A Discipline for Program Verification based on Backpointers and its Use in Observational Disjointness

Ioannis T. Kassios[1] and Eleftherios Kritikos[2]

[1] ETH Zurich, Switzerland
ioannis.kassios@inf.ethz.ch
[2] National Technical University of Athens, Greece
eleftherios.kritikos@gmail.com

Abstract. In the verification of programs that manipulate the heap, logics that emphasize *localized reasoning*, such as separation logic, are being used extensively. In such logics, state conditions may only refer to parts of the heap that are reachable from the stack. However, the correct implementation of some data structures is based on state conditions that depend on unreachable locations. For example, reference counting depends on the invariant that "*the number of nodes pointing to a certain node is equal to its reference counter*". Such conditions are cumbersome or even impossible to formalize in existing variants of separation logic.
In the first part of this paper, we develop a minimal programming discipline that enables the programmer to soundly express *backpointer conditions*, i.e., state conditions that involve heap objects that *point* to the reachable part of the heap, such as the above-mentioned reference counting invariant.
In the second part, we demonstrate the expressiveness of our methodology by verifying the implementation of *concurrent copy-on-write lists* (CCoWL). CCoWL is a data structure with *observational disjointness*, i.e., its specification *pretends* that different lists depend on disjoint parts of the heap, so that separation logic reasoning is made easy, while its implementation uses sharing to maximize performance. The CCoWL case study is a very challenging problem, to which we are not aware of any other solutions.

1 Introduction

The advent of *separation logic* [16] has revolutionized reasoning about programs with rich heap structure. The main motivation behind this line of work is *localized reasoning* (also referred to as "*reasoning in the small*"). In particular, the specifier is only allowed to talk about the locations of the heap s/he has explicit permission to, completely ignoring the rest of the heap. In separation logic, a state condition *contains its own permissions*. For example, $x \mapsto 3$ is a condition that not only expresses the fact that the number of 3 is the content of memory location x, but also that the programmer is permitted to read and write to x.

State conditions that contain their own permissions are called *stable* (*self-framing* in other literature [15, 18, 10]). A stable assertion has the important property that it *cannot be falsified by an unknown program*. As a result, the *localized* verification of our program cannot be falsified when this program is composed (sequentially, parallely, through method call, or through thread forking) with other programs. In concurrent variants of separation logic, permissions can be split [2] (e.g., in fractions [3]), thus enabling shared resources without data races. These well-known extensions of separation logic, maintain this important property: all expressible state conditions are stable.

Stable conditions cannot talk about objects that are unreachable by the pointers of the program under verification. However, there are cases when such conditions would be desirable.

For example, assume that we have a concurrent program operating on a graph. Normally, none of its threads has access to the whole graph, because that would mean that only one thread can perform changes, which defeats the purpose of concurrency. Consider now the following examples of *node invariants*:

- *Reference counting*. The value of the reference counter of a node N is equal to the number of nodes N' such that $N'.f = N$.
- *Priority Inversion Protocol* [17]. The priority of a node is the minimum of its initial priority and the priority of the node pointing to it (see also [20]).
- *The union-find structure*. In this structure each node represents a set of nodes. The set represented by a node N is $\{N\}$ unioned with the sets represented by all nodes that point to N.

Assume that a thread T has access to a node N. The invariant of N involves nodes that are unreachable from N, and therefore inaccessible to T. This makes the invariant of N unstable, and therefore inexpressible in existing variants of separation logic.

All the examples of node invariants that we mentioned are *conditions which may involve unreachable heap objects that point to reachable heap objects*. We call such conditions *backpointer conditions*. Our purpose is to enable the "reasoning in the small" style of separation logic, in verification problems that involve backpointer conditions.

1.1 Contributions

In this paper, we propose an extension of separation logic with a minimal programming discipline that makes it possible to express backpointer conditions in a stable way. Our methodology enables the verification, in the localized style of separation logic, of data structures with backpointer node invariants.

Furthermore, we use our technique to verify the case study of *concurrent copy-on-write lists* (hence CCoWL). This is a challenging problem of *observational disjointness*: the structure pretends that it supports mutually disjoint mutable sequences of integers, even though it uses data sharing under the hood, to enhance performance. The clients are happy to use the facilities of separation logic to verify their programs as if the lists were actually heap-disjoint, but

the verifier of the implementation is faced with a challenging reference counting mechanism. We are not aware of other solutions to the CCoWL verification problem.

We view the verification of CCoWL not just as a solution to a difficult problem, but also as a technique of backpointer bookkeeping that can serve as an example for the verification of similar data structures; perhaps even as a first step towards a generalized technique to verify *fully persistent data structures* [6].

1.2 Structure of the Paper

The paper is organized as follows: In Sect. 2, we motivate and introduce the discipline of backpointers. In Sect. 3, we show how the discipline can be used to verify CCoWLs, highlighting the most important parts of the implementation and the correctness proof. In Sect. 4, we discuss the relationship of our methodology to related work and point out some possibilities for future work. Sect. 5 concludes. In Appendix A, we give the specification, the implementation, and the proof of correctness of all the procedures of the CCoWL example.

2 The Backpointers Discipline

In this section, we introduce the discipline of backpointers. We start by introducing the background (Sect. 2.1) on which we work, a framework for locking, monitor invariants, and deadlock avoidance borrowed by Chalice [11]. We then provide a simple motivating example (Sect. 2.2) and we use it to extend our language with the backpointer formalism (Sect. 2.3).

Throughout the rest of the paper, familiarity with separation logic, abstract predicates, and fractional permissions is assumed.

2.1 Background

Records and Locking. Our language supports mutable records. A *monitor* is associated with each record and a *monitor invariant* is also associated with each monitor. The monitor invariant is an expression written in separation logic with fractional permissions.

Consider the following definition:

```
struct Pair
{
  x, y: int
  invariant ∃X, Y ∈ ℤ · (this.x ↦ X * this.y ↦^0.5 Y) ∧ X≥0
}
```

The definition introduces a set `Pair`. The members of `Pair` are: (a) the special record **null** and (b) records r such that $r\,.\,\mathsf{x}$ and $r\,.\,\mathsf{y}$ are heap locations that store integers.

Assume that `this` is a non-null record of type `Pair`. The monitor invariant associated with `this` asserts that `this.x` stores a positive value. It also grants write (full) access permission to `this.x` and 50% permission to `this.y`. In general, when we write monitor invariants, `this` refers to the current record and may be omitted when referring to its fields.

We are interested in thread-modular verification. From the point of view of the current thread, a record can be in one of the following three conditions: (a) local, (b) shared and not held by the current thread, (c) shared and held by the current thread. Fig. 1 shows all these conditions, together with the commands that perform the transitions between them.

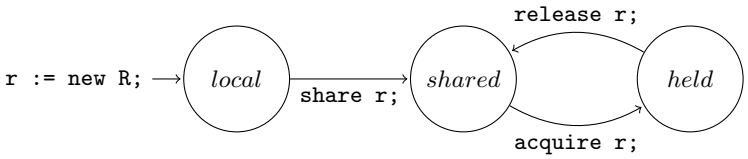

Fig. 1. A record's life cycle

The invariant of a monitor is always true when the associated record is shared but not held by any thread. To hold a record, a thread must *acquire* it. As long as it holds the record, the thread may invalidate the monitor invariant but must ensure that the invariant holds before it *releases* the record. Similarly, a thread that *shares* a record must first ensure that the associated invariant holds.

Sharing and releasing means that the current thread loses all permissions that are contained in the invariant. Acquiring means that the thread gains these permissions and that it may furthermore assume that the invariant holds immediately after the record is acquired.

The Chalice locking model has a simple mechanism to prevent cyclic dependencies between "acquire" requests, and thus to prevent deadlock. Assume that **Ord** is a set equipped with a strict partial ordering $\sqsubset$. We furthermore assume that $\sqsubset$ is *dense* in the sense that if $a \sqsubset b$ then there exists $c \in \mathbf{Ord}$ such that $a \sqsubset c \sqsubset b$. Every shared record is associated with a value in **Ord** called its *lock-level*. The maximum lock-level of all already acquired records by this thread is called its *max-lock*. A thread can only acquire a record with lock-level greater than its max-lock.

The rules that govern record creation, sharing, releasing and acquiring are shown in Fig. 2. In it:

- `local` and `shared` are abstract predicates that indicate that a record is resp. local or shared. The second argument of `shared` equals the lock-level of the record.
- Both predicates imply that their first argument, the record, is non-null:

$$\mathsf{shared}(r,\_) \ \vee \ \mathsf{local}(r) \ \Rightarrow r \neq \mathbf{null}$$

– `shared` is infinitely divisible, i.e.,

$$\mathsf{shared}(r,\ \mu) \iff \mathsf{shared}(r,\ \mu) \ * \ \mathsf{shared}(r,\ \mu)$$

– Each shared record has a single lock-level:

$$\mathsf{shared}(r,\ \mu) \ * \ \mathsf{shared}(r,\ \mu') \ \Rightarrow \mu = \mu'$$

– If r, r' are records, the notation $r \sqsubset r'$ is a shorthand for:

$$\exists\ \mu,\ \mu' \in \mathbf{Ord} \cdot \quad \mathsf{shared}(r,\ \mu) \ * \ \mathsf{shared}(r',\ \mu') \ * \ \mu \sqsubset \mu'$$

We extend this notation to "compare" a record r to a set of records:

$$R \ \sqsubset r \iff \forall r' \in R \cdot r' \sqsubset r$$

Note that $R \ \sqsubset r \ \Rightarrow r \notin R$

– $Inv(r)$ is the monitor invariant of record r

– `held` is a thread-local variable whose value is the set of all records held by the current thread

– `newRec` is an abstract predicate describing the situation directly after a new record is created. It gives access to all fields f_i of the new record r, initializes them to the default value of their type and asserts that r is local:

$$\mathsf{newRec}(r) \iff r.f_1 \mapsto d_1 \ * \ \ldots \ * \ \mathsf{r}.f_n \mapsto d_n \ * \ \mathsf{local}(r)$$

– The default value of all record types is **null**

$\{\mathbf{emp}\}$
`r:=new R`
$\{\mathsf{newRec(r)}\}$

$\{\mathsf{local(r)} \ * \ Inv(\mathsf{r}) \ * \ \mathsf{held} \mapsto O\}$
`share r`
$\{\mathsf{shared(r,\_)} \ * \ \mathsf{held} \mapsto O \ * \ O \sqsubset \mathsf{r}\}$

$\{\mathsf{shared(r,\_)} \ * \ \mathsf{held} \mapsto O \ * \ O \sqsubset \mathsf{r}\}$
`acquire r`
$\{\mathsf{shared(r)} \ * \ \mathsf{held} \mapsto O \cup \{\mathsf{r}\} \ * \ Inv(\mathsf{r})\}$

$\{(\mathsf{shared(r)} \ * \ \mathsf{held} \mapsto O \ * \ Inv(\mathsf{r})) \ \wedge \ \mathsf{r} \in O\}$
`release r`
$\{\mathsf{shared(r)} \ * \ \mathsf{held} \mapsto O - \{\mathsf{r}\}\}$

Fig. 2. Commands on Records and Monitors

The **share** command can specify bounds for the lock-level of the record being shared, for example:

```
share r below a
share r above b
share r between a and b
```

We omit the rules for these variants of **share** for brevity.

Counting Permissions. *Counting permissions* are an important alternative to fractional permissions. The idea is as follows. A counting permission is a natural number n, or -1. At any given execution time, there is one thread that holds a non-negative counting permission n to a heap location and n threads that hold a -1 counting permission. We call the holder of counting permission n the *main thread* for that heap location.

The main thread can give away -1 counting permissions, increasing its own counting permission accordingly. The holders of -1 counting permissions may return their counting permission to the main thread, decreasing its counting permission accordingly. If $n = 0$, then the main thread is the only thread that can access the location and thus has write privileges. Otherwise, all involved threads have read-only access.

