

# Fine-grained concurrency with separation logic

Kalpesh Kapoor<sup>1</sup>, Kamal Lodaya<sup>2</sup>, Uday Reddy<sup>3</sup>

<sup>1</sup> Dhirubhai Ambani Institute of Information and Communication Technology,  
Near Indroda Circle, Gandhinagar 382 007, India

<sup>2</sup> The Institute of Mathematical Sciences,  
C.I.T. Campus, Chennai 600 113, India

<sup>3</sup> School of Computer Sciences, University of Birmingham,  
Edgbaston, Birmingham B15 2TT, United Kingdom

**Abstract.** Separation Logic is a recent development in programming logic which has been applied by Peter O’Hearn to concurrency based on critical sections as well as semaphores. In this paper, we go one step further and apply it to fine-grained concurrency. We note that O’Hearn’s formulation of Concurrent Separation Logic is by and large applicable to fine-grained concurrent programs with only minor adaptations. However, proving substantial properties of such programs involves the employment of sophisticated “permissions” frameworks so that different processes can have different levels of access and exchange such access. We illustrate these techniques by showing the correctness proof of a concurrent garbage collector originally studied by Dijkstra et al.

**Key words:** Concurrency, Garbage collection, Heap storage, Separation Logic, Shared variables

## 1 Introduction

Separation logic [13] is a logic for reasoning about programs with pointers. O’Hearn [10] extended the logic to reason about concurrent shared-variable race-free programs, and Brookes [4] provided a semantics for this logic.

In this paper, we undertake the exercise of extending separation logic to fine-grained concurrent programs which do have races, but where the races do not affect the correctness of the program. Our technique is simple. We adapt the concurrent separation logic rules for resources and conditional critical regions [10] and apply them to shared variables in general. In other words, shared variables become “logical” resources. Associated with shared storage is a *resource invariant*, and every update of a shared variable maintains the invariant.

We try out our ideas on a well-known example: the concurrent garbage collector of Dijkstra, Lamport et al [6], which has been used as a challenge for correctness proofs [7, 1], theorem provers (Boyer-Moore [14], PVS [8], Isabelle/HOL [9], B and Coq [5]) and algorithm design [15]. Correctness proofs of garbage collectors have been provided in separation logic earlier [2] (cf. also the correctness of a graph marking algorithm by Yang [16]). To our knowledge, this is the first proof of a concurrent garbage collector using separation logic.

## 1.1 Concurrent separation logic

The key idea in separation logic for parallel programs with shared variables [10] is to consider them as disjoint processes with a set of shared resources (which have disjoint lists of variables belonging to them), accessed through conditional critical regions or semaphores which maintain mutual exclusion. The use of the separating conjunction  $\star$  in pre- and postconditions allows assertion and transfer of “ownership” among processes. The paper [10] gives convincing examples of the use of this paradigm.

However a variable free in one process cannot be changed in another, unless it belongs to a shared resource. Hence concurrent separation logic does not allow correctness proof of *racy* programs, those where two parallel processes may access the same portion of state at the same time.

But racy programs abound, typically in memory managers, OS kernels, etc. One classic example is the concurrent garbage collector of Dijkstra, Lamport and others [6], which collects unused memory in racy concurrency from a “user” process which is employing the memory (known as the *mutator* in this literature).

In Gries’s correctness proof [7] of the concurrent garbage collector, we observe that the racy concurrency works because the collector and mutator processes cooperate in that the atomic action performed on the shared variables by both maintains a common invariant. This suggests extending separation logic to parallel programs where the shared storage is viewed as a (monolithic) shared resource with an associated invariant which is maintained by all processes updating shared variables. Shared variable updates need not be forbidden in such programs.

Semantically, this means that if  $\{P\}C\{Q\}$  can be proven then any execution of  $C$  starting from a state satisfying  $P$  will not attempt to dereference a dangling pointer and will result, if  $C$  terminates, in a state satisfying  $Q$ . Since nothing is said about races, this means that if there is a race condition in  $C$  which can lead to deleterious effects, that would just mean that  $\{P\}C\{Q\}$  cannot be proven.

## 2 Garbage collection

Most general purpose programming languages provide some mechanism to create objects dynamically i.e. at run-time. This is facilitated by making use of a free store often referred to as a **heap**. Each object created at run-time has a lifetime associated with it during which it can be used. The lifetime may be decided by the scope where the object is declared or created. Once the lifetime of an object is over the space allocated can be reclaimed and reused.

The process of reclaiming unusable space is called **garbage collection**. This could be manual, in which case, it is the responsibility of a programmer to write an explicit command to dispose of an unused object. Or we can have automatic garbage collection, for example in a language like Lisp or Java, where the execution environment identifies and reuses space used by an object whose lifetime is over.

The paper [6] proposed a concurrent algorithm for automatic garbage collection, where the garbage collector runs concurrently with the user program (the “mutator”). The collector repeatedly checks for memory which was previously given to the mutator but is no longer accessible to it, and puts it for re-use into a *free list*.

## 2.1 The DLMSS algorithm

The DLMSS garbage collection algorithm [6] has two components, a *mutator* and a *collector*. The mutator represents a user process that can request for a memory cell at run time.

The memory potentially available to the mutator is represented as an array of fixed size. Each node (i.e. an element of the array) has, in addition to whatever data is stored in it (which we completely ignore), three fields for storing a left and a right pointer and a *colour*. The data used by the mutator forms a *binary graph* within the array using the left and right pointers, with a *root* node. Every node of the data graph is reachable from the root by a path of nodes, following either a left or a right pointer to go from one node to the next.

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

When the mutator needs more memory, it takes a node from the free list and puts it on the data graph. We call this the mutator’s **get** action. In addition, the mutator can **modify** a left or right pointer to point to some other node in the data graph, or even 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 collector’s job is to find such nodes and add them to the free list, so that they can be reused when required. 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 identifies nodes which are reachable from either the data root or the start of the free list, and then a *sweeping* phase which puts nodes which are not marked onto the free list. The colour field of each node represents the mark: **black** means a node is marked and **white** means it is unmarked.

