \star freeList^{\rho F \rho}(V) \star reachGraph^{\rho}(U,\, V \cup \{NIL\}) \star cells^F(W) \\
whiteI(X) \stackrel{def}{=} \forall i \in X : i.colour \stackrel{\rho}{\hookrightarrow} w \Rightarrow \\
\qquad\qquad (in\_marking \Rightarrow \exists j : j \in X \wedge propath(j,i)) \wedge \\
\qquad\qquad (\neg in\_marking \Rightarrow \neg scanned[i]) \\
grayI(X) \stackrel{def}{=} in\_marking \Rightarrow (\exists i : i \in X \wedge i.colour \stackrel{\rho}{\hookrightarrow} g) \Rightarrow \\
\qquad\qquad\qquad\qquad (\exists j : j \in [0..N] \wedge j.colour \stackrel{\rho}{\hookrightarrow} g \wedge \neg scanned[j]) \\
bwI(X) \stackrel{def}{=} in\_marking \Rightarrow \\
\qquad\qquad \forall k, j \in X : (bwedge(k,j) \vee Cwedge(k,j)) \Rightarrow (k = ladd \wedge k.left \stackrel{\rho}{\hookrightarrow} j) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (k = radd \wedge k.right \stackrel{\rho}{\hookrightarrow} j) \\
blackI \stackrel{def}{=} (\forall i \in [0..N] : i.colour \stackrel{\rho}{\hookrightarrow} b \Rightarrow tested[i]) \wedge \\
\qquad\qquad (\forall i \in [0..N] : tested[i] \Rightarrow i.colour \stackrel{\rho}{\hookrightarrow} g \vee i.colour \stackrel{\rho}{\hookrightarrow} b) \\
mutI \stackrel{def}{=} \exists U, V_0 : mutP(U, V_0) \wedge ladd = NIL = radd \wedge \neg avail \\
mutP(U,V) \stackrel{def}{=} reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(\_, \_, V) \\
colP \stackrel{def}{=} \exists e.\, ENDFREE \stackrel{\overline{\rho}}{\longmapsto} (e, NIL, \_) \star listseg^{\overline{\rho}}(e, NIL, \_) \star tested\_colours \\
tested\_colours \stackrel{def}{=} \circledast_{k \in [0..N]} (\neg tested[k] \wedge \mathbf{emp}) \vee \\
\qquad\qquad\qquad\qquad (\exists c : tested[k] \wedge k.colour \stackrel{\overline{\rho}}{\longmapsto} c \wedge c \in \{g, b\}) \\
colI \stackrel{def}{=} colP \wedge lgray = (NIL, NIL) = rgray \wedge reclaim = NIL \\
markI(i) \stackrel{def}{=} colI \wedge in\_marking \wedge i \in [0..N+1] \wedge \forall k \in [0..N] : (scanned[k] \iff k < i) \\
sweepI(i) \stackrel{def}{=} colI \wedge \neg in\_marking \wedge i \in [0..N+1] \wedge \\
\qquad\qquad \forall k \in [0..N] : (k < i \Rightarrow \neg scanned[k] \wedge \neg tested[k]) \wedge (k \geqslant i \Rightarrow scanned[k])
\end{array}
$$

Table 6 Resource and local invariants

C.1 Mutator getting a node from the free list

get new left edge(k):
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(\_, \_, \varnothing) \wedge k \in U \wedge \neg avail\}$
$\langle$f := [FREE.left]$\rangle$;
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, \_, \varnothing) \wedge k \in U \wedge \neg avail\}$
$\langle$e := [ENDFREE.left]; $avail$:= (f $\neq$ e)$\rangle$;
$\{\exists U, V: reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, \_, V) \wedge k \in U \wedge avail = (f \neq e)\}$
do f = e $\Rightarrow$ $\langle$e := [ENDFREE.left]; $avail$:= (f $\neq$ e)$\rangle$ **od**;
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, \_, \{f\}) \wedge k \in U \wedge avail\}$
$\langle$m := [f.left]$\rangle$;
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(f, m, \{f\}) \wedge k \in U \wedge avail\}$
addleft(k, f);
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{f, NIL\}) \star freeHead^{\overline{\rho}}(f, m, \{f\}) \wedge k \in U \wedge avail\}$
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{f, NIL\}) \star FREE \overset{\overline{\rho}}{\longmapsto} (f, NIL, \_) \star f \overset{\overline{\rho}}{\longmapsto} (m, NIL, \_) \wedge avail\}$
$\langle$addleft(FREE, m); $avail$:= false$\rangle$;
$\{\exists U: (reachGraph^{\overline{\rho}}(U, \{m, NIL\}) \wedge f \overset{\overline{\rho}}{\hookrightarrow} (m, NIL, \_)) \star FREE \overset{\overline{\rho}}{\longmapsto} (m, NIL, \_) \wedge \neg avail\}$
$\{\exists U: (reachGraph^{\overline{\rho}}(U, \{m, NIL\}) \wedge f \overset{\overline{\rho}}{\hookrightarrow} (m, NIL, \_)) \star freeHead^{\overline{\rho}}(m, \_, \varnothing) \wedge \neg avail\}$
addleft(f, NIL);
$\{\exists U: reachGraph^{\overline{\rho}}(U, \{NIL\}) \star freeHead^{\overline{\rho}}(\_, \_, \varnothing) \wedge \neg avail\}$