We do not need to invent new notation for counting permissions. Instead, we introduce an infinitesimal fractional permission ϵ to stand for the -1 counting permission. Then the counting permission n corresponds to fractional permission $1 - n\epsilon$. This approach is taken in the current Chalice permission model [8].

2.2 Non-stable Backpointer Requirements

Suppose that we want a cell with reference counting. Our cell contains an integer field `value` and a reference counter field `refCount`. The value of the reference counter counts how many *clients* point to the cell.

```
struct Cell
{
  value, refCount: int
  invariant ∃R ∈ ℤ· (value ↦^{1−Rϵ} _ * refCount ↦ R) ∧ R≥1
}
struct Client { cell: Cell }
```

Notice that the monitor invariant holds $1 - R\epsilon$ permissions to the `value` field. It is intended that each client that points to a cell c holds an ϵ permission to $c.\mathtt{value}$. Thus the reference counter may inform a client whether or not it has sole access to `value` and may write to it.

To show how this works, define a predicate `cellvalue` as follows:

```
predicate cellvalue(this: Client, v: int)
{
  this≠null ∧
```

```
    ∃C∈Cell· shared(C,_) * this.cell↦C * C.value↦ϵv
}
```

The procedure `set`, shown in Fig. 3, checks the reference counter to see if it has full access to the cell, before actually updating the value

```
procedure set(this:Client, newValue:int)
  requires cellvalue(this,_)
  ensures cellvalue(this,newValue)
{
  acquire this.cell
  if(this.cell.refCount≠1)
  {
    release this.cell
    this.cell:=new Cell
    this.cell.value:=newValue
    this.cell.refCount:=1
    share this.cell
  }
  else
  {
    this.cell.value:=newValue
    release this.cell
  }
}
```

Fig. 3. Updating the Value of a Cell

The problem here is that the monitor invariant of `Cell` does not express the desired condition that `refCount` counts all clients pointing to the current cell. As a result, this condition may be inadvertently broken by a client.

For example, the procedure `set` of Fig. 3 has a subtle bug. When the reference count of `this.cell` is more than 1, then it creates a new cell and makes `this.cell` point to it. This removes a reference from the value of the old `this.cell`. However, the procedure `set` *forgets* to update the reference counter of the old value of `this.cell`. This causes an ϵ permission to the `value` field of that cell to be lost forever: this heap location is now accidentally rendered immutable. The error is not caught: the procedure `set` can be verified.

As we commented in the Introduction, the reason why we cannot express the reference counting condition is that such an condition would be *unstable*. Indeed, assignments to pointers outside the heap region that is reachable by a cell influence the reference counting condition of that cell.

2.3 Backpointers

To make the backpointer invariants stable, we impose a restriction on the assignments which may potentially invalidate such invariants. Since separation logic enforces restrictions to the programs through the use of permissions, we express our solution using permissions as well.

Tracked Fields. Our first step is to identify those reference-valued fields, whose value influences backpointer invariants. We mark these fields as *tracked*, because we want to track assignments to them. In our running example, the `cell` field of the `Client` record time must be marked tracked:

```
struct Client { tracked cell: Cell }
```

Backpointer Definitional Axiom. Suppose now that a record type R has a tracked field f of type Q (where R, Q are not necessarily different). To express backpointer invariants in records of type Q, it should be possible to refer to "all records of type P that point to the current record of type Q through the field f". We write $q.(P.f)^{-1}$ to refer to "the set of all records $p : P$ such that $p.f = q$". In other words, the *definitional axiom* of backpointers is:

$$\forall p \in P, q \in Q - \{\mathbf{null}\} \cdot \; p \in q.(P.f)^{-1} \quad \Leftrightarrow \quad p.f = q \;\wedge\; p \neq \mathbf{null}$$

If P is clear from the context, we simply write $q.f^{-1}$.

In our running example, if `r` is a cell, then `r.cell`$^{-1}$ is the set of all client records `c` such that `c.cell`=`r`. The reference counting invariant of cell can be expressed as follows:

```
this.refCount=|this.cell^-1|
```

where $|X|$ returns the cardinality of set X. The keyword `this` can be omitted.

Backpointer Fields. The value of the expression $(P.f)^{-1}$ is not associated with any permission, which is what makes it unstable. To fix this, we turn $(P.f)^{-1}$ into a *a field* of Q. This field has access permissions like any regular field. However, it is a *ghost* field: it does not appear in the actual program; it is only part of its specification annotation. Furthermore, even "ghost assignments" to it are forbidden. In our running example, the reference counting condition must not only contain access permission to `refCount`, but to `cell`$^{-1}$ as well:

$$\exists S \in 2^{\mathsf{Cell}} \cdot \;\; \mathsf{refCount} \mapsto |S| \;\; * \;\; \mathsf{cell}^{-1} \mapsto S$$

We can conjoin the above to the invariant of cells, which is now stable and can express reference counting:

```
struct Cell
{
  value, refCount: int
  invariant ∃S ∈ 2^Cell · value ⟼[1−|S|ϵ] _ * refCount ↦ |S| * cell^-1 ↦ S
}
```

Tracked Assignments. Assume now that record A points to record B through a tracked field f. Consider the assignment:

$A.f{:=}C$

Notice that this assignment changes not only the value of $A.f$, but that of $B.f^{-1}$ and $C.f^{-1}$. The situation is depicted graphically in Fig. 4. Since the values of two backpointer fields are changed, the thread that executes the assignment *must have full permission to those fields*. In the case B or C are the **null** reference, then, of course, we do not require access to their backpointer fields. The new rule for assignment to track fields becomes:

$$
\begin{array}{l}
\{ \\
\quad A{\neq}\textbf{null} \ \wedge \\
\quad (\ (A.f \mapsto B) \\
\quad * \ (B{\neq}\textbf{null} \ \Rightarrow B.f^{-1} \mapsto S_1) \ * \ (C{\neq}\textbf{null} \ \Rightarrow C.f^{-1} \mapsto S_2) \\
\quad) \\
\} \\
\ A.f{:=}C \\
\{ \\
\quad A{\neq}\textbf{null} \ \wedge \\
\quad (\\
\quad\quad (A.f \mapsto C) \\
\quad * \ (B{\neq}\textbf{null} \ \Rightarrow B.f^{-1} \mapsto S_1 - \{A\}) \ * \ (C{\neq}\textbf{null} \ \Rightarrow C.f^{-1} \mapsto S_2 \cup \{A\}) \\
\quad) \\
\}
\end{array}
$$

In our running example, consider the command `this.cell:=new Cell` in the implementation of the `set` procedure (Fig. 3). Before executing the assignment, the thread should have full permission to the field `this.cell`, to the field $\texttt{this.cell.cell}^{-1}$, and to the field $n.\texttt{cell}^{-1}$, where n is the newly created cell. The latter permission is available, because a thread has all permissions to the fields of a newly created record. The first permission is available because the thread has the predicate `cellvalue`. However, the thread does not have permission to the field $\texttt{this.cell.cell}^{-1}$, which *makes the assignment illegal*.

To fix this problem, the program must get the permission to the field $\texttt{this.cell.cell}^{-1}$. To do that, it must acquire `this.cell`:

```
...
acquire this.cell;
this.cell:=new Cell; // the assignment is possible now
...
```

However, after executing the assignment, the invariant $\texttt{refCount} \mapsto |S|$ of the old cell is not valid anymore. The program may not release the old cell, unless it fixes its reference counter. The solution fixes the bug that we spotted earlier. The command `this.cell:=new Cell` should be substituted by the following:

```
temp:=this.cell;
```

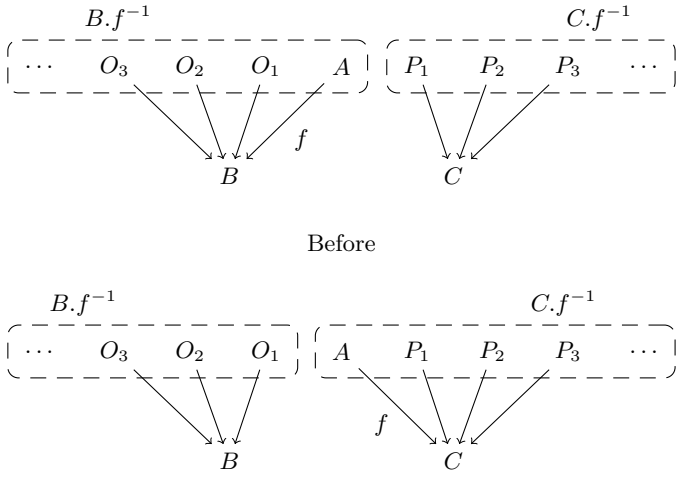

Fig. 4. Assignment on a tracked field

```
acquire temp;
this.cell:=new Cell;
temp.refCount:=temp.refCount−1;
release temp
```

Default Value of Backpointer Fields. A backpointer field should be initialized to $\emptyset$. A creation command that assigns the new record to a heap location, such as `this.cell:=new Cell` should be understood as a shorthand for a creation command and an assignment. For example, `this.cell:=new Cell` is a shorthand for

```
temporary_stack_variable:=new Cell;
this.cell:=temporary_stack_variable
```

The creation command above sets the value of $\texttt{this.cell.cell}^{-1}$ to $\emptyset$. The tracked assignment that follows sets it to {this}.

Notice that without setting the backpointer fields initially to $\emptyset$, the command command **share** this of Fig. 3 would not be legal (we would not be able to prove its precondition).

3 Concurrent Copy-on-Write Lists

We now turn our attention to a hard verification problem, that of concurrent copy-on-write lists (CCoWL). We discuss how backpointers help us verify this data structure.

In this section, we highlight the most important aspects of the verification. The specifications, implementations, and proof outlines for all the procedures can be found in Appendix A.

3.1 Description of the Problem

A CCoWL data structure supports a record called "list", which represents a mutable sequence of integers. One can create new empty sequences, insert items to the beginning of an existing sequence, update an item at a specific index, and copy one sequence to another. For simplicity, we restrict ourselves to the operations mentioned here, which can already generate all possible graphs in the underlying data structure.

The clients of lists, which may be one or more threads, are given the impression that every list is completely heap-disjoint from all the others and thus can reason about mutations using ordinary separation logic. The specification of the procedures that are available to the clients is shown in Fig. 5[1]. In it, `list(`l`,`L`)` is an abstract predicate that expresses the fact that the list record l represents the integer sequence L, the operator $+\!\!+$ denotes concatenation, and the expression $L\langle i \rightarrow v \rangle$ denotes the sequence L with the content of index i updated to value v. Indexes are zero-based.

For example, consider the following client:

```
list1:=new List;
initEmpty(list1);
insert(list1, 3); insert(list1, 2); insert(list1, 1);
list2:=new List;
copy(list2, list1);
set(list1, 1, 4)
```

We can use ordinary separation logic and the specifications of Fig. 5 to prove that, at the end of the execution, `list1` contains the sequence $[1, 4, 3]$, and `list2` contains $[1, 2, 3]$.

Behind the scenes however, the data structure performs *lazy copying*: all operations are implemented with reference manipulations *as long as this does not influence the clients' disjointness illusion.* Copying happens only when necessary.

The underlying representation uses linearly linked lists of "node" records. First the implementation creates a such a linked list to represent that `list1` contains the sequence [1,2,3] (Fig. 6a). After that, a new list `list2` is created and it is initialized by copying `list1`. The client thinks that the lists are disjoint, but the implementation is being lazy: it just sets the head node reference of `list2` to point to the head node of `list1`, producing the situation in Fig. 6b.

[1] Notice in our specification of lists that before the client can call any operation on a list l, the client must ensure that it has not locked a record above l in the lock ordering. This is undesirable, as it exposes the fact that locks are being used by the implementation. The problem is due to a weakness of the deadlock-avoidance of Chalice, which is orthogonal to the problem at hand and which we plan to address in future work.