The basic idea behind the marking phase is that it begins by marking the data root and the free list start, and then keeps running through the array marking the successors of marked nodes. When no marked node has a successor, the unmarked nodes are garbage.

The sweeping phase runs through the array, adding nodes left white 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 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 [10].

This mark-sweep cycle continues forever. “Stop the world” collectors work by once in a while freezing the mutator’s actions, doing their cleanup, and then allowing it to continue.

The DLMSS collector, in contrast, works all the time, *concurrently* interleaving its work with the mutator’s actions. To facilitate this, DLMSS introduced a **gray** colour intermediate between “marked” and “unmarked”. The data root and the free list start are first coloured gray. The marking phase makes repeated runs through the array; when it finds a gray node, its successors are coloured gray (if they were unmarked), the node is marked by colouring it black, and a new run is started if one of the successors was already processed during this run.

Hence, progressing from the root and start nodes respectively, the data graph and the free list are coloured black at the beginning, then they have a gray frontier where marking is in progress, and then they are unmarked (white).

## 2.2 Proving the DLMSS algorithm

The algorithm presented in [6] is a rather challenging concurrent program to prove correct. The authors 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. Around the same time, Gries [7] outlined a proof using a formal inference system, the well-known proof system for shared variable programs due to Owicki and Gries [11]. Since then many researchers have given alternative proofs and algorithms. For example Ben-Ari, in [1], gave an algorithm that uses two colours and has less complexity. Flaws in his correctness proof were found when checking the proof mechanically [14].

Let us summarize briefly the critical ideas used in the correctness proofs, right from the 1970s [6, 7].

A **white invariant** is used in the proof of the marking phase: every white node is reachable from a gray node by a path consisting only of white nodes. Although the marking phase only updates colours, this invariant relies on the structure of the data graph and free list.

Unfortunately this pretty picture is 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. When the mutator gets or modifies a node, the mutator obliges the collector by colouring this new node gray (if it was unmarked), which restores the white invariant.

*But this is a race condition.* Both the mutator and the collector are now updating colours.

A **gray invariant** is also used in the proof of the marking phase: if there is a gray node in 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 in 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.

A **black-to-white invariant** should say that there are no black-to-white edges in the graph (because they are mediated by gray nodes). It turns out that this can be violated during the middle of a mutator modification. Hence the invariant specifies 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 a modification.

Gries's proof [7] makes do with this because of the way Owicki-Gries interference freedom works, but the DLMSS proof [6] needs a further variation. They define a *C*-edge to be a gray-to-white edge which occurs in the marking phase when a gray node is greying its children, because of a mutator modification. The black-to-white invariant above is modified to include this possibility as well, and the white invariant is strengthened to avoid *C*-edges in the gray-to-white path. Since our proof does not use interference-freedom, we will use the invariants from the DLMSS proof.

### 3 Inference rules

Our purpose in verifying the DLMSS algorithm is to attack a fine-grained concurrent program which is a challenge for the proof system. In this section we proceed to develop a proof outline using the inference rules of concurrent separation logic, which we present as we go along.

The boolean arrays *marked* and *swept* and the pointer variables *mod*, *leftgray* and *rightgray* are auxiliary variables for the purpose of the proof. The two processes mutator and collector have invariants *mutI* and *colI* associated with them, which are detailed in the proofs of these processes.

```
gc  $\stackrel{def}{=}
\text{const ROOT, FREE, NIL: [0..N];}
\text{var ENDFREE: [0..N];}
\text{var i: unsigned;}
\text{auxvar marked[0..N]: bool; swept[0..N]: bool updated by collector;}
\text{in\_marking: bool updated by collector;}
\text{leftgray, rightgray: [0..N] updated by collector;}
\text{mod: [0..N] updated by mutator;}
\{cells^1[0..N]\}
\text{for i := 0 to N do whiten(i); swept[i] := true; marked[i] := false od;}
\text{in\_marking := false;}
\text{mod, leftgray, rightgray := NIL, NIL, NIL;}
RI \vdash \{mutI \star (colI \wedge \neg in\_marking \wedge \forall i \in [0..N] : swept[i] \wedge \neg marked[i])\}
\text{mutator || collector}
\{false \star false\}$ 
```

This can be proved using the declaration rule below, modelled on the resource declaration rule of concurrent separation logic [10]. Associated with the shared

variables is the **resource invariant**  $RI$ , a predicate over the (shared) states of the processes. We define the resource invariant for our proof in the next section.

$$\frac{\{P\}init\{P' \star RI\}, \quad RI \vdash \{P'\}C_1 || \dots || C_n\{Q\}}{\{P\}Prog\{Q\}}$$

The judgement form  $RI \vdash \{P\}C\{Q\}$  means that  $\{P\}C\{Q\}$  holds in the context of a resource satisfying the resource invariant  $RI$ .

The parallel composition can be proved by the inference rule for parallel composition, also similar to [10], except that we explicitly mention working under the resource invariant.

$$\frac{RI \vdash \{P_1\}C_1\{Q_1\}, \quad \dots, \quad RI \vdash \{P_n\}C_n\{Q_n\}}{RI \vdash \{P_1 \star \dots \star P_n\}C_1 || \dots || C_n\{Q_1 \star \dots \star Q_n\}},$$

where no local variable free in  $P_i$  or  $Q_i$  is changed in  $C_j$ , for  $i \neq j$  in  $\{1, \dots, n\}$ .

By using the frame rule, the associativity and commutativity of the parallel operator  $||$  and the rule of conjunction over the global state, we can combine the proofs for each process into the form necessary for the declaration rule. Hence we are left to prove:

$$\begin{aligned} &RI \vdash \{mutI\}mutator\{false\} \\ &RI \vdash \{colI \wedge \neg in\_marking \wedge \forall i \in [0..N] : swept[i] \wedge \neg marked[i]\}collector\{false\} \end{aligned}$$

These proof outlines will occupy us for the rest of this paper. But to use shared variables inside a process, we will need further proof rules. These are specified next.

