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 solution.

1 Introduction

The advent of *separation logic* [1] 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

⋆ The first author was funded by the Hasler Foundation.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 149–168, 2013.

that not only expresses the fact that 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 *self-framing* [2–4]. A self-framing 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, parallelly, through method call, or through thread forking) with other programs. In concurrent variants of separation logic, permissions can be split [5] (e.g., in fractions [6]), thus enabling shared resources without data races. These well-known extensions of separation logic, maintain this important property: all expressible state conditions are self-framing.

Self-framing 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 Inheritance Protocol* [7]. The priority of a node is the minimum of its initial priority and the priority of the node pointing to it (see also [8]).
- *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 non-self-framing, 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 self-framing 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.

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.

Our online technical report [9] contains an appendix with the full specification, implementation and correctness proof 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 from Chalice [10]. We then extend our language with the backpointer formalism (Sect. 2.2) and provide an argument about the soundness of this extension (Sect. 2.3).

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.

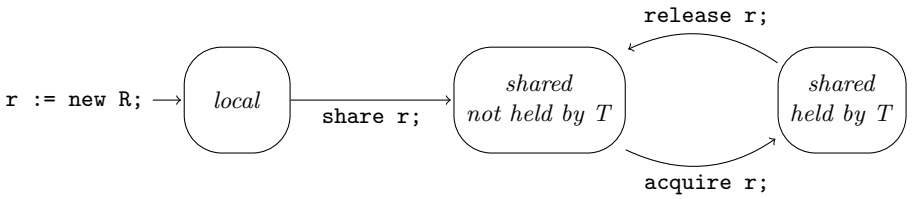

Fig. 1. A record's life cycle from the point of view of thread T

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 [10]. 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*. A thread is allowed to acquire a record, only when that record is greater in $\sqsubset$ than all the other records that the thread holds.

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

$$\texttt{shared}(r,\_) \lor \texttt{local}(r) \quad \Rightarrow \quad r \neq \mathbf{null}$$

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

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

  This means that, unlike in Chalice, the lock-level of an object is immutable.
- 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:

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

  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:

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

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

The **share** command can specify bounds for the lock-level of the record being shared. 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 [11].

```
{emp}
    r:=new R
{newRec(r)}

{local(r) * Inv(r) * held↦O ∧ r∉O}
    share r
{shared(r,_) * held↦O}

{shared(r,μ) * held↦O * O⊏r}
    acquire r
{shared(r,μ) * held↦O∪{r} * Inv(r)}

{shared(r,μ) * held↦O * Inv(r) ∧ r∈O}
    release r
{shared(r,μ) * held↦O−{r}}
```

Fig. 2. Commands on records

2.2 Backpointers

To make the backpointer properties self-framing, we impose a restriction on the assignments which may potentially invalidate such properties.

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.

```
struct C { tracked f:D; }
```

Backpointer Definitional Axiom. Suppose now that a record type C has a tracked field f of type D (where C, D are not necessarily different). To express backpointer properties, it should be possible to refer to "all allocated records of type C that point to the record d of type D through the field f". We write $d.(C.f)^{-1}$ to refer to that set. In other words, the *definitional axiom* of backpointers is (for every state σ):

$$[\![\forall c \in \alpha C, d \in \alpha D \cdot \; c \in d.(C.f)^{-1} \quad \Leftrightarrow \quad c.f = d]\!]\,(\sigma) \tag{1}$$

where

- $[\![E]\!]\,(\sigma)$ evaluates expression E in state σ
- αT is the set of all non-null allocated records of type T in a given state

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

Backpointer Fields. The value of the expression $(C.f)^{-1}$ is not associated with any permission, which is what makes it non-self-framing. To fix this, we turn $(C.f)^{-1}$ into *a field* of D. 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 explicit "ghost assignments" to it are forbidden[1].

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

$$r.f := p$$

Notice that this assignment changes not only the value of $r.f$, but also that of $q.f^{-1}$ and $p.f^{-1}$. The situation is depicted graphically in Fig. 3. 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 q or p are the **null** reference, then, of course, we do not require access to their backpointer fields.