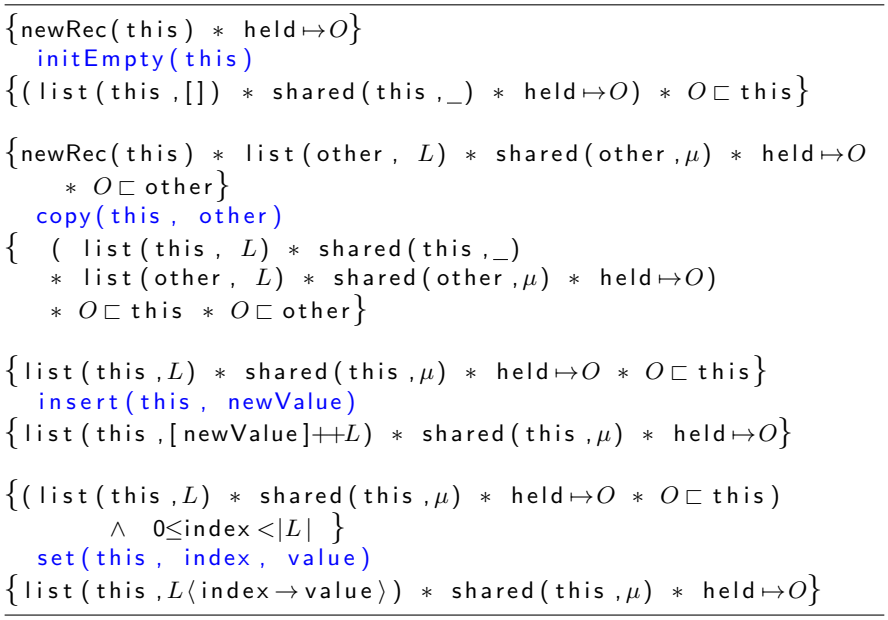

Fig. 5. Public Specification of CCoWLs

Finally, the client sets the item 1 of `list1` to 4. The change must influence only `list1` and not `list2`. The implementation must now copy the first two nodes of the common underlying structure, and then perform the set operation in a way that ensures that `list2` is not affected. The last node remains shared. The final situation is shown in Fig. 6c.

To achieve this copy-on-write effect, the nodes are equipped with a reference counter. When a `set` operation occurs, then the affected list is traversed from the head to the index where the update should happen. During the traversal, the reference counter of all the nodes is examined. As long as the reference count equals 1, the procedure knows that only one list is affected. As soon as the procedure meets a reference count greater than 1, it knows that, from that point on, more than one lists are affected. At that point, the procedure copies the nodes of the list all the way to the index where the update should happen.

Starting from Fig. 6c, a `set(list1, 1, 10)` operation will only find reference counts of 1 in its way and will perform no copying. On the contrary, `set(list1, 2, 10)` will find that the reference count of the node it is trying to mutate is 2, thus it must copy this node, separating the two lists completely.

3.2 Record Definitions, Abstract Predicates, and Invariants

Our implementation contains list and node records. A *list* record contains a reference to a *head node*. The reference should be tracked, because it should be counted in the reference count of the head node.

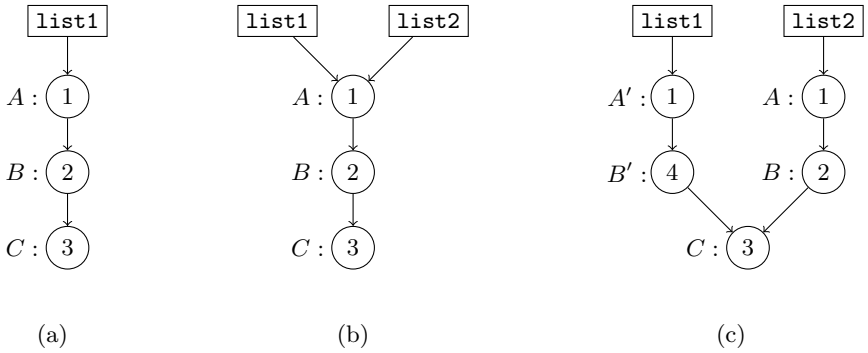

Fig. 6. An Example of CCoWL History

```
struct List { tracked head:Node }
```

If `head` points to **`null`**, then the list record represents the empty sequence.

A *node* record contains a value, a tracked reference to the next node, and a reference count. We defer the monitor invariant of nodes for later.

```
struct Node
{
  value, refCount:int;
  tracked next:Node;
  invariant ...
}
```

We now define the abstract predicate `list`. The definition uses the auxiliary abstract predicate `node`:

```
predicate list(this:List, L:ℤ*)
{
   ∃H∈Node· shared(this,_) * this.head↦H
     * ((node(H,L) * this⊏H) ∨ (H=null ∧ L=[]))
}

predicate node(this:Node, L:ℤ*)
{
   L≠[] ∧  ∃N∈Node·
        this.value↦ϵL[0] * this.next↦ϵ N * shared(this,_)
     * ((node(N,L[1..]) * this⊏N) ∨ (N=null ∧ |L|=1))
}
```

The predicate `node` traverses the structure following recursively the `next` references of the node records it encounters. The represented sequence is not empty. The first item `L[0]` of the sequence is stored in field `value`. The rest of

the sequence `L[1..]` is represented by the node pointed by field `next`, if one exists. The lock-order of node n is below that of $n.\texttt{next}$, because we intend to acquire monitors of nodes in the order in which we traverse the structure. Similarly, the lock-order of a list l is below that of $l.\texttt{head}$.

If a node record n is reachable from a list record l, then it contributes to the value of the sequence that l represents. We then say that l is *interested* in n.

Note that each holder of a $\texttt{list}(l, L)$ predicate has ϵ access to all the `value` and `next` fields of the nodes in which l is interested. The rest of the permissions to these fields are in the monitors of their respective records. So, if a node record interests n different lists, then it stores in its monitor $1 - n\epsilon$ permission to its fields `value` and `next`.

So far, this pattern is exactly the same as the one we have seen in the running example of Sect. 2.3. There is however a complication: the reference counter of a node does *not* indicate how many lists are interested in it. For example, consider Fig. 7, in which a possible state of a CCoWL structure is shown. Both nodes A and B interest three lists, however their reference counters are 2 and 1 respectively.

To deal with this problem, we introduce a *ghost* field in the record type `Node`. This field counts how many lists are interested in the current node. We call it `transRefCount` (for *transitive reference counter*). In Fig. 7, we see not only the reference counters but also the transitive reference counters of all the nodes.

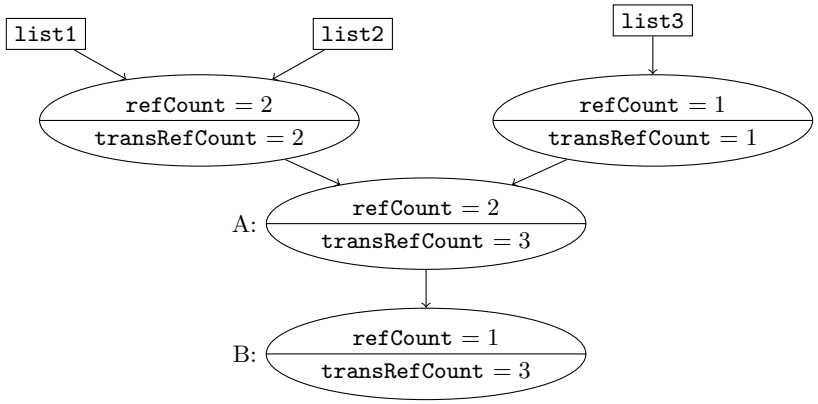

Fig. 7. Reference and Transitive Reference Counters in a CCoWL

We now know the following about the monitor invariant of the `Node` type:

– It grants permission $1 - T\epsilon$ to the fields `value` and `next`, where T is the value of the transitive reference counter:

$$\texttt{value} \overset{1-T\epsilon}{\longmapsto} V \;*\; \texttt{next} \overset{1-T\epsilon}{\longmapsto} N$$

– It grants full access to the fields head^{-1} and next^{-1}:

$$\mathsf{head}^{-1} \mapsto B_1 \ * \ \mathsf{next}^{-1} \mapsto B_2$$

– The value of the reference counter is equal to $|B_1|+|B_2|$. The field `refCount` is granted full access, as it should be possible for the thread that acquires the node to update the reference counter correctly:

$$\mathsf{refCount} \mapsto |B_1| + |B_2|$$

Notice that the value of the transitive reference counter is equal to the sum of *the transitive reference counters of all nodes that point to the current node* plus the number of list records that point directly to the current node. In order to be able to express this condition, we must grant to the monitor invariant of the current node *read access* to the `transRefCount` field of *all the nodes that point to the current node*. We give them 0.5 permission:

$$\exists F \in \mathsf{Node} \to \mathbb{Z} \cdot \circledast n \in B_2 \cdot n \,.\, \mathsf{transRefCount} \overset{0.5}{\longmapsto} F(n)$$

The value of the field `transRefCount` is given by

$$T \ = \ |B_1| \ + \ \sum n \in B_2 \cdot F(n)$$

The permission to the field `transRefCount` cannot be 1, since, as we have discussed above, the node N that follows the current one has 0.5 permission to it. Therefore, the invariant conjunct that relates `transRefCount` to its value is:

$$\mathsf{transRefCount} \overset{0.5}{\longmapsto} T$$

The final detail: if N is **null**, then there is no other node that has 0.5 permission to the current node's `transRefCount` field. In this case, the monitor invariant of the current node should include the extra permission:

$$N{=}\mathbf{null} \ \Rightarrow \mathsf{transRefCount} \overset{0.5}{\longmapsto} T$$

Putting it all together, the definition of `Node`, together with the monitor invariant, is:

```
struct Node
{
  value, refCount : int;
  ghost transRefCount : int;
  tracked next : Node;
  invariant ∃T ∈ ℤ, N ∈ Node, B1 ∈ 2^Head, B2 ∈ 2^Node, F ∈ Node → ℤ ·
    ( value ↦^{1−Tϵ} _ * next ↦^{1−Tϵ} N * head^{−1} ↦ B1 * next^{−1} ↦ B2
    * refCount ↦ |B1| + |B2| * transRefCount ↦^{0.5} T
    * (⊛ n ∈ B2 · n . transRefCount ↦^{0.5} F(n))
    * (N=null ⇒ transRefCount ↦^{0.5} T)
    ) ∧ T = |B1| + Σ n ∈ B2 · F(n)
}
```

3.3 Some Highlights of the Implementation

In this section, we discuss three interesting aspects of the implementation: how lists gain and lose interest to nodes and how the updating procedure decides how to substitute in-place update by copy-and-update.

Gaining Interest. In our procedures, the only place where a list gains interest to new nodes is lazy list copying. When a list is copied, only the head reference of the target list changes. The target list gains interest to all the nodes of the source list. To ensure that our bookkeeping is correct, we must update the transitive reference counters of all these nodes. We do this with a *ghost* procedure[2] `addOneToTransRefCount`, which traverses the whole list and adds 1 to all transtive reference counters.

Losing Interest. Our copy-and-update procedure `node_copy_set` takes as parameters (besides the obvious index/value pair) a *source* node `this` and a *target* node `new_node`. The precondition of `node_copy_set` asserts that the caller has a predicate `node(this,`L`)`. Its postcondition returns a predicate `node(new_node,`$L\langle$`index`$\rightarrow$`value`$\rangle$`)`. The predicate `node(this,`L`)` of the postcondition is lost. Indeed, the permissions `node(this,`L`)` are taken away from the thread. Those monitors of the nodes to which the source list loses interest obtain an extra ϵ permission to the corresponding `value` and `next` field. For the nodes to which no interest is lost, the thread maintains its ϵ permissions, but they are now part of the `node(new_node,`$L\langle$`index`$\rightarrow$`value`$\rangle$`)` predicate. In this way, no permission to fields `value` and `next` is ever lost.