### 3.1 Rules for using shared variables

The basic idea is to treat every basic command implicitly as a critical region which assumes the resource invariant in the beginning and re-establishes it in the end. This is described by the rule:

$$\frac{\{P \star RI\}C\{Q \star RI\}}{RI \vdash \{P\}C\{Q\}}$$

where  $C$  is a basic command and the free variables of  $P$  or  $Q$  are not modified in other processes. The intuitive sense of the rule is that a basic command “grabs” all the cells and their permissions held by the resource invariant, does its action on the combined state of the process and the resource invariant, and then releases the resource. Several processes can be doing this kind of action concurrently, and they interleave each other.

Here is the syntax of the traditional commands we use, where the first four forms are *basic commands*:

$$\begin{aligned} C ::= &x := E \mid x := [E] \mid [E] := E' \mid \langle C' \rangle \\ &\mid \mathbf{if} \ E_1 \Rightarrow C_1 \square \dots \square E_n \Rightarrow C_n \ \mathbf{fi} \\ &\mid \mathbf{do} \ E_1 \Rightarrow C_1 \square \dots \square E_n \Rightarrow C_n \ \mathbf{od} \end{aligned}$$

In addition to these, we have the need for a form of conditional command with atomic conditional branching. The form of the command is a bit involved:

$$\mathbf{if} \langle C_0 \mid E_1 \Rightarrow |C_1\rangle; C'_1 \square \cdots \square E_n \Rightarrow |C_n\rangle; C'_n \mathbf{fi}$$

The semantics of the command is that the initial setup command  $C_0$ , the conditional test  $E_i$  and the corresponding initial steps of the chosen branch  $C_i$  are done atomically. The proof rule for the command is:

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

Contrast this with the rule for ordinary conditional branching:

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

In order for more than one process to write on a shared variable, they all must have write permission. The interleaving of accesses in the semantics of the program at the level of memory operations guarantees that only one process can write at a time, but since this interleaving is fine-grained, assertions about shared variables can be violated “under a process’s feet.” Hence the process is not allowed to make any local assertions about shared variables, all such assertions are in the resource invariant, and all processes will honour the commitment to re-establish it at every update.

The proof rules for other commands are modified to add the presence of the resource invariant.

A weakness of our proof rules is that the entire shared storage is treated as one resource. But shared storage has its own locality, one shared variable might be updated by a subset of processes. In our example program, such a situation does not arise since we have just two processes. Extending the ideas in this paper to a “local” treatment of individual shared variables is an interesting problem.

## 4 Storage, permissions and colours

The model of separation logic we need for our concurrent garbage collector is inspired by that of counting permissions [3]. Full permission (for reading as well as writing on a heap location) is denoted 1, read access permission is denoted  $R$ , and the complement of a read permission is denoted  $-R$ . A read permission and its complement can be combined to obtain a full permission  $R * (-R) = 1$ . Both  $R$  and  $-R$  permissions allow reading, but only 1 allows writing (in addition to reading).

The proof rules for reading and writing heap locations are as follows:

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

where  $p$  is either  $R$  or  $-R$ . Note that, using the Frame rule of Separation Logic, we can also conclude  $\{E \vdash^1 E'\} x := [E] \{E \vdash^1 E' \wedge x = E\}$ .

In addition to the processes themselves, permissions are also deposited in the resource invariant. When accessing a resource, a process grabs the heap cells described by the invariant along with their permissions: the conjunction  $(i \vdash^p j) \star (i \vdash^q j)$  is equivalent to providing access  $i \vdash^{p \star q} j$ . We often write  $i \vdash^1 j$  as simply  $i \mapsto j$ . The notation  $i \overset{p}{\leftarrow} j$ , inherited from Reynolds [13], means  $i \vdash^p j \star \text{true}$ . This is an “intuitionistic” predicate.

#### 4.1 Heap permissions

As a prequel to the proof of the program, let us present how the heap storage is structured.

Each heap item in the DLMSS algorithm consists of three fields: a left pointer, a right pointer and a colour. We denote the three fields by  $i.\text{left}$ ,  $i.\text{right}$  and  $i.\text{colour}$ .

The *storage invariant* says the heap is divided into three separate parts: a graph of reachable nodes, a list of free nodes, the remainder being garbage nodes.

$$SI(U, V, W) \stackrel{def}{=} \text{reachGraph}^R(U, V) \star \text{free}^R \text{List}^F(V) \star \text{garbage}^F(W)$$

The superscripts  $R$  and  $F$  have to do with the permissions for the various cells held in the storage invariant. Roughly speaking, the invariant holds something close to a *read* permission for the cells in the reachable graph and the free list header, but something close to a *full* permission for the remainder of the free list and all the garbage cells. The permissions for heap cells are defined in more detail below.

Let the predicate  $\text{cells}^p$  define an arbitrary set of heap data with permission  $p$ :

$$\text{cells}^p(X) \stackrel{def}{=} \prod_{i \in X} \exists j \in [0..N], k \in [0..N], c : i \vdash^p (j, k, c)$$

where  $\prod$  stands for iterated separating conjunction. The predicate  $\text{reachGraph}^p$  defines permission  $p$  to a directed graph of nodes reachable from a *root* node:

$$\begin{aligned} \text{reachGraph}^p(U, V) &\stackrel{def}{=} \text{cells}^p(U) \wedge \forall i. i \in U \iff (i = \text{nil} \vee \text{path}^p(\text{root}, i)) \wedge i \notin V \\ \text{path}^p(j, i) &\equiv j = i \vee \exists k. \text{edge}^p(j, k) \wedge \text{path}^p(k, i) \\ \text{edge}^p(j, k) &\equiv j \overset{p}{\leftarrow} (k, -, -) \vee j \overset{p}{\leftarrow} (-, k, -) \end{aligned}$$

Note that  $U$  is the set of nodes reachable from *root* except for the nodes in  $V$ . The exception list  $V$  is used temporarily when the graph needs to link in nodes from the free list.