We introduce two axiomatic rules for tracked assignments[2]. First, for the case $p \neq q$:

$$\begin{array}{l}
\{ \\
\quad r{=}R{\neq}\mathbf{null} \ \wedge \ p{=}P{\neq}Q \ \wedge \ r.f \mapsto Q \\
\quad * \ (p{\neq}\mathbf{null} \ \Rightarrow \ p.f^{-1} \mapsto S_1) \ * \ (Q{\neq}\mathbf{null} \ \Rightarrow \ Q.f^{-1} \mapsto S_2) \\
\} \\
\quad r.f := p \\
\{ \\
\quad r{=}R{\neq}\mathbf{null} \ \wedge \ p{=}P{\neq}Q \ \wedge \ r.f \mapsto p \\
\quad * \ (p{\neq}\mathbf{null} \ \Rightarrow \ p.f^{-1} \mapsto S_1 - \{r\}) \ * \ (Q{\neq}\mathbf{null} \ \Rightarrow \ Q.f^{-1} \mapsto S_2 \cup \{r\}) \\
\}
\end{array}$$

and second, for the contrived case $p = q$

$$\begin{array}{l}
\{ \ r{=}R{\neq}\mathbf{null} \ \wedge \ p{=}P \ \wedge \ r.f \mapsto P \ * \ (P{\neq}\mathbf{null} \ \Rightarrow \ P.f^{-1} \mapsto S) \ \} \\
\quad r.f := p \\
\{ \ r{=}R{\neq}\mathbf{null} \ \wedge \ p{=}P \ \wedge \ r.f \mapsto P \ * \ (P{\neq}\mathbf{null} \ \Rightarrow \ P.f^{-1} \mapsto S) \ \}
\end{array}$$

Example 1. In this simple example, we will show how the backpointers discipline makes it possible to express reference counting, and how we can use reference counting to protect shared data from mutation.

Suppose that we have two types `Cell` and `Client`. Clients have a reference field `f` to cells. Many clients may share a cell and we are interested in keeping track of them. Therefore `f` is a tracked field:

```
struct Client { tracked f:Cell; }
```

[1] In this sense, backpointer fields are like JML's model fields [12]. Unlike model fields however, backpointer fields are associated with permissions.

[2] For simplicity, assume that r and p are local variables.

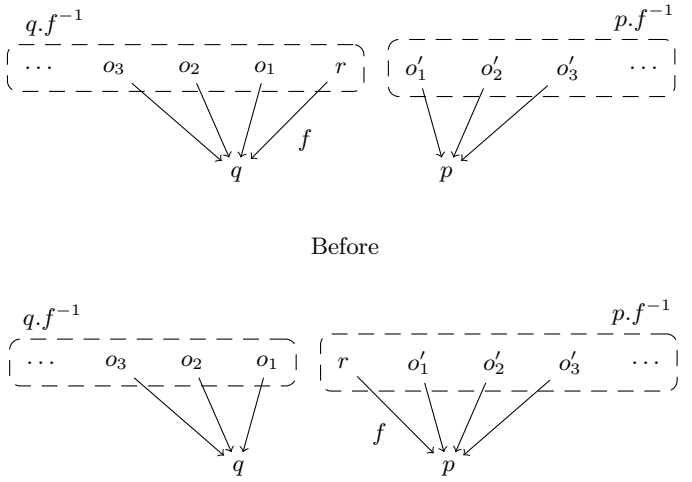

Fig. 3. Assignment to a tracked field $r.f := p$. This diagram depicts the case where p is not equal to the original value q of $r.f$ and where both p and q are non-null.

A cell has an integer field `data` and a reference counter `refCount`. If n clients point to the cell, then each of them holds ϵ permission and $1 - n\epsilon$ remains in the monitor invariant of the cell. The reference counter must be equal to n. Using the ghost field $\mathtt{f}^{-1}$, the requirement is stably expressible:

```
struct Cell
{
  data, refCount: int;
  invariant ∃B · f⁻¹ ↦ B * refCount ↦ |B| * data ⟼[1−|B|ε] _
}
```

It is impossible for a client to add/remove a reference to a cell c without first acquiring it (because one needs write access to $c.\mathtt{f}^{-1}$ to perform such an assignment). After acquiring c, if the client wishes to release c, it must also update the reference counter appropriately, since otherwise the monitor invariant of c will not hold. Here is an example of a client which correctly adds a reference to a cell:

```
acquire c;
cl:=new Client;
cl.f:=c;
c.refCount := c.refCount+1;
release c;
```

Every client that references c holds an ϵ permission to $c.\texttt{data}$. For example, in the above code, the client has gained an ϵ permission to `c.data`, because it added a new reference to `c`.