For example, consider the situation in Fig. 6b. The permission to the `value` and `next` fields that is stored in the monitor of nodes A, B, C is $1 - 2\epsilon$. There is a thread that holds a `list(list1,[1,2,3])` predicate, that grants ϵ permission to the `value` and `next` fields of these nodes. Now `set(list1,1,4)` is called. Since the reference count of A is 2, a new node A' is created and the `node_copy_set` procedure is called with source A and target A'. The procedure takes away the `node(`A`,[1,2,3])` predicate of the caller and returns a new `node(`A'`,[1,4,3])` predicate. The final state is shown in Fig. 6c.

List `list1` lost interest to nodes A, B. The ϵ permissions to their `value` and `next` fields are returned from the `node(`A`,[1,2,3])` predicate back to their monitor, which now maintains $1-\epsilon$ permission to these fields. The list maintained interest to node C, so an ϵ permission to the `value` and `next` fields of C is transfered from `node(`A`,[1,2,3])` to `node(`A'`,[1,4,3])`. The monitor of C has $1-2\epsilon$ permission to those fields, as before. The predicate `node(`A'`,[1,4,3])` has ϵ permission to the `value` and `next` fields of the newly generated A' and B' nodes. There was no loss of permission; only permission transfer.

Setting without Copying. As we have explained, the algorithm decides to start the copy-and-update procedure once it sees a reference counter greater than 1.

[2] A ghost procedure updates the state by assigning only to ghost fields, and therefore is not executed in the actual program.

To verify that this policy is indeed correct, we include a precondition to our update-in-place procedure `node_set` that the transitive reference counter of the node it is applied to equals 1.

The algorithm calls `node_set` on the next node, under the circumstance "I have not yet seen a reference counter greater than 1 and the reference counter of the next node is 1". In our formalism, this circumstance is translated into the following condition:

$$(\ \mathsf{this.refCount} \mapsto 1\ *\ \mathsf{this.transRefCount} \overset{0.5}{\longmapsto} 1 \\ *\ \mathsf{this.next} \mapsto N\ *\ N.\mathsf{refCount} \mapsto 1)\ \wedge\ N \neq \mathbf{null}$$

From this condition, together with the fact that $Inv(\mathsf{this})$ and $Inv(N)$ hold, one must prove that the value of the transitive reference counter of N is 1. In the following, we explain how we prove this property.

Let B_1 be the value of $N.\mathsf{head}^{-1}$ and B_2 be the value of $N.\mathsf{next}^{-1}$. By the definitional axiom, we know that $\mathsf{this} \in B_2$. By $Inv(N)$, we conclude that $B_2 = \{\mathsf{this}\}$ and $B_1 = \emptyset$. Again by $Inv(N)$, we get that the value of $N.\mathsf{transRefCount}$ is equal to the value of $\mathsf{this.transRefCount}$, which is 1.

The above argument applies when `node_set` recursively calls itself. Initially however, it is procedure `set` (the update procedure on lists) that decides whether it should call `node_set` or `node_copy_set` on its head node. The argument for this decision is similar.

4 Discussion

In what follows, we discuss the relationship of our work with invariant disciplines and research on observational disjointness, as well as topics for future work.

Invariant Disciplines. An *invariant discipline* is a set of rules that specifiers and programmers have to follow to ensure that some state (or history) conditions remain true throughout a computation (or at specific states thereof). Some such conditions are independent of the program, for example, our methodology guarantees that $r \in r.f.f^{-1}$, in any state, for any non-null record r, and tracked field f such that $r.f \neq \mathbf{null}$. We call these conditions *system invariants*. Some other conditions are given by the programmer, for example *object* or *monitor invariants*. There are several flavors of treating program-specific invariants, mostly focusing on the special case of object invariants [7]. Various forms of ownership [1, 13] are popular invariant disciplines.

In [14], the point is made that object invariants are inflexible, in comparison to the use of abstract predicates. In [20], the authors answer by making the case for object invariants as an independent specification tool. Most of their arguments have to do with the usefulness of object invariants in practical software engineering contexts; but they also provide an example (the priority inheritance protocol [17]) as one in which object invariants can turn a seemingly global property (which, in our terminology, a backpointer property) into a local one. It

seems, the authors argue, that the priority inheritance protocol example is not easy to handle with abstract predicates alone.

Our paper provides a monitor invariant discipline that can handle such backpointer examples. The discipline consists of restricting the use of assignments to tracked fields. We have expressed our discipline not as a set of rules, as is common, but by using permissions in the separation logic style. Our proposal makes it possible to treat backpointer conditions as special cases of separation logic conditions, turning them into local properties, which supports the argument of [20], in the concurrent case.

Our verification of CCWoLs is influenced by *considerate reasoning* [19], a framework in which it is possible for a procedure to "notify" via specification annotations all interested parties about the object invariants that it might break. Our specification and implementation of `addOneToTransRefCount` is a direct adaptation of their `addToTotal` method.

Observational Disjointness. While separation logic has been a revolution in the specification of heap-intensive computations, it has been observed, especially in the context of concurrency, that the association of separating conjunction with actual heap separation is too restrictive: sometimes we want the client(s) to "observe" disjointness, but, at the same time, allow the implementers the opportunity to share heap under the hood.

In our work, we make use of a standard solution to loosen the heap disjointness requirement: fractional and counting permissions. Furthermore, our use of backpointers permits us to maintain bookkeeping information about the clients of observationally disjoint data structures. These two ingredients together suffice for the verification of the CCoWL case study.

Concurrent abstract predicates [5] support the hidden sharing of state with the use of *capabilities*, i.e., special predicates that allow exclusive access to a shared region. This idea has been successfully applied to the specification and verification of indexing structures [4]. The work presented here cannot substitute capabilities. On the other hand, it is not clear how one would handle the CCoWL example with CAPs. It seems that backpointers and CAPs are orthogonal tools and could be integrated into a single specification language.

Fictional Separation Logic [9] is an ambitious mathematical framework that allows the implementer to *choose* their own separation algebra as part of the implementation. This idea completely decouples heap disjointness from separating conjunction. The use of fractional permissions as well as other examples of observational disjointness are shown to be special cases of this very general methodology. The generality comes at the price of complexity at the part of the implementer, so it remains an open question if this idea scales up to reasonably-sized programs. Furthermore, it seems that fictional separation logic has no provision for object and monitor invariants, nor does it provide the means of mentioning unreachable parts of the heap, like we do.

In [12], the verification of *snapshotable trees* is proposed as a challenge. The problem is very similar to the CCoWLs: the clients see a mutable tree and immutable snapshots of previous states of that tree. A snapshot can be created

at any time. All snapshots and the tree appear to be heap-disjoint, but, in fact, the implementation uses lazy copying and shares as much as possible. There are four different versions of the structure, one of which is verified by the authors, using whole-heap predicates (and therefore restricting it to sequential programs).

The fact that snapshots are immutable is a very crucial difference compared to the CCoWL example, in which all lists are mutable. In the terminology of [6] snapshotable trees are *partially persistent*, while CCoWLs are *fully persistent*. The implementers of snapshotable trees need no permission accounting, because they do not wish to reclaim write permissions to the part of the structure that becomes immutable. Contrary to that, we ensure that no permissions are lost. For example, suppose that exactly two lists l_1, l_2 are interested in a node n. At this state, no thread can change the fields of n. Suppose now that l_2 loses interest. The fields of n become mutable again: the list l_1 may gain write permissions to them. To achieve this, the bookkeeping of backpointers is essential (see also Sect. 3.3, "losing interest").

Work in Progress and Future Work. The *automation* of the backpointer methodology is implemented as an extension of the Chalice verifier. Unfortunately, for many examples that we have tried, including CCoWLs, the verification times are prohibitively large. The improvement of our tool is work in progress.

The current version of the methodology works for object-based systems, but does not consider *subtyping and inheritance*, a topic for further research.

Finally, note that the specifications of CCoWLs expose a significant implementation detail to the client: the fact that the underlying implementation uses locks (see Fig. 5). Furthermore, they impose a non-trivial requirement about the locking behavior of the client. This weakness of the specification is due to the technicalities of the Chalice deadlock-avoidance policy. We are currently working to improve this deadlock-avoidance system.

5 Conclusion

We have introduced an invariant discipline to enhance the expressiveness of separation logic with backpointer conditions. We have used our methodology to specify and verify concurrent copy-on-write lists, a challenging case study of observational disjointness, which, to the best of our knowledge, has not been tackled before.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# specification language: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS'04*, volume 3362 of *Lecture Notes In Computer Science*, pages 49–69. Springer-Verlag, 2004.
2. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05*, pages 259–270, 2005.

3. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SA'03*, volume 2694 of *Lecture Notes In Computer Science*, pages 55–72. Springer-Verlag, 2003.
4. P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A simple abstraction for complex concurrent indexes. In *OOPSLA'11*, pages 845–864. ACM, 2011.
5. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP'10*, volume 6183 of *Lecture Notes In Computer Science*, pages 504–528. Springer-Verlag, 2010.
6. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC'86*, pages 109–121. ACM, 1986.
7. S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. A unified framework for verification techniques for object invariants. In *ECOOP'08*, Lecture Notes In Computer Science. Springer-Verlag, 2008.
8. S. Heule, K. R. M. Leino, P. Müller, and A. Summers. Fractional permissions without the fractions. In *FTfJP'11*, 2011.
9. J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP'12*, volume 7211 of *Lecture Notes In Computer Science*, pages 377–396. Springer-Verlag, 2012.
10. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM'06*, volume 4085 of *Lecture Notes In Computer Science*, pages 268–283. Springer-Verlag, 2006.
11. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP'09*, volume 5502 of *Lecture Notes In Computer Science*, pages 378–393. Springer-Verlag, 2009.
12. H. Mehnert, F. Sieczkowski, L. Birkedal, and P. Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE'12*, volume 7152 of *Lecture Notes In Computer Science*, pages 179–195. Springer-Verlag, 2012.
13. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes In Computer Science*. Springer-Verlag, 2002.
14. M. Parkinson. Class invariants: the end of the road? In *IWACO'07*, 2007.
15. M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. In Gilles Barthe, editor, *ESOP'11*, volume 6602 of *Lecture Notes In Computer Science*. Springer-Verlag, 2011.
16. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE Computer Society, 2002.
17. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.
18. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP'09*, Genoa, pages 148–172. Springer-Verlag, 2009.
19. A. Summers and S. Drossopoulou. Considerate reasoning and the composite design pattern. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI'10*, volume 5944 of *Lecture Notes In Computer Science*, pages 328–344. Springer-Verlag, 2010.
20. A. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *IWACO '09*, pages 1–9. ACM, 2009.

A Proof Outlines

In the appendix, we provide the full specification, implementation and proof outlines of the CCoWL verification.