Table 7 Mutator get operation

The proof of the **get** operation of the mutator, outlined in Table 7 is also interesting, since a node has to be extricated while carefully avoiding trespassing on the free list except for the first free node, and the node must not get detached from both the structures at any time. (The procedure **addleft** is called with different preconditions/postconditions in different occurrences. For instance, if $n \notin U$ in the precondition then, in the postcondition, we have $reachGraph^{\overline{\rho}}(U, V \cup \{n\})$. These specifications should be easy for the reader to reconstruct if needed.) Note that $freeHead^{\overline{\rho}}(f, g, V)$ means

$$FREE \overset{\overline{\rho}}{\longmapsto} (f, NIL, \_) \star ((\neg avail \wedge f = g \wedge V = \varnothing \wedge \mathbf{emp}) \vee (avail \wedge f \neq g \wedge f \overset{\overline{\rho}}{\longmapsto} (g, NIL, \_) \wedge V = \{f\}))$$

The $freeHead$ could be either empty or a singleton list segment. Which it is depends on whether the free list has any nodes other than the end node. The auxiliary variable $avail$ makes this information available to the mutator.

1. In the first step, the header node $FREE$ is read using the read-complement permission available for the freeHead.
2. After the second step, a busy wait, we are assured that f is distinct from $ENDFREE$, and hence the free list is nonempty. The node f is now a free node.
3. In the third step, the node f is read.
4. In the fourth step, the node f is attached to the graph at the node k. However, we are careful not to count the node f as part of the reachable graph because it is still a part of the free list. The reachable graph and the free list are required to be separate in our invariants. The predicate $reachGraph(U, \{f, NIL\})$ spans all the cells reachable from $ROOT$ except for f and NIL.
5. Next the node f is detached from the free list by advancing the pointer $FREE.left$. It becomes an integral part of the reachable graph (and, hence, the set U). However, since the node f still points into the free list starting at node m, the reachable graph must be blocked from encroaching into the free list at node m.
6. Finally, f's left pointer is reset to NIL and the local invariant is reestablished.

Note that the last three commands have to be ordered carefully. If they are reordered, for instance, as:

addleft(FREE, m); addleft(k, f); addleft(f, NIL);

then the node f is detached from the free list too early. It becomes a garbage node and it is liable to be garbage collected in between the first two commands. Our invariants prohibit this order. Recall that the resource permission $RP(U, V, W)$ specifies all the nodes outside U, V and $\{NIL\}$ to be in W, and the central resource has full permission for the nodes in W. So it is not possible to satisfy the precondition for addleft(k, f) which requires $\overline{\rho}$ permission for f.

The first assertion to be proved is $RI \vdash \{P_0\}\, C_1\, \{P_1\}$ with

$$\begin{aligned} P_0 &\equiv \exists U : (reachGraph^{\overline{\rho}}(U, \{NIL\}) \wedge k \in U) \star freeHead^{\overline{\rho}}(\_, \_, \varnothing) \wedge \neg avail \\ C_1 &\equiv f := [FREE.left]; \\ P_1 &\equiv \exists U : (reachGraph^{\overline{\rho}}(U, \{NIL\}) \wedge k \in U) \star freeHead^{\overline{\rho}}(f, \_, \varnothing) \wedge \neg avail \end{aligned}$$

This only requires read permission on the node pointed to by *FREE*. The next assertion $RI \vdash \{P_1\}\, \langle C_2 \rangle\, \{P_2\}$ with

$$\begin{aligned} C_2 &\equiv (e := [ENDFREE.left]; avail := (f \neq e)) \\ P_2 &\equiv \exists U, V : (reachGraph^{\overline{\rho}}(U, \{NIL\}) \wedge k \in U) \star freeHead^{\overline{\rho}}(f, \_, V) \wedge avail = (f \neq e) \end{aligned}$$

establishes the invariant for the loop that follows. Note that *ENDFREE* can be read by borrowing such a permission from the resource invariant but since the mutator has no permission on it, it cannot be mentioned in the assertion. Thus the auxiliary *avail*, using the inequality of the local copies f and e, transfers information from the central resource about whether there is a node available for the mutator to obtain.

The next assertion $RI \vdash \{P_2\}\, C_3\, \{P_3\}$ with

$$\begin{aligned} C_3 &\equiv \textbf{do}\ f = e \Rightarrow \langle e := [ENDFREE.left]; avail := (f \neq e) \rangle \textbf{od}; \\ P_3 &\equiv \exists U : (reachGraph^{\overline{\rho}}(U, \{NIL\}) \wedge k \in U) \star freeHead^{\overline{\rho}}(f, \_, \{f\}) \wedge avail \end{aligned}$$

is more interesting. The exit condition of the busy wait, $f \neq e$, relates through the auxiliary *avail* to the existence of a left successor for f and enables $freeHead$ to extend to it. Using the next $RI \vdash \{P_3\}\, C_4\, \{P_4\}$, mutator gets a local copy in m.

$$\begin{aligned} C_4 &\equiv m := [f.left]; \\ P_4 &\equiv \exists U : (reachGraph^{\overline{\rho}}(U, \{NIL\}) \wedge k \in U) \star freeHead^{\overline{\rho}}(f, m, \{f\}) \wedge avail \end{aligned}$$

Now we have the sequence of three **addleft** commands mentioned above, where mutator attaches the free node to k, and advances the head of the free list to m (*avail* being reset to its old status), and the local variable f is cleaned up.