A holder of an ϵ permission to $c.\texttt{data}$ can probe the reference counter of c, to see if it shares the cell with any other client. If the reference counter is 1, then the holder may acquire the cell, combine its ϵ permission with the $1-\epsilon$ permission to $c.\texttt{data}$, and obtain write permission to $c.\texttt{data}$. Here is a client that does this correctly:

```
// here: c.data ↦^ε _
acquire c;
if(c.refCounter=1)
{
  // here we can prove c.data↦_
  c.data:=42;
}
```

So long as the reference counter is greater than 1, it is not possible for a client to gain write access to the data. □

2.3 Soundness

In this subsection, we give an brief informal argument to explain why the backpointers discipline is sound.

The extension of a specification and programming language with backpointers imposes the soundness requirement that the definitional axiom of backpointers (1) is a *system invariant*, i.e., a property that holds at any given state during the execution of the program.

Consider a programming language that supports all the features that we have introduced so far: mutable records, locking, assignment, conditionals, procedures, sequential and parallel composition. Assume a standard small step semantics for that programming language. The rule for field assignment in this language is

$$[\![e_1 \neq \textbf{null}]\!](\sigma) \quad \Rightarrow \quad \langle e_1.f := e_2,\ \sigma\rangle \rightsquigarrow \sigma[\ ([\![e_1]\!](\sigma).f) \twoheadrightarrow [\![e_2]\!](\sigma)\ ] \qquad (2)$$

where $\langle s, \sigma\rangle$ is a configuration, $\rightsquigarrow$ is the operational semantics relation and $[\cdot \twoheadrightarrow \cdot]$ is the update notation.

The introduction of backpointers entails the following change to the operational semantic rules:

- Rule (2) applies only when f is a non-tracked field
- Backpointers are introduced as ghost fields. Explicit assignments to them are forbidden.
- If f is a tracked field, then (2) is replaced by the following rule

$$[\![e_1 \neq \textbf{null}]\!](\sigma) \quad \Rightarrow \quad \langle e_1.f := e_2,\ \sigma\rangle \rightsquigarrow \sigma'[\ o.f \twoheadrightarrow [\![e_2]\!](\sigma)\ ] \qquad (3)$$

where

$$o = [\![e_1]\!](\sigma)$$

$$\sigma' = \begin{cases} \sigma''[\ o.f^{-1} \twoheadrightarrow [\![e_1.f.f^{-1}]\!](\sigma) - \{o\}\ ] & if\ [\![e_1.f \neq \texttt{null}]\!](\sigma) \\ \sigma'' & otherwise \end{cases}$$

$$\sigma'' = \begin{cases} \sigma[\ [\![e_2]\!](\sigma).f^{-1} \twoheadrightarrow [\![e_2.f^{-1}]\!](\sigma) \cup \{o\}\ ] & if\ [\![e_2 \neq \texttt{null}]\!](\sigma) \\ \sigma & otherwise \end{cases}$$

– The rule for the creation of new records is revised as follows:
  - The new record can only be assigned to a local variable[3]
  - All reference-typed fields of the new record are initialized to **null** and all backpointer fields of the new record are initialized to $\emptyset$

To prove that (1) is a system invariant, we perform a standard induction on the structure of the statements of the language. Notice that (1) can only be falsified by rule (3) and by the creation of new records.

It is easy to see that (3) does not falsify (1). For the creation of new records, we also assume that there are no dangling pointers, as is the case with languages that support garbage collection. Under this assumption, the creation of new records as described above does not falsify (1).

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. As we commented above, the specifications, implementations, and proof outlines for all the procedures can be found in [9].

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 at 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. 4. In it, $\texttt{list}(l, L)$ 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 $L[i \twoheadrightarrow v]$ denotes the sequence L with the content of index i updated to value v. Indexes are zero-based.