Implementations

The procedure `initEmpty` initializes its argument to be an empty list

```
procedure initEmpty(this:List)
{
  share this
}
```

The procedure `copy` initializes its first list argument by copying (lazily) its second list argument.

```
procedure copy(this, other:List)
{
  share this below other;
  acquire this;
  acquire other;
  if(other.head≠null)
  {
    acquire other.head;
    this.head := other.head;
    this.head.refCount := this.head.refCount + 1;
    addOneToTransRefCount(this.head);
    release this.head
  }
  release other;
  release this
}
```

The procedure `insert` inserts an item at the beginning of a sequence represented by a list.

```
procedure insert(this:List, new_value:int)
{
  var n:Node;
  n:=new Node;
  n.value:=new_value;
  if(this.head≠null)
  {
    acquire this.head
  }
```

```
  n.next:=this.head;
  this.head:=n;
  this.head.refCount:=1;
  this.head.transRefCount:=1;
  if(this.head.next≠null)
  {
    release this.head.next
  }
  share head between this and head.next
}
```

The procedure `set` takes a list, an index, and a value and updates the represented sequence by changing the item at the position indicated by the index.

```
procedure set(this:List, index, new_value:int)
{
  var h:Node;
  acquire this.head;
  if(this.head.refCount=1)
  {
    node_set(this.head, index, value)
  }
  else
  {
    this.head.refCount:=this.head.refCount-1;
    h:=this.head;
    this.head:=new Node;
    node_set_copy(h, index, value, this.head)
  }
}
```

The procedure `node_set` updates the sequence represented by a node which is known to be of interest to exactly one list.

```
procedure node_set(this:Node, index, new_value:int)
{
  var h:Node;
  if(index=0)
  {
    this.value:=new_value;
    release this
  }
  else
  {
    acquire this.next;
    if(this.next.refCount=1)
```

```
    {
      release this;
      node_set(this.next, index-1, new_value)
    }
    else
    {
      this.next.refCount:=this.next.refCount-1;
      h:=this.next;
      this.next:=new Node;
      node_set_copy(h, index-1, new_value, this.next);
      release this
    }
  }
}
```

The procedure `node_set_copy` updates the sequence represented by a node by performing copying.

```
procedure node_set_copy
            (this:Node, index, new_value:int, new_node:Node)
{
  new_node.refCount:=1;
  new_node.transRefCount:=1;
  if(this.next≠null)
  {
    acquire this.next
  }
  this.transRefCount:=this.transRefCount-1;
  if(index=0)
  {
    new_node.value:=new_value;
    new_node.next:=this.next;
    if(this.next≠null)
    {
      this.next.refCount:=this.next.refCount+1
      release this.next
    }
  }
  else
  {
    new_node.value:=this.value;
    if(this.next≠null)
    {
      new_node.next:=new Node;
      node_set_copy
        (this.next, index-1, new_value, new_node.next)
    }
    else
```

```
    {
      new_node.next:=null
    }
  }
  share new_node between this and new_node.next;
  release this
}
```

The ghost procedure addOneToTransRefCount updates the transitive reference counters of all nodes in which a new list becomes interested.

```
ghost procedure addOneToTransRefCount(this:Node)
{
  if(this.next≠null)
  {
    acquire this.next
  }
  this.transRefCount := this.transRefCount + 1;
  if(this.next≠null)
  {
    addOneToTransRefCount(this.next);
    release this.next
  }
}
```

Useful Definitions for the Proofs

First we define some predicates to be used in the proofs.

The predicate inv expresses the fact that the monitor invariant of a node is true and, furthermore, the second parameter of inv is the value of the next field of that node:

```
predicate inv(this, nx:Node)
{
  Inv(this) ∧ this.next↦^ϵ nx
}
```

The predicate inv1 expresses the fact that the monitor invariant of a node is true and, furthermore, its reference counters are both equal to 1 (therefore it is possible for a holder of a node predicate to write directly to the fields of the node):

```
predicate inv1(this, nx:Node)
{
  ∃B1, B2, F ·
    ( this.value↦^{1−ϵ} _ * this.next↦^{1−ϵ} nx
    * this.head^{−1} ↦ B1 * this.next^{−1} ↦ B2
```

```
    * this.refCount↦1 * this.transRefCount⟼^0.5 1
    * (⊛n ∈ B2·n.transRefCount⟼^0.5 F(n))
    * (nx=null ⇒ this.transRefCount⟼^0.5 1)
    ) ∧ 1 = |B1| + (∑n ∈ B2·F(n)) = |B1|+|B2|
}
```

The predicate brokenT_inv expresses the fact that the monitor invariant of a node is true *except* the transitive reference counter is off by the third parameter:

```
predicate brokenT_inv(this, nx:Node, p:int)
{
  ∃T, N, B1, B2, F·
    ( this.value⟼^(1−Tε)_ * this.next⟼^(1−Tε)nx
    * this.head^−1↦B1 * this.next^−1↦B2
    * this.refCount↦|B1|+|B2| * this.transRefCount⟼^0.5 T+p
    * (⊛n ∈ B2·n.transRefCount⟼^0.5 F(n))
    * (nx=null ⇒ transRefCount⟼^0.5 T+p)
    ) ∧ T = |B1| + ∑n ∈ B2·F(n)
}
```

The predicate brokenR_inv expresses the fact that the monitor invariant of a node is true *except* the reference counter is off by the third parameter:

```
predicate brokenR_inv(this, nx:Node, p:int)
{
  ∃T, N, B1, B2, F·
    ( this.value⟼^(1−Tε)_ * this.next⟼^(1−Tε)nx
    * this.head^−1↦B1 * this.next^−1↦B2
    * this.refCount↦|B1|+|B2|+p * this.transRefCount⟼^0.5 T
    * (⊛n ∈ B2·n.transRefCount⟼^0.5 F(n))
    * (nx=null ⇒ transRefCount⟼^0.5 T)
    ) ∧ T = |B1| + ∑n ∈ B2·F(n)
}
```

The predicate brokenRT_inv expresses the fact that the monitor invariant of a node is true *except* the reference counter is off by the third parameter and that the transitive reference counter is off by the fourth parameter:

```
predicate brokenRT_inv(this, nx:Node, p1,p2:int)
{
  ∃T, N, B1, B2, F·
    ( this.value⟼^(1−Tε)_ * this.next⟼^(1−Tε)nx
    * this.head^−1↦B1 * this.next^−1↦B2
    * this.refCount↦|B1|+|B2|+p1 * this.transRefCount⟼^0.5 T+p2
    * (⊛n ∈ B2·n.transRefCount⟼^0.5 F(n))
    * (nx=null ⇒ transRefCount⟼^0.5 T+p2)
    ) ∧ T = |B1| + ∑n ∈ B2·F(n)
}
```

Specifications Proofs

Proof of initEmpty

```
procedure initEmpty(this:List)
  requires newRec(this) * held↦O
  ensures  list(this,[])
         * shared(this,_)
         * held↦O
         * O⊏this
{
    // note that the invariant of List is trivially true
  share this
    {    shared(this,_)
       * held↦O
       * O⊏this
       * this.head↦null
    }
⇒
    {    list(this,[])
       * shared(this,_)
       * held↦O
       * O⊏this
    }
}
```

Proof of copy

```
procedure copy(this, other:List)
  requires   newRec(this)
           * list(other, L)
           * shared(other,μ)
           * held↦O
           * O⊏other
  ensures   list(this, L)
          * shared(this,_)
          * list(other, L)
          * shared(other,μ)
          * held↦O
          * O⊏this
          * O⊏other
{
  share this below other;
    {    shared(this,_)
       * this.head↦null
       * list(other, L)
       * shared(other,μ)
       * held↦O
```

```
      * O ⊏ this ⊏ other
    }
  acquire this;
  acquire other;
    {    shared(this,_)
      * this.head↦null
      * list(other, L)
      * shared(other,μ)
      * held↦O∪{this,other}
      * O ⊏ this ⊏ other
    }
⇒
    {∃H·
         shared(this,_)
      * this.head↦null
      * other.head↦H
      * ((node(H,L) * other ⊏ H) ∨ (H=null ∧ L=[]))
      * shared(other,μ)
      * held↦O∪{this,other}
      * O ⊏ this ⊏ other
    }
  if(other.head≠null)
  {
      {∃H·
           shared(this,_)
        * this.head↦null
        * other.head↦H
        * node(H,L)
        * shared(other,μ)
        * held↦O∪{this,other}
        * O ⊏ this ⊏ other<H
      }
    acquire other.head;
      {∃H·
           shared(this,_)
        * this.head↦null
        * other.head↦H
        * node(H,L)
        * shared(other,μ)
        * held↦O∪{this,other,H}
        * Inv(H)
        * O ⊏ this ⊏ other<H
      }
    this.head := other.head;
      {∃H·
           shared(this,_)
        * this.head↦H
        * other.head↦H
```

```
        * node(H, L)
        * shared(other, μ)
        * held ↦ O ∪ {this, other, H}
        * brokenRT_inv(H, _, −1, −1)
        * O ⊏ this ⊏ other < H
    }
  this.head.refCount := this.head.refCount + 1;
    {∃H ·
        shared(this, _)
      * this.head ↦ H
      * other.head ↦ H
      * node(H, L)
      * shared(other, μ)
      * held ↦ O ∪ {this, other, H}
      * brokenT_inv(H, _, −1)
      * O ⊏ this ⊏ other < H
    }
  // see Sect.3.3, Losing Interest
  addOneToTransRefCount(this.head);
    {∃H ·
        shared(this, _)
      * this.head ↦ H
      * other.head ↦ H
      * node(H, L)
      * node(H, L)
      * shared(other, μ)
      * held ↦ O ∪ {this, other, H}
      * Inv(H)
      * O ⊏ this ⊏ other < H
    }
  release this.head
    {∃H ·
        shared(this, _)
      * this.head ↦ H
      * other.head ↦ H
      * node(H, L)
      * node(H, L)
      * shared(other, μ)
      * held ↦ O ∪ {this, other}
      * O ⊏ this ⊏ other < H
    }
⇒
    {   list(this, L)
      * shared(this, _)
      * list(other, L)
      * shared(other, μ)
      * held ↦ O ∪ {this, other}
      * O ⊏ this ⊏ other
```

```
      }
  }
    {     list(this,L)
        * shared(this,_)
        * list(other,L)
        * shared(other,μ)
        * held↦O∪{this,other}
        * O⊏this⊏other
    }
  release other;
  release this
    {     list(this,L)
        * shared(this,_)
        * list(other,L)
        * shared(other,μ)
        * held↦O
        * O⊏this⊏other
    }
⇒
   {     list(this, L)
       * shared(this,_)
       * list(other, L)
       * shared(other,μ)
       * held↦O
       * O⊏this
       * O⊏other
   }
}
```