C.2 Operations during the collector's sweeping phase

We illustrate the proof of the *raison d'être* of this program, the **collect** action in Table 8. From the white invariant, we see that a white scanned node must be in the unreachable garbage. The invariant has full permission on its links, and, since $tested[i]$ is false by the black invariant, the colour field as well. By setting *reclaim* to i, the collector process extracts the full permission to the node i from the resource invariant. This is represented in the pre-condition of the **collect** action.

The action starts by setting the *scanned* flag of i to false, a necessary step to preserve the white invariant before the node i is made accessible. Next, the link fields of i are set to *NIL* using the 1 permission.

Since the collector has $\overline{\rho}$ permission for *ENDFREE* as well as the sentinel node e of the free list, it is able to link in the node i as the successor to e. This is done in the command $\langle$[e.left] := i; *reclaim* := *NIL*$\rangle$. Once the node i is linked in, it becomes reachable from *ENDFREE* and, so, ρ permission should be given to the resource invariant. Thus it is necessary to set *reclaim* to *NIL*. The collector now retains only $\overline{\rho}$ permission for node i.

When the pointer *ENDFREE* is moved to i, the former sentinel node e becomes part of the usable portion of the free list, and its $\overline{\rho}$ permission is transferred to the resource invariant. The node i becomes the new sentinel node. None of the resource invariants are affected by these actions.

Collect white node(i):

$\{sweepI(i) \star i \overset{1}{\longmapsto} (\_, \_, w) \wedge i \leqslant N \wedge reclaim = i)\}$
$\langle scanned[\mathrm{i}] := \mathrm{false}\rangle;$
$\{sweepI(i+1) \star i \overset{1}{\longmapsto} (\_, \_, w) \wedge i \leqslant N \wedge reclaim = i\}$
$\langle[\mathrm{i.left}] := \mathrm{NIL}\rangle; \langle[\mathrm{i.right}] := \mathrm{NIL}\rangle;$
$\{sweepI(i+1) \star i \overset{1}{\longmapsto} (NIL, NIL, w) \wedge i \leqslant N \wedge reclaim = i\}$
$\langle \mathrm{e} := [\mathrm{ENDFREE.left}]\rangle;$
$\{sweepI(i+1) \star i \overset{1}{\longmapsto} (NIL, NIL, w) \wedge i \leqslant N \wedge reclaim = i \wedge ENDFREE \overset{\overline{\rho}}{\longrightarrow} (e, NIL, \_)\}$
$\langle[\mathrm{e.left}] := \mathrm{i};\ reclaim := \mathrm{NIL}\rangle;$
$\{sweepI(i+1) \wedge i \leqslant N \wedge reclaim = NIL$
$\quad \wedge ENDFREE \overset{\overline{\rho}}{\longrightarrow} (e, NIL, \_) \wedge e \overset{\overline{\rho}}{\longrightarrow} (i, NIL, \_) \wedge i \overset{\overline{\rho}}{\longrightarrow} (NIL, NIL, \_)\}$
$\langle[\mathrm{ENDFREE.left}] := \mathrm{i}\rangle;$
$\{sweepI(i+1) \wedge i \leqslant N \wedge reclaim = NIL \wedge ENDFREE \overset{\overline{\rho}}{\longrightarrow} (i, NIL, \_) \wedge i \overset{\overline{\rho}}{\longrightarrow} (NIL, NIL, \_)\}$
$\{sweepI(i+1) \wedge reclaim = NIL\}$
i := i+1
$\{sweepI(i) \wedge reclaim = NIL\}$

Whiten black node(i):

$\{sweepI(i) \wedge reclaim = NIL \wedge i \leqslant N \wedge i.colour \overset{\overline{\rho}}{\longrightarrow} b\}$
$\langle[\mathrm{i.colour}] := \mathrm{white};\ scanned[\mathrm{i}] := \mathrm{false};\ tested[\mathrm{i}] := \mathrm{false}\rangle;$
$\{sweepI(i+1) \wedge reclaim = NIL \wedge i \leqslant N\}$
i := i+1
$\{sweepI(i) \wedge reclaim = NIL\}$

Table 8 Collect and other actions

The operations **skip gray node** and **whiten black node** are straightforward. In both cases, we know that the node i must have been *tested* (as per the black invariant) and, so, the collector has $\overline{\rho}$ permission to its colour field. By combining with the resource invariant's ρ permission, the collector is able to change the colour field of a black node to white. The *scanned* flag is set to false simultaneously in order to preserve the white invariant.