The predicate  $\text{free}^p \text{List}^q$  similarly defines a free list reachable from a header node *free*. The free list consists of standard heap cells where the right pointer of a node is always set to *NIL*. The free list ends at *endfree*, which is a special tail node that is not part of  $V$ . The first free node of the free list (next to the

header node) is special because it is available for the mutator to be extracted. The invariant holds an  $R$  permission to this node, but full permission to the rest of the nodes in the free list until *endfree*.

$$\begin{aligned}
free^P List^a(V) &\stackrel{def}{=} \exists f, g, V_1, V_2 : \\
&V = \{free\} \uplus V_1 \uplus V_2 \wedge \\
&(free \vdash^P (f, nil, \_) \star listseg^p(f, g, V_1) \star listseg^a(g, endfree, V_2)) \wedge \\
&|V_1| \leq 1 \wedge (|V_1| = 0 \supset |V_2| = 0) \\
listseg^a(j, e, V) &\stackrel{def}{=} (j = e \wedge V = \emptyset \wedge \mathbf{emp}) \vee \\
&\exists k : j \in V \wedge (j \vdash^a (k, nil, \_) \star listseg^a(k, e, V \setminus \{j\}))
\end{aligned}$$

Note that the condition  $|V_1| \leq 1 \wedge (|V_1| = 0 \supset |V_2| = 0)$  can also be expressed as  $|V_1| \leq 1 \wedge (|V_1| = 0 \supset f = endfree)$  or as  $|V_1| = 1 \vee (|V_1| = 0 \wedge f = endfree)$ .

The predicate *garbage<sup>P</sup>* says nothing in particular about its constituents.

$$garbage^P(W) \stackrel{def}{=} cells^P(W)$$

These three predicates *uniquely* partition the heap data into three disjoint parts, i.e., if a heap satisfies  $reachGraph(U, V) \star freeList(V) \star garbage(W)$  as well as  $reachGraph(U', V') \star freeList(V') \star garbage(W')$  then  $U = U'$ ,  $V = V'$  and  $W = W'$ . All the nodes reachable from *free* except *endfree* comprise the set  $V$ , as well as  $V'$ , and all the nodes in the heap reachable from *root* except  $V$  comprise the set  $U$ , as well as  $U'$ . All the remaining nodes in the heap comprise the set  $W$  as well as  $W'$ . Hence,  $(U, V, W) = (U', V', W')$ .

Next, we define the permissions for heap cells under consideration. An  $R$  permission for a heap cell means that we have read permission for all its fields, but full permission for the colour field if the variable *swept*[*i*] is true:

$$\begin{aligned}
i \vdash^R (j, k, c) &\stackrel{def}{=} i.left \vdash^R j \star i.right \vdash^R k \star \\
&((i.colour \vdash^R c \wedge \neg swept[i]) \vee (i.colour \vdash^1 c \wedge swept[i]))
\end{aligned}$$

The control variable *swept*[*i*] allows a process to achieve “permission transfer” (like similar variables in [10] that achieve ownership transfer). By setting *swept*[*i*] to false, a process can retrieve a  $-R$  permission from the invariant and return it by setting *swept*[*i*] to true.

We define  $-R$  permission for a heap cell as just having  $-R$  permission for the link fields (and no access to the colour field). There is nothing deep about this; it just suits our purpose.

$$i \vdash^{-R} (j, k, c) \stackrel{def}{=} i.left \vdash^{-R} j \star i.right \vdash^{-R} k$$

We define the  $F$  permission for a heap cell in a similar way to the  $R$  permission:

$$\begin{aligned}
i \vdash^F (j, k, c) &\stackrel{def}{=} i.left \vdash^1 j \star i.right \vdash^1 k \star \\
&((i.colour \vdash^R c \wedge \neg swept[i]) \vee (i.colour \vdash^1 c \wedge swept[i]))
\end{aligned}$$

Note that:

$$i \xrightarrow{R} (j, k, c) \star i \xrightarrow{-R} (j, k, c) \iff i \xrightarrow{F} (j, k, c)$$

For all the cells in  $U \cup V \cup W$ , the storage invariant always holds at least a read permission for the colour field.

The reachable graph nodes are those which can be updated by the mutator process, the invariant only has read permission for the links. We will see below that the mutator will control write access to these links using “read complement” permissions.

The node pointed to by *free* has the read permissions as the reachable nodes, but the links of the rest of the free list nodes are under full control of the invariant. It is the collector process’s job to reclaim garbage nodes and hand them over to the free list.

## 4.2 Colour properties and the resource invariant

The global resource invariant is given by

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

We have already seen the storage invariant in the previous section. Its storage is expected to span all the cells numbered  $0..N - 1$ , except for the node *endfree*, making  $SI(U, V, W)$  a “precise” predicate. The remaining predicates state additional conditions for the same part of the storage in an “intuitionistic” way. Hence  $RI$  is a precise predicate.

The other components of the invariant maintain several properties of the heap nodes, which are detailed next. Notice that the atomic predicates below are intuitionistic and only assert a read permission for the invariant. The auxiliary predicates *marked* and *swept* are maintained by the collector process. The collector also maintains a pair of variables *leftgray* and *rightgray* which indicate nodes whose descendants are being greyed. A notion called *C*-edge was defined in [6], which we use as well:

$$Cedge(k, j) \stackrel{def}{=} k = leftgray \neq nil \wedge k \xrightarrow{R} (j, -, -) \vee k = rightgray \neq nil \wedge k \xrightarrow{R} (-, j, -)$$

*White invariant:* During the marking phase, every white reachable node is reachable from a gray reachable node, but without passing through a *C*-edge. During the sweeping phase, every white reachable node is unmarked.