[3] Assignment to a field is considered syntactic sugar.

```
{newRec(this) * held↦O}
  initEmpty(this)
{list(this, []) * held↦O * O⊏this}

{newRec(this) * list(other, L) * held↦O * O⊏other}
  copy(this,other)
{  list(this, L) * list(other, L) * held↦O
 * O⊏this * O⊏other}

{list(this, L) * held↦O * O⊏this}
  insert(this,newValue)
{list(this, [newValue]++L) * held↦O}

{list(this, L) * held↦O * O⊏this ∧ 0≤index<|L| }
  set(this,index,value)
{list(this, L[index↠value]) * held↦O}
```

Fig. 4. Public Specification of CCoWLs

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. 4 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 such a linked list to represent that `list1` contains the sequence [1,2,3] (Fig. 5a). After that, a new list `list2` is created and it is initialized by copying `list1`. The client may pretend 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. 5b. 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. 5c.

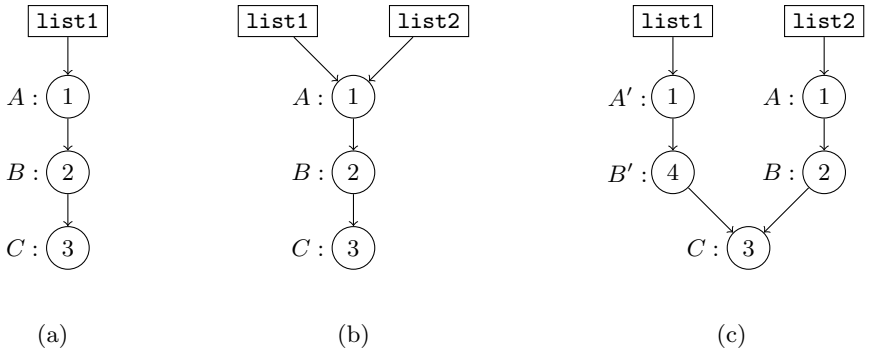

Fig. 5. An example of CCoWL history

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. 5c, 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.

```
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↦^ϵ L[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 to by field `next`, if one exists. The lock-order of node n is below that of n.`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.`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 `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 Ex. 1. 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. 6, 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 `Node`. This field counts how many lists are interested in the current node. We call it `transRefCount` (for *transitive reference counter*). In Fig. 6, we see not only the reference counters but also the transitive reference counters of all the nodes.

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:

```
value ↦^{1−Tϵ} V * next ↦^{1−Tϵ} N
```

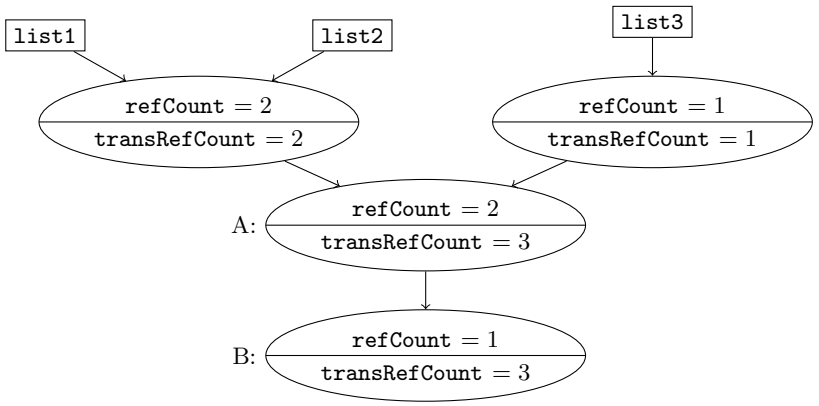

Fig. 6. Reference and Transitive Reference Counters in a CCoWL

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

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

– 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:

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

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:

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

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:

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

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=**null** $\Rightarrow$ 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| + ∑ 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[4] addOneToTransRefCount, which traverses the whole list and adds 1 to all transitive 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[index↠value]). The predicate node(this, L) of the precondition 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[index↠value]) predicate. In this way, no permission to fields value and next is ever lost.

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

For example, consider the situation in Fig. 5b. 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. 5c.

List `list1` lost interest in 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 transferred 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. 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 is translated into:

$$\texttt{this.refCount} \mapsto 1 \;*\; \texttt{this.transRefCount} \overset{0.5}{\longmapsto} 1$$
$$* \;\texttt{this.next} \mapsto N \;*\; N.\texttt{refCount} \mapsto 1 \;\wedge\; N \neq \textbf{null}$$

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

4.1 Related 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 the backpointer definitional axiom (1) holds in any state σ. 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 [13]. Various forms of ownership [14, 15] are popular invariant disciplines.

Parkinson [16] comments that object invariants are inflexible, in comparison to the use of abstract predicates. Summers et al. [8] 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 [7]) as one in which object invariants can turn a seemingly global property (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 [8], in the concurrent case.

Our verification of CCoWLs is influenced by *considerate reasoning* [17], 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 [18] 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 [19]. The work presented here cannot substitute for 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 [20] 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 [21], 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 [22] 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").

4.2 Evaluation and Work in Progress

Two significant questions that have not been answered so far are (a) how expressive is the new specification language and (b) how automatable it is.

Expressiveness. In the Introduction, we have mentioned three examples, in which backpointers seem useful. From these examples, we have focused on reference counting, which we have used in a very complex example, CCoWLs, which we have specified, implemented and verified.

It is worth mentioning that our CCoWL example is a fully-persistent data structure [22]. It is a further research direction to investigate how much our proof technique generalizes to fully-persistent data structures in general.

Besides reference counting and the CCoWLs, we have also specified, implemented, and verified the priority inheritance protocol. We are currently working on specifying union-find structures; a challenging problem for which backpointers seem to be particularly promising.

We believe that the potential of the methodology has not yet been fully explored and we expect new interesting case studies to be revealed as experience accumulates.

Automation. We have implemented a prototype verifier for backpointers as an extension of Chalice. We have tested it on a suite of 20 unit tests, observing significant variation in verification times, which is undesirable.

To counter the problem we have experimented with various degrees of restricting the automation. For example, we have given the programmer the possibility to control the triggering of backpointer and set theoretic axioms. We have also introduced explicit annotations for the application of the frame rule, for the framing of aggregate expressions.

The automation of the CCoWL case study has been extremely challenging. At the time of this writing, our tool has verified all but one of the procedures of the present example. The verification of most procedures happens within less than 5 minutes, which is satisfactory. The procedure `node_set_copy` verifies in 90 minutes. The verification of one of the branches of the procedure `node_set` unfortunately seems not to terminate.

To conclude, the automation of the discipline does not yet deliver consistently low verification times and seems to diverge in some cases. Much improvement has been achieved since the beginning of the project, but further research is required to achieve consistently satisfactory performance and less annotation.

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.

Acknowledgements. The authors are deeply grateful to P. Müller and to the three anonymous ESOP reviewers, whose deep and insightful comments significantly helped improve the quality of the paper.

References

1. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society (2002)
2. Parkinson, M.J., Summers, A.J.: The Relationship between Separation Logic and Implicit Dynamic Frames. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 439–458. Springer, Heidelberg (2011)
3. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)

4. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL 2005, pp. 259–270 (2005)
6. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
7. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput. 39(9), 1175–1185 (1990)
8. Summers, A., Drossopoulou, S., Müller, P.: The need for flexible object invariants. In: IWACO 2009, pp. 1–9. ACM (2009)
9. Kassios, I.T., Kritikos, E.: A discipline for program verification based on backpointers and its use in observational disjointness. Technical Report 772, Dept. of Computer Science, ETH Zurich (2012), `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=KassiosKritikos12.pdf`
10. Leino, K.R.M., Müller, P.: A Basis for Verifying Multi-threaded Programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
11. Heule, S., Leino, K.R.M., Müller, P., Summers, A.: Fractional permissions without the fractions. In: FTfJP 2011 (2011)
12. Leavens, G., Baker, A.L., Ruby, C.: JML: a notation for detailed design. In: Kilov, I., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer (1999)
13. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A Unified Framework for Verification Techniques for Object Invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
14. Leino, K.R.M., Müller, P.: Object Invariants in Dynamic Contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
15. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
16. Parkinson, M.: Class invariants: the end of the road? In: IWACO 2007 (2007)
17. Summers, A.J., Drossopoulou, S.: Considerate Reasoning and the Composite Design Pattern. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 328–344. Springer, Heidelberg (2010)
18. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
19. da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., Wheelhouse, M.: A simple abstraction for complex concurrent indexes. In: OOPSLA 2011, pp. 845–864. ACM (2011)
20. Jensen, J.B., Birkedal, L.: Fictional Separation Logic. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 377–396. Springer, Heidelberg (2012)
21. Mehnert, H., Sieczkowski, F., Birkedal, L., Sestoft, P.: Formalized Verification of Snapshotable Trees: Separation and Sharing. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 179–195. Springer, Heidelberg (2012)
22. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: STOC 1986, pp. 109–121. ACM (1986)