Proof of `insert`

```
procedure insert(this:List, new_value:int)
  requires list(this,L) * shared(this,μ) * held↦O * O⊏this
  ensures  list(this,[new_value]++L) * shared(this,μ)
         * held↦O
{
  var n:Node;
  n:=new Node;
  n.value:=new_value;
    {∃H·
       * shared(this,μ)
       * this.head↦H
       * ((node(H,L) * this⊏H) ∨ (H=null ∧ L=[]))
       * held↦O
       * O⊏this
       * local(n)
       * n.value↦new_value
```

```
      * n.next↦null
      * n.refCount↦_
      * n.transRefCount↦_
      * n.head^{-1}↦∅
      * n.next^{-1}↦∅
  }
if(this.head≠null)
{
  acquire this.head
}
  {∃H,T·
      * shared(this,μ)
      * this.head↦H
      * ((node(H,L) * this⊏H) ∨ (H=null ∧ L=[]))
      * (H≠null ⇒held↦O∪{H})
      * (H=null ⇒held↦O)
      * O⊏this
      * local(n)
      * n.value↦new_value
      * n.next↦null
      * n.refCount↦_
      * n.transRefCount↦T
      * n.head^{-1}↦∅
      * n.next^{-1}↦∅
      * (H≠null ⇒Inv(H))
  }
n.next:=this.head;
  {∃H,T·
      * shared(this,μ)
      * this.head↦H
      * ((node(H,L) * this⊏H) ∨ (H=null ∧ L=[]))
      * (H≠null ⇒held↦O∪{H})
      * (H=null ⇒held↦O)
      * O⊏this
      * local(n)
      * n.value↦new_value
      * n.next↦H
      * n.refCount↦_
      * (H≠null ⇒n.transRefCount↦^{0.5} T)
      * (H==null ⇒n.transRefCount↦T)
      * n.head^{-1}↦∅
      * n.next^{-1}↦∅
      * (H≠null ⇒brokenRT_inv(H,_,-1,-T))
  }
this.head:=n;
  {∃H,T·
      * shared(this,μ)
      * this.head↦n
      * ((node(H,L) * this⊏H) ∨ (H=null ∧ L=[]))
```

```
      * (H≠null ⇒ held↦O∪{H})
      * (H=null ⇒ held↦O)
      * O ⊏ this
      * local(n)
      * n.value↦new_value
      * n.next↦H
      * n.refCount↦_
      * (H≠null ⇒ n.transRefCount↦^0.5 T)
      * (H==null ⇒ n.transRefCount↦T)
      * n.head^-1↦{this}
      * n.next^-1↦∅
      * (H≠null ⇒ brokenT_inv(H,_,1−T))
  }
this.head.refCount:=1;
this.head.transRefCount:=1;
  {∃H·
      * shared(this,μ)
      * this.head↦n
      * ((node(H,L) * this ⊏ H) ∨ (H=null ∧ L=[]))
      * (H≠null ⇒ held↦O∪{H})
      * (H=null ⇒ held↦O)
      * O ⊏ this
      * local(n)
      * n.value↦new_value
      * n.next↦H
      * n.refCount↦1
      * (H≠null ⇒ n.transRefCount↦^0.5 1)
      * (H==null ⇒ n.transRefCount↦1)
      * n.head^-1↦{this}
      * n.next^-1↦∅
      * (H≠null ⇒ Inv(H))
  }
if(this.head.next≠null)
{
  release this.head.next
}
  {∃H·
      * shared(this,μ)
      * this.head↦n
      * ((node(H,L) * this ⊏ H) ∨ (H=null ∧ L=[]))
      * held↦O
      * O ⊏ this
      * local(n)
      * n.value↦new_value
      * n.next↦H
      * n.refCount↦1
      * (H≠null ⇒ n.transRefCount↦^0.5 1)
      * (H==null ⇒ n.transRefCount↦1)
```

```
        * n.head^{-1} ↦ {this}
        * n.next^{-1} ↦ ∅
    }
⇒
    {∃H·
        * shared(this, μ)
        * this.head ↦ n
        * ((node(H, L) * this ⊏ H) ∨ (H=null ∧ L=[]))
        * held ↦ O
        * O ⊏ this
        * local(n)
        * n.value ↦^ϵ new_value
        * n.next ↦^ϵ H
        * Inv(n)
    }
  share head between this and head.next
    {   shared(this, μ)
        * this.head ↦ n
        * held ↦ O
        * O ⊏ this ⊏ n
        * node(n, [new_value]++L)
    }
⇒
    {   list(this, [new_value]++L)
        * shared(this, μ)
        * held ↦ O
    }
}
```

Proof of `set`

```
procedure set(this:List, index, new_value:int)
  requires ( list(this, L)
           * shared(this, μ)
           * held ↦ O
           * O ⊏ this) ∧ 0≤index<|L|
  ensures    list(this, L⟨index→value⟩)
           * shared(this, μ)
           * held ↦ O
{
  var h:Node;
    // by the precondition, the head is not null
    { ∃H·
        ( this.head ↦ H
        * node(H, L)
        * shared(this, μ)
        * held ↦ O
```

```
        * O ⊏ this ⊏ H) ∧ 0≤index<|L|
    }
  acquire this.head;
    { ∃H·
        ( this.head↦H
        * node(H,L)
        * shared(this,μ)
        * held↦O∪{H}
        * Inv(H)
        * O ⊏ this ⊏ H) ∧ 0≤index<|L|
    }
  if(this.head.refCount=1)
  {
      // see Sect.3.3 Setting without Copying
      { ∃H·
          ( this.head↦H
          * node(H,L)
          * shared(this,μ)
          * held↦O∪{H}
          * inv1(H,_)
          * O ⊏ this ⊏ H) ∧ 0≤index<|L|
      }
    node_set(this.head, index, value)
      { ∃H·
            this.head↦H
          * node(H,L⟨index→value⟩)
          * shared(this,μ)
          * held↦O
          * O ⊏ this ⊏ H
      }
⇒
      {  list(this,L⟨index→value⟩)
       * shared(this,μ)
       * held↦O
      }
  }
  else
  {
      { ∃H·
          ( this.head↦H
          * node(H,L)
          * shared(this,μ)
          * held↦O∪{H}
          * Inv(H)
          * O ⊏ this ⊏ H) ∧ 0≤index<|L|
      }
    this.head.refCount:=this.head.refCount−1;
      { ∃H·
```

```
        ( this.head↦H
        * node(H,L)
        * shared(this,μ)
        * held↦O∪{H}
        * brokenR_inv(H,_,−1)
        * O⊏this⊏H) ∧ 0≤index<|L|
    }
  h:=this.head;
    { ∃H·
        ( this.head↦H
        * node(H,L)
        * shared(this,μ)
        * held↦O∪{H}
        * brokenR_inv(H,_,−1)
        * O⊏this⊏H) ∧ 0≤index<|L| ∧ h=H
    }
  this.head:=new Node;
    { ∃H,H′·
        ( this.head↦H′
        * H′.value↦_
        * H′.next↦null
        * H′.refCount↦_
        * H′.transRefCount↦_
        * H′.head⁻¹↦{this}
        * H′.next⁻¹↦∅
        * local(H′)
        * node(H,L)
        * shared(this,μ)
        * held↦O∪{H}
        * brokenT_inv(H,_,1)
        * O⊏this⊏H) ∧ 0≤index<|L| ∧ h=H
    }
  node_set_copy(h, index, value, this.head)
    { ∃H′·
          this.head↦H′
        * node(H′,L⟨index→value⟩)
        * shared(this,μ)
        * held↦O
        * O⊏this⊏H′
    }
⇒
    {   list(this,L⟨index→value⟩)
      * shared(this,μ)
      * held↦O
    }
}
  {   list(this,L⟨index→value⟩)
    * shared(this,μ)
```

```
        * held↦O
    }
}
```

Proof of node_set

```
procedure node_set(this:Node, index, new_value:int)
  requires ( node(this,L)
           * inv1(this,_)
           * held↦O∪{this}
           * O⊏this
           ) ∧   0≤index<|L|
  ensures node(this, L⟨index→new_value⟩) * held↦O
{
  var h:Node;
  if(index=0)
  {
      {    ( node(this,L)
           * inv1(this,_)
           * held↦O∪{this}
           * O⊏this
           ) ∧  0=index<|L|
      }
    this.value:=new_value;
      {       node(this,L⟨index→new_value⟩)
           * inv1(this,_)
           * held↦O∪{this}
           * O⊏this
      }
    release this
      { node(this, L⟨index→new_value⟩) * held↦O }
  }
  else
  {
      {    ( node(this,L)
           * inv1(this,_)
           * held↦O∪{this}
           * O⊏this
           ) ∧   0<index<|L|
      }
    acquire this.next;
      { ∃N·
           ( node(this,L)
           * inv1(this,N)
           * Inv(N)
           * held↦O∪{this,N}
           * O⊏this⊏N
```

```
        ) ∧   0<index<|L|
    }
  }
  if(this.next.refCount=1)
  {
    // see Sect.3.3, Setting without Copying
    { ∃N·
      ( node(this,L)
      * inv1(this,N)
      * inv1(N)
      * held↦O∪{this,N}
      * O⊏this⊏N
      ) ∧  0<index<|L|
    }
  release this;
    { ∃N·
      ( node(this,L)
      * inv1(N)
      * held↦O∪{N}
      * O⊏this⊏N
      ) ∧  0<index<|L|
    }
⇒
    { ∃N·
      ( this.value↦ϵ L[0]
      * this.next↦ϵ N
      * node(N,L[1..])
      * inv1(N)
      * held↦O∪{N}
      * O⊏this⊏N
      ) ∧  0<index<|L|
    }
  node_set(this.next, index−1, new_value)
    { ∃N·
      ( this.value↦ϵ L[0]
      * this.next↦ϵ N
      * node(N,L[1,..]⟨index−1→new_value⟩)
      * held↦O
      ) ∧  0<index<|L|
    }
⇒
    { node(this, L⟨index→new_value⟩) * held↦O }
  }
  else
  {
    { ∃N·
      ( node(this,L)
      * inv1(this,N)
      * Inv(N)
```

$$
\begin{array}{l}
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}, N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad) \ \wedge \ \ 0 < \mathsf{index} < |L| \\
\} \\
\mathsf{this.next.refCount} := \mathsf{this.next.refCount} - 1; \\
\{ \ \exists N \cdot \\
\quad (\ \mathsf{node}(\mathsf{this}, L) \\
\quad * \ \mathsf{inv1}(\mathsf{this}, N) \\
\quad * \ \mathsf{brokenR\_inv}(N, \_, -1) \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}, N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad) \ \wedge \ \ 0 < \mathsf{index} < |L| \\
\} \\
\Rightarrow \\
\{ \ \exists N, B_1, B_2, F \cdot \\
\quad (\ \mathsf{this.value} \mapsto L[0] \\
\quad * \ \mathsf{this.next} \mapsto N \\
\quad * \ \mathsf{this.head}^{-1} \mapsto B_1 \\
\quad * \ \mathsf{this.next}^{-1} \mapsto B_2 \\
\quad * \ \mathsf{this.refCount} \mapsto 1 \\
\quad * \ \mathsf{this.transRefCount} \mapsto 1 \\
\quad * \ (\circledast n \in B_2 \cdot n.\mathsf{transRefCount} \mapsto F(\mathsf{n})) \\
\quad * \ \mathsf{node}(N, L[1..]) \\
\quad * \ \mathsf{brokenR\_inv}(N, \_, -1) \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}, N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad) \ \wedge \ \ 0 < \mathsf{index} < |L| \\
\quad \wedge \ \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\} \\
\mathsf{h} := \mathsf{this.next}; \\
\{ \ \exists N, B_1, B_2, F \cdot \\
\quad (\ \mathsf{this.value} \mapsto L[0] \\
\quad * \ \mathsf{this.next} \mapsto N \\
\quad * \ \mathsf{this.head}^{-1} \mapsto B_1 \\
\quad * \ \mathsf{this.next}^{-1} \mapsto B_2 \\
\quad * \ \mathsf{this.refCount} \mapsto 1 \\
\quad * \ \mathsf{this.transRefCount} \mapsto 1 \\
\quad * \ (\circledast n \in B_2 \cdot \mathsf{n.transRefCount} \mapsto F(n)) \\
\quad * \ \mathsf{node}(N, L[1..]) \\
\quad * \ \mathsf{brokenR\_inv}(N, \_, -1) \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}, N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad) \ \wedge \ \ 0 < \mathsf{index} < |L| \\
\quad \wedge \ \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\quad \wedge \ \ \mathsf{h} = N \\
\} \\
\mathsf{this.next} := \textbf{new Node}; \\
\{ \ \exists N, N', B_1, B_2, F \cdot \\
\quad (\ \mathsf{this.value} \mapsto L[0]
\end{array}
$$