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

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

$$\text{grayI}(X) \stackrel{\text{def}}{=} \text{in\_marking} \supset (\exists i : i \in X \wedge i.\text{colour} \xrightarrow{R} g) \supset \\ (\exists j : j \in [0..N] \wedge j.\text{colour} \xrightarrow{R} g \wedge \neg \text{marked}[j])$$

*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 the auxiliary shared variable *mod*, which is maintained by the mutator. This is initially established by colouring all the nodes white, and setting the auxiliary variables to *NIL*.

$$\text{bwI}(X) \stackrel{\text{def}}{=} \text{in\_marking} \supset \\ \forall k, j \in X : (\text{bwedge}(k, j) \vee \text{Cwedge}(k, j)) \supset k = \text{mod} \neq \text{nil} \\ \text{bwedge}(k, j) \stackrel{\text{def}}{=} (k \xrightarrow{R} (j, -, b) \vee k \xrightarrow{R} (-, j, b)) \wedge j.\text{colour} \xrightarrow{R} w \\ \text{Cwedge}(k, j) \stackrel{\text{def}}{=} \text{Cedge}(k, j) \wedge j.\text{colour} \xrightarrow{R} w$$

*Black invariant:* Unswept nodes can be gray or black and only unswept nodes can be black. The first conjunct equivalently says white nodes have to be swept, which is initially established.

$$\text{blackI} \stackrel{\text{def}}{=} (\forall i \in [0..N] : i.\text{colour} \xrightarrow{R} b \supset \neg \text{swept}[i]) \wedge \\ (\forall i \in [0..N] : \neg \text{swept}[i] \supset i.\text{colour} \xrightarrow{R} g \vee i.\text{colour} \xrightarrow{R} b)$$

Now that we have the resource invariant in place, the proof of the concurrent garbage collector can be presented. This is done in the next section.

## 5 The top-level proof

Now that we have the resource invariant in place, we come back to our task of proving the mutator and the collector. The required proofs are:

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

We deal with each in turn.

### 5.1 Mutator process

The mutator itself is a simple loop over its operations. Hence we have to show that each mutator operation preserves the **mutator invariant**

$$\text{mutI} \stackrel{\text{def}}{=} \exists U, V_0, f, g : \\ (\text{reachGraph}^{-R}(U, \emptyset) \star \text{freeHead}^{-R}(f, g, V_0)) \wedge k, j \in U \setminus \{\text{nil}\}$$

The predicate  $freeHead$  describes the head of the free list, which is a list segment of length at most 1:

$$freeHead^p(f, g, V) \stackrel{def}{=} (free \mapsto^p (f, nil, -) \star listseg^p(f, g, V)) \wedge ((|V| = 1 \wedge f = endfree) \vee (|V| = 0 \wedge f = endfree))$$

Even though the resource invariant allows the  $reachGraph$  to store pointers into the free list, our mutator is written so that the  $reachGraph$  is self-contained. There is no conflict here, because any heap that satisfies  $reachGraph(U, \emptyset)$  without encroaching on the free list also satisfies  $reachGraph(U, V)$ .

In its proof, the assertion  $mutI \star SI(U, V, W)$  allows the mutator to update the link fields of the nodes in  $reachGraph$  and the header of the free list (using the fact that  $-R \star R = 1$ ). It can also read the colour fields of all these nodes provided but it can only update the colour fields of the swept nodes. However, it must do so without mentioning the colours in its assertions. The rest of the  $freeList$  and  $garbage$  nodes do not appear in its assertions.

```
mutator  $\stackrel{def}{=}$ 
  var k,j,f: unsigned;
  do {mutI  $\wedge$  mod = nil}
    true  $\Rightarrow$  delete left edge(k);
  □ true  $\Rightarrow$  delete right edge(k);
  □ true  $\Rightarrow$  modify left edge(k,j);
  □ true  $\Rightarrow$  modify right edge(k,j);
  □ true  $\Rightarrow$  get new left edge(k);
  □ true  $\Rightarrow$  get new right edge(k);
  od
```

The proof outline of the mutator operations will appear in the next section.

## 5.2 Collector process

The collector repeatedly goes through a marking phase and a sweeping phase. Its proof uses the **collector invariant** which asserts its ownership of  $endfree$  and its permissions on colours as mediated by the  $swept$  array.

$$colI \stackrel{def}{=} endfree \mapsto^1 (-, -, -) \star unsweptI \wedge leftgray = nil \wedge rightgray = nil$$

$$unsweptI \stackrel{def}{=} \prod_{j \in [0..N]} (swept[j] \wedge \mathbf{emp} \vee \exists c. \neg swept[j] \wedge j.colour \mapsto^{-R} c \wedge c \in \{g, b\})$$

The collector extracts a  $-R$  permission for the colour fields of certain nodes by resetting  $swept[i]$ . This has the effect of prohibiting the mutator from changing these colours. The collector is then free to make assertions about these colours with the knowledge that they will not be falsified by the mutator.

```

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

```

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

### 5.3 Marking phase

Now we attempt the proof of the marking phase of the collector, which is an initialization followed by a loop over the marking operations. This make use of the marking invariant, which conjoins a few properties to the collector invariant.

$$markI(i) \stackrel{def}{=} colI \wedge in\_marking \wedge i \in [0..N+1] \wedge \forall j \in [0..N] : (marked[j] \iff j < i)$$

This is a local *loop invariant* of the collector process, and only has to hold at the beginning and end of each operation. Setting the *in\_marking* flag requires us to establishes the stronger version of the white invariant, viz., that every white reachable node is gray-reachable. Since all the nodes are initially swept, the black invariant implies that they are non-black. By greying the root nodes, we can make all the white reachable nodes gray-reachable. However, it is theoretically possible for a malicious mutator to spoil our efforts by turning these roots back to white. In the following, we prohibit this possibility by setting the swept flags of the roots.

```

mark  $\stackrel{def}{=}$ 
  {colI  $\wedge$   $\neg$ in_marking  $\wedge$   $\forall j \in [0..N] : \neg$ marked[j]  $\wedge$  swept[j]}
  <atleastgrey(ROOT); swept[ROOT] := false >;
  <atleastgrey(FREE); swept[FREE] := false >;
  <atleastgrey(NIL); swept[NIL] := false >;
  in_marking := true;
  i := 0;
  {markI(i)}
  do i  $\leq$  N  $\Rightarrow$  {markI(i)}
    if (c := [i.colour] |
      c  $\neq$  gray  $\Rightarrow$  | marked[i] := true);
      {markI(i+1)}
      i := i+1

```

```

    □ c = gray ⇒ | swept[i] := false;
                  {markI(i) ∧ ¬swept[i] ∧ i.colour  $\xrightarrow{-R}$  g}
                  restart run on gray node(i)
    fi
od ;
{(colI ∧ ∀j ∈ [0..N] : marked[j])}
in_marking := false {(colI ∧ ¬in_marking ∧ ∀j ∈ [0..N] : marked[j])}

```

In this program block, we have used atomic conditional branching to test the colours of nodes. As a result of the initialisation  $\langle c := [i.colour] \mid$  and the test  $c \neq gray$ , we conclude that  $i$  is non-gray. After setting  $marked[i]$  to true, the global invariants are restored (especially  $grayI$ ). In the case where the node is gray,  $swept[i]$  is set to false and the collector acquires read complement permission over the colour field. A longer sequence of statements, whose proof appears later, is used to blacken the node.

The postcondition of the marking phase asserts that all nodes are marked. Hence from the gray invariant, we get that there are no reachable gray nodes. Hence from the white invariant, we get that all white nodes are unreachable, that is, garbage. This is the basis for the sweeping phase and  $in\_marking$  can be set to false.

#### 5.4 Sweeping phase

Next we have the proof of the sweeping phase of the collector, which is a loop over the sweeping operations.

The proof uses the **sweeping invariant**  $sweepI(i)$ , which is a loop invariant which has to hold at the beginning and end of each operation:

$$sweepI(i) \stackrel{def}{=} colI \wedge \neg in\_marking \wedge i \in [0..N + 1] \wedge \forall j \in [0..N] : (j < i \supset \neg marked[j] \wedge swept[j]) \wedge (j \geq i \supset marked[j])$$

```

sweep  $\stackrel{def}{=}
  i := 0;
  {sweepI(i)}
  do i ≤ N ⇒ {sweepI(i)}
    if ⟨c := [i.colour] |
      c = white ⇒ | skip; {sweepI(i)}
                  collect white node(i)
    □ c = black ⇒ | skip; {sweepI(i) ∧ i.colour  $\xrightarrow{-R}$  b}
                  whiten black node(i)
                  {sweepI(i)}
    □ c = gray ⇒ | skip; {sweepI(i) ∧ i.colour  $\xrightarrow{-R}$  g}
                  skip gray node(i);
                  {sweepI(i)}$ 
```

```

                fi
    od
    {colI ∧ ¬in_marking ∧ ∀j ∈ [0..N] : swept[j] ∧ ¬marked[j]}

```

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

In the sweeping phase, whitening a node preserves  $bwI$  even though black-to-white edges might be introduced. From the black invariant, we have that no black nodes are left at the end of sweeping, hence again  $bwI$  is preserved. This is the basis for the marking phase which repeats after.

At this stage we have completed the top-level proof of the concurrent garbage collector, and are left with a few proofs of the operations of the mutator and of the collector during the marking and sweeping phases. It is these proofs which require the most careful analysis, since we have to deal with the nitty-gritty of our permissions model of shared variable accesses. We deal with these in the next section.

## 6 Proving the operations

We first list the basic operations on the shared variables. The first two are performed by the collector, whereas *atleastgray* is performed by both the mutator and collector processes.

```

whiten(i):  [i.colour] := white
blacken(i): [i.colour] := black
atleastgray(i):  if ⟨c := [i.colour] |
                  c = white ⇒ | [i.colour] := gray⟩
                □ c ≠ white ⇒ | skip⟩
                fi

```

We also gather all the resource and local invariants into the Table 1 for easy reference.

### 6.1 Mutator operations

We give proof outlines of the **delete** and **modify** operations of the mutator in Table 2. They use an operation called **addleft** for setting the left child of a node to a particular value.

We illustrate how a sample proof is performed by proving the **addleft** operation. The proof uses the auxiliary shared variable *mod*. 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.

$ \begin{aligned} RI &\stackrel{def}{=} \exists U, V, W, X: U \cup V \cup W = [0..N] \setminus \{endfree\} \wedge X = U \cup V \cup \{endfree\} \wedge \\ &\quad SI(U, V, W) \wedge whiteI(X) \wedge grayI(X) \wedge bwI(X) \wedge blackI \\ SI(U, V, W) &\stackrel{def}{=} reachGraph^R(U, V) \star free^R List^F(V) \star garbage^F(W) \\ whiteI(X) &\stackrel{def}{=} \forall i \in X: i.colour \xrightarrow{R} w \supset \\ &\quad (in\_marking \supset \exists j: j \in X \wedge gwpath(j, i)) \wedge \\ &\quad (\neg in\_marking \supset \neg marked[i]) \\ grayI(X) &\stackrel{def}{=} in\_marking \supset (\exists i: i \in X \wedge i.colour \xrightarrow{R} g) \supset \\ &\quad (\exists j: j \in [0..N] \wedge j.colour \xrightarrow{R} g \wedge \neg marked[j]) \\ bwI(X) &\stackrel{def}{=} in\_marking \supset \forall k, j \in X: (bwedge(k, j) \vee Cwedge(k, j)) \supset k = mod \neq nil \\ blackI &\stackrel{def}{=} (\forall i \in [0..N]: i.colour \xrightarrow{R} b \supset \neg swept[i]) \wedge \\ &\quad (\forall i \in [0..N]: \neg swept[i] \supset i.colour \xrightarrow{R} g \vee i.colour \xrightarrow{R} b) \\ mutI &\stackrel{def}{=} \exists U, V, f, g: \\ &\quad (reachGraph^{-R}(U, \emptyset) \star freeHead^{-R}(f, g, V)) \wedge k, j \in U \setminus \{nil\} \\ colI &\stackrel{def}{=} endfree \xrightarrow{1} (-, -, -) \wedge unsweptI \wedge leftgray = nil \wedge rightgray = nil \\ markI(i) &\stackrel{def}{=} colI \wedge in\_marking \wedge i \in [0..N + 1] \wedge \forall j \in [0..N]: (marked[j] \iff j < i) \\ sweepI(i) &\stackrel{def}{=} colI \wedge \neg in\_marking \wedge i \in [0..N + 1] \wedge \\ &\quad \forall j \in [0..N]: (j < i \supset \neg marked[j] \wedge swept[j]) \wedge (j \geq i \supset marked[j]) \end{aligned} $
---