$$
\begin{array}{l}
\quad * \ \mathsf{this.next} \mapsto N' \\
\quad * \ \mathsf{this.head}^{-1} \mapsto B_1 \\
\quad * \ \mathsf{this.next}^{-1} \mapsto B_2 \\
\quad * \ \mathsf{this.refCount} \mapsto 1 \\
\quad * \ \mathsf{this.transRefCount} \mapsto 1 \\
\quad * \ (\circledast n \in B_2 \cdot \mathsf{n.transRefCount} \mapsto F(n)) \\
\quad * \ \mathsf{node}(N, L[1..]) \\
\quad * \ \mathsf{brokenT\_inv}(N, \_, +1) \\
\quad * \ \mathsf{local}(N') \\
\quad * \ N'.\mathsf{value} \mapsto \_ \\
\quad * \ N'.\mathsf{next} \mapsto \mathbf{null} \\
\quad * \ N'.\mathsf{refCount} \mapsto \_ \\
\quad * \ N'.\mathsf{transRefCount} \mapsto \_ \\
\quad * \ N'.\mathsf{head}^{-1} \mapsto \emptyset \\
\quad * \ N'.\mathsf{next}^{-1} \mapsto \{\mathsf{this}\} \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}, N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad) \ \wedge \ 0 < \mathsf{index} < |L| \\
\qquad \wedge \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\qquad \wedge \ \mathsf{h} = N \\
\} \\
\Rightarrow \\
\{ \ \exists N, N', B_1, B_2, F \cdot \\
\quad (\ \mathsf{this.value} \mapsto L[0] \\
\quad * \ \mathsf{this.next} \mapsto N' \\
\quad * \ \mathsf{this.head}^{-1} \mapsto B_1 \\
\quad * \ \mathsf{this.next}^{-1} \mapsto B_2 \\
\quad * \ \mathsf{this.refCount} \mapsto 1 \\
\quad * \ \mathsf{this.transRefCount} \overset{0.5}{\longmapsto} 1 \\
\quad * \ (\circledast n \in B_2 \cdot \mathsf{n.transRefCount} \mapsto F(n)) \\
\quad * \ \mathsf{node}(N, L[1..]) \\
\quad * \ \mathsf{brokenT\_inv}(N, \_, 1) \\
\quad * \ \mathsf{local}(N') \\
\quad * \ N'.\mathsf{value} \mapsto \_ \\
\quad * \ N'.\mathsf{next} \mapsto \mathbf{null} \\
\quad * \ N'.\mathsf{refCount} \mapsto \_ \\
\quad * \ N'.\mathsf{transRefCount} \mapsto \_ \\
\quad * \ N'.\mathsf{head}^{-1} \mapsto \emptyset \\
\quad * \ N'.\mathsf{next}^{-1} \mapsto \{\mathsf{this}\} \\
\quad * \ (\circledast n \in \{\mathsf{this}\} \cdot n.\mathsf{transRefCount} \overset{0.5}{\longmapsto} 1) \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}, N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad) \ \wedge \ 0 < \mathsf{index} < |L| \\
\qquad \wedge \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\qquad \wedge \ \mathsf{h} = N \\
\} \\
\mathtt{node\_set\_copy(h,\ index-1,\ new\_value,\ this.next);} \\
\{ \ \exists N, N', B_1, B_2, F \cdot \\
\quad (\ \mathsf{this.value} \mapsto L[0]
\end{array}
$$

```
            * this.next↦N'
            * this.head^-1↦B1
            * this.next^-1↦B2
            * this.refCount↦1
            * this.transRefCount↦^0.5 1
            * (⊛n∈B2·n.transRefCount↦F(n))
            * node(N',L[1..]⟨index−1→new_value⟩)
            * held↦O∪{this}
            * O⊏this⊏N⊏N'
            ) ∧ 0<index<|L|
              ∧ 1 = |B1| + (∑n∈B2·F(n)) = |B1|+|B2|
              ∧ h=N
        }
⇒
        {   ( node(this, L⟨index→new_value⟩)
            * inv1(this,_)
            * held↦O∪{this}
            ) ∧ this∉O
        }
      release this
        { node(this, L⟨index→new_value⟩) * held↦O }
    }
      { node(this, L⟨index→new_value⟩) * held↦O }
  }
}
```

Proof of node_set_copy

```
procedure node_set_copy
            (this:Node, index, new_value:int, new_node:Node)
  requires ( node(this, L)
           * new_node.value↦_ * new_node.next↦null
           * new_node.refCount↦_
           * new_node.transRefCount↦_
           * new_node.head^-1↦B1
           * new_node.next^-1↦B2
           * local(new_node)
           * (⊛n∈B2·n.transRefCount↦^0.5 F(n))
           * brokenT_inv(this,N,1)
           * held↦O∪{this}
           * O⊏this
           ) ∧ 1 = |B1| + (∑n∈B2·F(n)) = |B1|+|B2|
             ∧ 0≤index<|L|
  ensures ( node(new_node,L⟨index→new_value⟩)
          * held↦O
          * this⊏new_node
          )
```

```
{
  new_node.refCount:=1;
  new_node.transRefCount:=1;
    {   ( this.value ↦^ε L[0]
        * this.next ↦^ε N
        * ((node(N,L[1..]) * this ⊏ N) ∨ (N=null ∧ |L|=1))
        * new_node.value ↦ _
        * new_node.next ↦ null
        * new_node.refCount ↦ 1
        * new_node.transRefCount ↦ 1
        * new_node.head^-1 ↦ B1
        * new_node.next^-1 ↦ B2
        * local(new_node)
        * (⊛ n ∈ B2 · n.transRefCount ↦^0.5 F(n))
        * brokenT_inv(this,N,1)
        * held ↦ O∪{this}
        * O ⊏ this
        ) ∧ 1 = |B1| + (∑ n ∈ B2 · F(n)) = |B1|+|B2|
          ∧ 0≤index<|L|
    }
  if(this.next≠null)
  {
    acquire this.next
  }
    {   ( this.value ↦^ε L[0]
        * this.next ↦^ε N
        * ((node(N,L[1..]) * this ⊏ N) ∨ (N=null ∧ |L|=1))
        * new_node.value ↦ _
        * new_node.next ↦ null
        * new_node.refCount ↦ 1
        * new_node.transRefCount ↦ 1
        * new_node.head^-1 ↦ B1
        * new_node.next^-1 ↦ B2
        * local(new_node)
        * (⊛ n ∈ B2 · n.transRefCount ↦^0.5 F(n))
        * brokenT_inv(this,N,1)
        * (N≠null ⇒ Inv(N))
        * (N≠null ⇒ held ↦ O∪{this}∪{N})
        * (N=null ⇒ held ↦ O∪{this})
        * O ⊏ this
        ) ∧ 1 = |B1| + (∑ n ∈ B2 · F(n)) = |B1|+|B2|
          ∧ 0≤index<|L|
    }
  this.transRefCount:=this.transRefCount−1;
    // note that the epsilon permissions to this.value
    // and this.next return to the invariant of this
    // see Sect.3.3, Losing Interest
    {   (((node(N,L[1..]) * this ⊏ N) ∨ (N=null ∧ |L|=1))
```

```
      * new_node.value↦_
      * new_node.next↦null
      * new_node.refCount↦1
      * new_node.transRefCount↦1
      * new_node.head^{-1}↦B1
      * new_node.next^{-1}↦B2
      * local(new_node)
      * (⊛n∈B2·n.transRefCount↦^{0.5} F(n))
      * inv(this,N)
      * (N≠null ⇒brokenT_inv(N,_,1))
      * (N≠null ⇒held↦O∪{this}∪{N})
      * (N=null ⇒held↦O∪{this})
      * O⊏this
      ) ∧ 1 = |B1| + (∑n∈B2·F(n)) = |B1|+|B2|
        ∧ 0≤index<|L|
  }
if(index=0)
{
    {  (((node(N,L[1..]) * this⊏N) ∨ (N=null ∧ |L|=1))
        * new_node.value↦_
        * new_node.next↦null
        * new_node.refCount↦1
        * new_node.transRefCount↦1
        * new_node.head^{-1}↦B1
        * new_node.next^{-1}↦B2
        * local(new_node)
        * (⊛n∈B2·n.transRefCount↦^{0.5} F(n))
        * inv(this,N)
        * (N≠null ⇒brokenT_inv(N,_,1))
        * (N≠null ⇒held↦O∪{this}∪{N})
        * (N=null ⇒held↦O∪{this})
        * O⊏this
        ) ∧ 1 = |B1| + (∑n∈B2·F(n)) = |B1|+|B2|
          ∧ 0=index<|L|
    }
  new_node.value:=new_value;
  new_node.next:=this.next;
    {  (((node(N,L[1..]) * this⊏N) ∨ (N=null ∧ |L|=1))
        * new_node.value↦new_value
        * new_node.next↦N
        * new_node.refCount↦1
        * new_node.transRefCount↦^{0.5}1
        * (N≠null ⇒new_node.transRefCount↦^{0.5}1)
        * new_node.head^{-1}↦B1
        * new_node.next^{-1}↦B2
        * local(new_node)
        * (⊛n∈B2·n.transRefCount↦^{0.5} F(n))
        * inv(this,N)
```

```
          * (N≠null ⇒brokenR_inv(N,_,−1))
          * (N≠null ⇒held↦O∪{this}∪{N})
          * (N=null ⇒held↦O∪{this})
          * O⊏this
          ) ∧ 1 = |B1| + (∑n∈B2·F(n)) = |B1|+|B2|
            ∧ 0=index<|L|
      }
    if(this.next≠null)
    {
      this.next.refCount:=this.next.refCount+1
      release this.next
    }
      { (((node(N,L[1..]) * this⊏N) ∨ (N=null ∧ |L|=1))
          * new_node.value↦new_value
          * new_node.next↦N
          * new_node.refCount↦1
          * new_node.transRefCount↦^0.5 1
          * new_node.head^−1↦B1
          * new_node.next^−1↦B2
          * local(new_node)
          * (⊛n∈B2·n.transRefCount↦^0.5 F(n))
          * held↦O∪{this}
          * O⊏this
          ) ∧ 1 = |B1| + (∑n∈B2·F(n)) = |B1|+|B2|
            ∧ 0=index<|L|
      }
⇒
     { ∃N'·
         ( new_node.value↦^ϵ L⟨index→new_value⟩[0])
         * new_node.next↦^ϵ N'
         * local(new_node)
         * ((N'≠null ∧ node(N',L⟨index→new_value⟩[1..]))
              ∨ (N'=null ∧ |L|=1))
         * Inv(this) * Inv(new_node)
         * held↦O∪{this}
         )
     }
  }
  else
  {
      { (((node(N,L[1..]) * this⊏N) ∨ (N=null ∧ |L|=1))
          * new_node.value↦_
          * new_node.next↦null
          * new_node.refCount↦1
          * new_node.transRefCount↦1
          * new_node.head^−1↦B1
          * new_node.next^−1↦B2
          * local(new_node)
```

```
        * (⊛n ∈ B2 · n.transRefCount ↦0.5 F(n))
        * inv(this, N)
        * (N≠null ⇒ brokenT_inv(N, _, 1))
        * (N≠null ⇒ held ↦ O∪{this}∪{N})
        * (N=null ⇒ held ↦ O∪{this})
        * O ⊏ this
        ) ∧ 1 = |B1| + (∑n ∈ B2 · F(n)) = |B1|+|B2|
          ∧ 0<index<|L|
  }
new_node.value:=this.value;
  { (((node(N, L[1..]) * this ⊏ N) ∨ (N=null ∧ |L|=1))
        * new_node.value ↦ L[0]
        * new_node.next ↦ null
        * new_node.refCount ↦ 1
        * new_node.transRefCount ↦ 1
        * new_node.head^-1 ↦ B1
        * new_node.next^-1 ↦ B2
        * local(new_node)
        * (⊛n ∈ B2 · n.transRefCount ↦0.5 F(n))
        * inv(this, N)
        * (N≠null ⇒ brokenT_inv(N, _, 1))
        * (N≠null ⇒ held ↦ O∪{this}∪{N})
        * (N=null ⇒ held ↦ O∪{this})
        * O ⊏ this
        ) ∧ 1 = |B1| + (∑n ∈ B2 · F(n)) = |B1|+|B2|
          ∧ 0<index<|L|
  }
if(this.next≠null)
{
    {   ( node(N, L[1..])
        * new_node.value ↦ L[0]
        * new_node.next ↦ null
        * new_node.refCount ↦ 1
        * new_node.transRefCount ↦ 1
        * new_node.head^-1 ↦ B1
        * new_node.next^-1 ↦ B2
        * local(new_node)
        * (⊛n ∈ B2 · n.transRefCount ↦0.5 F(n))
        * Inv(this)
        * shared(N, _)
        * brokenT_inv(N, _, 1)
        * held ↦ O∪{this}∪{N}
        * O ⊏ this ⊏ N
        ) ∧ 1 = |B1| + (∑n ∈ B2 · F(n)) = |B1|+|B2|
          ∧ 0<index<|L|
    }
  new_node.next:=new Node;
    { ∃N' ·
```