**Table 1.** Resource and local invariants

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

$$\begin{aligned}
P &\equiv \exists U: reachGraph^{-R}(U, V) \wedge p \xrightarrow{-R} (l, m, -) \wedge p \neq nil \wedge q \in U \wedge mod = nil \\
C &\equiv [p.left] := q; mod := p; \\
Q &\equiv \exists U: reachGraph^{-R}(U, V) \wedge p \xrightarrow{-R} (q, m, -) \wedge q \in U \wedge mod = p \neq nil
\end{aligned}$$

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

$$P' \equiv \exists U: reachGraph^{-R}(U, V) \wedge p \xrightarrow{-R} (-, m, -) \wedge q \in U \wedge p = p \neq nil$$

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

- For the store invariant, the only node to worry about is  $l$ , the initial left child of  $p$ , since the edge from  $p$  to  $l$  has been removed.
  - If  $l$  is reachable from  $root$  then the postcondition retains the  $-R$  permission for it as part of  $reachGraph^{-R}(U, \emptyset)$ . A read permission is left with the invariant, as required.

<p><b>addleft(p, q):</b></p> $\{\exists U : reachGraph^{-R}(U, V) \wedge p \xrightarrow{-R} (l, m, \_) \wedge p \neq nil \wedge q \in U \wedge mod = nil\}$ $\langle [p.left] := q; mod := p; \rangle$ $\{\exists U : reachGraph^{-R}(U, V) \wedge p \xrightarrow{-R} (q, m, \_) \wedge q \in U \wedge mod = p \neq nil\}$ $\langle atleastgrey(q); mod := NIL \rangle$ $\{\exists U : reachGraph^{-R}(U, V) \wedge p \xrightarrow{-R} (q, m, \_) \wedge q \in U \wedge mod = nil\}$ <p><b>delete left edge(k):</b></p> $\{mutI \wedge mod = nil\}$ $addleft(k, NIL)$ $\{mutI \wedge mod = nil\}$ <p><b>modify left edge(k,j):</b></p> $\{mutI \wedge mod = nil\}$ $addleft(k, j)$ $\{mutI \wedge mod = nil\}$
--

**Table 2.** Mutator modify operations

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

Notice that, if  $l$  becomes unreachable, the node  $l$  silently moves in the storage invariant from  $reachGraph$  into  $garbage$ . This means a *permission transfer*: the

mutator retains no permissions on it in the postcondition and the invariant takes on a  $F$  permission. We will see below that this will enable the collector to later sweep this node into the free list.

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

$$\begin{aligned} C' &\equiv \text{atleastgrey}(q); \text{mod} := \text{NIL} \\ Q' &\equiv \exists U : \text{reachGraph}^{-R}(U, V) \wedge p \xrightarrow{-R} (q, m, -) \wedge q \in U \wedge \text{mod} = \text{nil} \end{aligned}$$

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

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

It is interesting to consider what happens if the two commands are interchanged, i.e., the definition of **addleft** is changed to:

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

But this cannot work. If the first command is to achieve some purpose then its post-condition must be able to assert that  $q$  is gray. However, the full permission for all the colour fields rests with the invariant. So, the postcondition has no way to assert the colour of  $q$ . (In fact, the colour of  $q$  can be changed by the collector before the second assignment takes place. It is said that an early version of the paper [6] had this problem.)

The proof of the **get** operation of the mutator, outlined in Table 3 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  $\text{reachGraph}^{-R}(U, V \cup \{n\})$ . These specifications should be easy for the reader to reconstruct if needed.)

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.

```

get new left node(k):
  { $\exists U, V : reachGraph^{-R}(U, \emptyset) \star freeHead^{-R}(-, -, V) \wedge k \in U \setminus \{nil\}$ }
  f := [free.left];
  { $\exists U, V : reachGraph^{-R}(U, \emptyset) \star freeHead^{-R}(f, -, V) \wedge k \in U \setminus \{nil\}$ }
  do f = ENDFREE  $\Rightarrow$  skip od ;
  { $\exists U, : reachGraph^{-R}(U, \emptyset) \star free \xrightarrow{-R} (f, nil, -) \star f \xrightarrow{-R} (-, nil, -) \wedge f \neq endfree \wedge k \in U \setminus \{nil\}$ }
  { $\exists U : reachGraph^{-R}(U, \emptyset) \star freeHead^{-R}(f, -, \{f\}) \wedge k \in U \setminus \{nil\}$ }
  m := [f.left];
  { $\exists U : reachGraph^{-R}(U, \emptyset) \star freeHead^{-R}(f, m, \{f\}) \wedge k \in U \setminus \{nil\}$ }
  addleft(k, f);
  { $\exists U : reachGraph^{-R}(U, \{f\}) \star freeHead^{-R}(f, m, \{f\}) \wedge k \in U \setminus \{nil\}$ }
  addleft(FREE, m);
  { $\exists U, V : reachGraph^{-R}(U, \{m\}) \star freeHead^{-R}(m, -, V) \wedge k \in U \setminus \{nil\}$ }
  addleft(f, NIL);
  { $\exists U, V : reachGraph^{-R}(U, \emptyset) \star freeHead^{-R}(-, -, V) \wedge k \in U \setminus \{nil\}$ }

```

**Table 3.** Mutator get operation

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  $reachGraph(U, \{f\})$  spans all the cells reachable from  $root$  except for  $f$ .
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:

$$\text{addleft(FREE, m); addleft(k, f); addleft(f, NIL);}$$

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 storage invariant  $SI(U, V, W)$  specifies all the nodes outside  $U, V$  and  $\{endfree\}$  to be in  $W$ , and the resource invariant has full permission for the nodes in  $W$ . So it is not possible to satisfy the precondition for  $\text{addleft(k, f)}$  which requires  $-R$  permission for  $f$ .

## 6.2 Operations during the marking phase

Next we look at the proof outline of the action of the collector when it encounters a gray node during the marking phase. This is shown in Table 4.

**Restart run on gray node(*i*):**

```

{markI(i) ∧ i.colour  $\xrightarrow{-R}$  g}
⟨atleastgrey(i.left); leftgray := i ⟩
⟨atleastgrey(i.right); rightgray := i ⟩
⟨blacken(i); leftgray := NIL; rightgray := NIL;
  for j := 0 to i-1 do marked[j] := false; ⟩
{markI(0)}
i := 0;
{markI(i)}
```

**Table 4.** Marking phase operations

Unlike the proof of the mutator, the collector has no direct permissions to the heap except for the cell *endfree*. All its actions are performed by borrowing permissions from the resource invariant in atomic operations.

The local invariant *markI*(*i*) is easily maintained, but each step has to maintain the resource invariant.

Observe that for the node  $l = i.left$  either *swept*[*l*] is true and the collector can grey it using the full permission of the invariant, or *swept*[*l*] is false and the collector can grey it putting together its  $-R$  permission with the invariant's *R* permission.

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

We cannot assert after greying *i.left* that it is not white, since the mutator may modify the left pointer after the greying, perhaps to a white node, leading to  $mod = leftgray = i$  holding. This is why the *C*-edges were introduced in [6].

The proof for greying the right child is symmetric. So let us come to the proof of the blackening step. First of all, *RI* together with the local permission  $i.color \xrightarrow{-R} g$ , allows write access to *i.colour*. The local postconditions hold, so we have to show that *RI* is re-established. The black invariant holds as *swept*[*i*] is false. The gray invariant holds since all nodes are unmarked. The white invariant holds since if a white node was gray-reachable using a path without *C*-edges (and hence without the edges from *i* to its children), such a path is unaffected.

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

### 6.3 Sweeping operations

Next we have the proof of the operations during the sweeping phase of the collector in Table 5.

<p><b>Skip gray node(i):</b></p> $\{sweepI(i) \wedge i.colour \xrightarrow{-R} g\}$ $swept[i] := true; marked[i] := false; i := i+1;$ $\{sweepI(i)\}$ <p><b>Whiten black node(i):</b></p> $\{sweepI(i) \wedge i.colour \xrightarrow{-R} b\}$ $\langle whiten(i); swept[i] := true; marked[i] := false \rangle;$ $\{sweepI(i+1)\}$ $i := i+1$ $\{sweepI(i)\}$ <p><b>Collect white node(i):</b></p> $marked[i] := false;$ $\{sweepI(i+1) \wedge endfree \xrightarrow{1} (-, nil, -)\}$ $[ENDFREE.left] := i;$ $\{sweepI(i+1) \wedge endfree \xrightarrow{1} (i, nil, -)\}$ $ENDFREE := i;$ $\{sweepI(i+1) \wedge endfree \xrightarrow{1} (-, -, -)\}$ $[ENDFREE.left] := NIL; [ENDFREE.right] := NIL;$ $\{sweepI(i+1) \wedge endfree \xrightarrow{1} (nil, nil, -)\}$ $i := i+1$ $\{sweepI(i)\}$
--

**Table 5.** Sweeping phase operations

The two assignments accompanying whitening the node preserve the white invariant.

We illustrate the proof of the *raison d'être* of this program, the **collect** action. Since by the sweeping invariant, *in\_marking* is false, from the white invariant a white marked node has to be outside  $X$ , hence in the unreachable *garbage*. The

invariant has full permission on its links and, since  $swept[i]$  is true by the black invariant, the colour field as well. The cell ENDFREE is owned by the collector. So, it is possible to link in the node  $i$  as the successor to ENDFREE. When the pointer ENDFREE is moved to  $i$ , the former ENDFREE node becomes part of the free list, and its permission is transferred to the resource invariant. At the same time, since ENDFREE cell is not part of the resource, the invariant gives up its full permission to the node  $i$ . Now, the collector has acquired full access to the node  $i$  and it proceeds to clean it up before turning to the next heap cell.

## 7 Conclusion

Separation Logic was initially conceived as a logic to conveniently reason about spatial separation of program components. However, it is slowly emerging that the notion of separation can be stretched by inventing novel kinds of components. O’Hearn [10] 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 recent work, Parkinson et al [12] 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 should admit that 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 [6, 7], but our methods in turn posed their own challenges. The main difference from the proof of Gries is that our proof is based on global invariants, which is more modular than the former but less flexible in the treatment of interference between processes. This is exhibited in the number of auxiliary variables that we needed to introduce (4 scalar variables and 2 arrays) compared to the one scalar variable required in Gries’s proof. On balance, the invariant-based proof is modular and, hence, less work is involved in checking for interference between processes.

## References

1. M. Ben-Ari. Algorithms for On-the-fly Garbage Collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.
2. L. Birkedal, N. Torp-Smith, and J. Reynolds. Local Reasoning about a Copying Garbage Collector. In *Symposium on Principles of Programming Languages*, pages 220–231. ACM Press, 2004.

3. 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.
4. S. Brookes. A Semantics for Concurrent Separation Logic. In P. Gardner and N. Yoshida, editors, *CONCUR*, volume 3170 of *LNCS*, pages 16–34, 2004.
5. L. Burdy. B vs Coq to Prove a Garbage Collector. Technical Report EDI-INF-RR-0046, University Edinburgh, September 2001.
6. 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.
7. D. Gries. An Exercise in Proving Parallel Programs Correct. *Communications of the ACM*, 20(12):921–930, December 1977.
8. K. Havelund. Mechanical Verification of a Garbage Collector. In J. D. P. Rolim et al, editor, *Parallel and Distributed Processing*, volume 1586 of *LNCS*, pages 1258–1283. Springer-Verlag, 1999.
9. 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 *LNCS*, pages 619–628, 2000.
10. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In P. Gardner and N. Yoshida, editors, *15th International Conference on Concurrency Theory (CONCUR)*, volume 3170 of *LNCS*, pages 49–76, 2004.
11. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
12. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Principles of Programming Languages*, page (to appear), 2007.
13. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.
14. D. M. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects Computing*, 6(4):359–390, 1994.
15. M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving Derivation of Concurrent Garbage Collection Algorithms. In *PLDI*, 2006.
16. H. Yang. An Example of Local Reasoning in BI Pointer Logic: the Schorr-Waite Graph Marking Algorithm, 2000.