$$
\begin{array}{l}
\quad (\ \mathsf{node}(N, L[1..]) \\
\quad * \ \mathsf{new\_node.value} \mapsto L[0] \\
\quad * \ \mathsf{new\_node.next} \mapsto N' \\
\quad * \ \mathsf{local}(N') \\
\quad * \ N'.\mathsf{value} \mapsto \_ \\
\quad * \ N'.\mathsf{next} \mapsto \mathbf{null} \\
\quad * \ N'.\mathsf{refCount} \mapsto \_ \\
\quad * \ N'.\mathsf{transRefCount} \mapsto \_ \\
\quad * \ N'.\mathsf{head}^{-1} \mapsto \emptyset \\
\quad * \ N'.\mathsf{next}^{-1} \mapsto \{\mathsf{new\_node}\} \\
\quad * \ \mathsf{new\_node.refCount} \mapsto 1 \\
\quad * \ \mathsf{new\_node.transRefCount} \mapsto 1 \\
\quad * \ \mathsf{new\_node.head}^{-1} \mapsto B_1 \\
\quad * \ \mathsf{new\_node.next}^{-1} \mapsto B_2 \\
\quad * \ \mathsf{local}(\mathsf{new\_node}) \\
\quad * \ (\circledast n \in B_2 \cdot n.\mathsf{transRefCount} \overset{0.5}{\longmapsto} F(n)) \\
\quad * \ \mathsf{inv}(\mathsf{this}, N) \\
\quad * \ \mathsf{shared}(N, \_) \\
\quad * \ \mathsf{brokenT\_inv}(N, \_, 1) \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}\} \cup \{N\} \\
\quad * \ O \sqsubset \mathsf{this} \sqsubset N \\
\quad)\ \wedge\ \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\qquad \wedge\ \ 0 < \mathsf{index} < |L| \\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{node\_set\_copy} \\
\quad (\mathsf{this.next},\ \mathsf{index}-1,\ \mathsf{new\_value},\ \mathsf{new\_node.next}) \\
\{\ \exists N' \cdot \\
\quad (\ \mathsf{new\_node.value} \mapsto L[0] \\
\quad * \ \mathsf{new\_node.next} \mapsto N' \\
\quad * \ \mathsf{new\_node.refCount} \mapsto 1 \\
\quad * \ \mathsf{new\_node.transRefCount} \overset{0.5}{\longmapsto} 1 \\
\quad * \ \mathsf{new\_node.head}^{-1} \mapsto B_1 \\
\quad * \ \mathsf{new\_node.next}^{-1} \mapsto B_2 \\
\quad * \ \mathsf{local}(\mathsf{new\_node}) \\
\quad * \ (\circledast n \in B_2 \cdot n.\mathsf{transRefCount} \overset{0.5}{\longmapsto} F(n)) \\
\quad * \ Inv(\mathsf{this}) \\
\quad * \ \mathsf{node}(N', L[1..]\langle \mathsf{index}-1 \rightarrow \mathsf{new\_value} \rangle) \\
\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}\} \\
\quad)\ \wedge\ \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\qquad \wedge\ \ 0 < \mathsf{index} < |L| \\
\}
\end{array}
$$

$$
\Rightarrow
$$

$$
\begin{array}{l}
\{\ \exists N' \cdot \\
\quad (\ \mathsf{new\_node.value} \overset{\epsilon}{\mapsto} L\langle \mathsf{index} \rightarrow \mathsf{new\_value} \rangle[0]) \\
\quad * \ \mathsf{new\_node.next} \overset{\epsilon}{\mapsto} N' \\
\quad * \ \mathsf{local}(\mathsf{new\_node}) \\
\quad * \ ((N' \neq \mathbf{null} \ \wedge\ \mathsf{node}(N', L\langle \mathsf{index} \rightarrow \mathsf{new\_value} \rangle[1..])) \\
\qquad \vee\ \ (N' = \mathbf{null} \ \wedge\ |\mathsf{L}| = 1))
\end{array}
$$

$$
\begin{array}{l}
\quad\quad * \ Inv(\mathsf{this}) \ * \ Inv(\mathsf{new\_node}) \\
\quad\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}\} \\
\quad\quad) \\
\quad \} \\
\} \\
\textbf{else} \\
\{ \\
\quad \{ \quad (\ \mathsf{new\_node.value} \mapsto L[0] \\
\quad\quad * \ \mathsf{new\_node.next} \mapsto \textbf{null} \\
\quad\quad * \ \mathsf{new\_node.refCount} \mapsto 1 \\
\quad\quad * \ \mathsf{new\_node.transRefCount} \mapsto 1 \\
\quad\quad * \ \mathsf{new\_node.head}^{-1} \mapsto B_1 \\
\quad\quad * \ \mathsf{new\_node.next}^{-1} \mapsto B_2 \\
\quad\quad * \ \mathsf{local}(\mathsf{new\_node}) \\
\quad\quad * \ (\circledast n \in B_2 \cdot n.\mathsf{transRefCount} \xmapsto{0.5} F(n)) \\
\quad\quad * \ \mathsf{inv}(\mathsf{this}, \textbf{null}) \\
\quad\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}\} \\
\quad\quad * \ O \sqsubset \mathsf{this} \\
\quad\quad) \ \wedge \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\quad\quad \wedge \ 0 < \mathsf{index} < |L| = 1 \ \wedge \ N = \textbf{null} \\
\quad \} \\
\quad \mathsf{new\_node.next} := \textbf{null} \\
\quad \{ \quad (\ \mathsf{new\_node.value} \mapsto L[0] \\
\quad\quad * \ \mathsf{new\_node.next} \mapsto \textbf{null} \\
\quad\quad * \ \mathsf{new\_node.refCount} \mapsto 1 \\
\quad\quad * \ \mathsf{new\_node.transRefCount} \mapsto 1 \\
\quad\quad * \ \mathsf{new\_node.head}^{-1} \mapsto B_1 \\
\quad\quad * \ \mathsf{new\_node.next}^{-1} \mapsto B_2 \\
\quad\quad * \ \mathsf{local}(\mathsf{new\_node}) \\
\quad\quad * \ (\circledast n \in B_2 \cdot n.\mathsf{transRefCount} \xmapsto{0.5} F(n)) \\
\quad\quad * \ \mathsf{inv}(\mathsf{this}, \textbf{null}) \\
\quad\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}\} \\
\quad\quad * \ O \sqsubset \mathsf{this} \\
\quad\quad) \ \wedge \ 1 = |B_1| + (\sum n \in B_2 \cdot F(n)) = |B_1| + |B_2| \\
\quad\quad \wedge \ 0 < \mathsf{index} < |L| = 1 \ \wedge \ N = \textbf{null} \\
\quad \} \\
\Rightarrow \\
\quad \{ \ \exists N' \cdot \\
\quad\quad (\ \mathsf{new\_node.value} \xmapsto{\epsilon} L\langle \mathsf{index} \rightarrow \mathsf{new\_value} \rangle [0]) \\
\quad\quad * \ \mathsf{new\_node.next} \xmapsto{\epsilon} N' \\
\quad\quad * \ \mathsf{local}(\mathsf{new\_node}) \\
\quad\quad * \ ((N' \neq \textbf{null} \ \wedge \ \mathsf{node}(N', L\langle \mathsf{index} \rightarrow \mathsf{new\_value} \rangle [1..])) \\
\quad\quad\quad\quad \vee \ (N' = \textbf{null} \ \wedge \ |\mathsf{L}| = 1)) \\
\quad\quad * \ Inv(\mathsf{this}) \ * \ Inv(\mathsf{new\_node}) \\
\quad\quad * \ \mathsf{held} \mapsto O \cup \{\mathsf{this}\} \\
\quad\quad) \\
\quad \} \\
\}
\end{array}
$$

```
  }
    { ∃N′·
        ( new_node.value↦ϵ L⟨index→new_value⟩[0])
        * new_node.next↦ϵ N′
        * local(new_node)
        * ((N′≠null ∧ node(N′,L⟨index→new_value⟩[1..]))
            ∨ (N′=null ∧ |L|=1))
        * Inv(this) * Inv(new_node)
        * held↦O∪{this}
        )
    }
  share new_node between this and new_node.next;
    { ∃N′·
        ( new_node.value↦ϵ L⟨index→new_value⟩[0])
        * new_node.next↦ϵ N′
        * shared(new_node)
        * ((node(N′,L⟨index→new_value⟩[1..]) * new_node⊏N′)
            ∨ (N′=null ∧ |L|=1))
        * Inv(this)
        * held↦O∪{this}
        * this⊏new_node
        )
    }
⇒
    {   ( node(new_node,L⟨index→new_value⟩)
        * Inv(this)
        * held↦O∪{this}
        * this⊏new_node
        )
    }
  release this
    {   ( node(new_node,L⟨index→new_value⟩)
        * held↦O
        * this⊏new_node
        )
    }
}
```

Proof of addOneToTransRefCount

```
ghost procedure addOneToTransRefCount(this:Node)
  requires   node(this,L)
           * held↦O∪{this}
           * O⊏this
           * brokenT_inv(this,_,−1)
  ensures    node(this,L)
           * node(this,L)
```

```
                * held↦O∪{this}
                * O⊏this
                * Inv(this)
{
    { ∃N·
            this.value↦^ε L[0]
        * this.next↦^ε N
        * ((node(N,L[1..]) * this⊏N) ∨ (N=null ∧ |L|=1))
        * held↦O∪{this}
        * O⊏this
        * brokenT_inv(this,_,−1)
    }
  if(this.next≠null)
  {
    acquire this.next
  }
    { ∃N·
            this.value↦^ε L[0]
        * this.next↦^ε N
        * ((node(N,L[1..]) * this⊏N * Inv(N))
              ∨ (N=null ∧ |L|=1))
        * (N=null ⇒ held↦O∪{this})
        * (N≠null ⇒ held↦O∪{this,N})
        * O⊏this
        * brokenT_inv(this,_,−1)
    }
  this.transRefCount := this.transRefCount + 1;
    { ∃N·
            this.value↦^{2ε} L[0]
        * this.next↦^{2ε} N
        * ((node(N,L[1..]) * this⊏N * brokenT_inv(N))
              ∨ (N=null ∧ |L|=1))
        * (N=null ⇒ held↦O∪{this})
        * (N≠null ⇒ held↦O∪{this,N})
        * O⊏this
        * Inv(this)
    }
  if(this.next≠null)
  {
      { ∃N·
              this.value↦^{2ε} L[0]
          * this.next↦^{2ε} N
          * node(N,L[1..])
          * brokenT_inv(N)
          * held↦O∪{this,N}
          * O⊏this⊏N
          * Inv(this)
```

```
      }
    addOneToTransRefCount(this.next);
      { ∃N·
            this.value ↦^{2ε} L[0]
          * this.next ↦^{2ε} N
          * node(N, L[1..])
          * node(N, L[1..])
          * Inv(N)
          * held ↦ O ∪ {this, N}
          * O ⊏ this ⊏ N
          * Inv(this)
      }
    release this.next
      { ∃N·
            this.value ↦^{2ε} L[0]
          * this.next ↦^{2ε} N
          * node(N, L[1..])
          * node(N, L[1..])
          * held ↦ O ∪ {this}
          * O ⊏ this ⊏ N
          * Inv(this)
      }
⇒
      {     node(this, L)
          * node(this, L)
          * held ↦ O ∪ {this}
          * O ⊏ this
          * Inv(this)
      }
  }
    {     node(this, L)
        * node(this, L)
        * held ↦ O ∪ {this}
        * O ⊏ this
        * Inv(this)
    }
}
```