# Modular Verification of Intrusive List and Tree Data Structures in Separation Logic

**Marc Hermes**
Radboud University, Nijmegen, The Netherlands

**Robbert Krebbers**
Radboud University, Nijmegen, The Netherlands

## Abstract

Intrusive linked data structures are commonly used in low-level programming languages such as C for efficiency and to enable a form of generic types. Notably, intrusive versions of linked lists and search trees are used in the Linux kernel and the Boost C++ library. These data structures differ from ordinary data structures in the way that nodes contain only the meta data (*i.e.* pointers to other nodes), but not the data itself. Instead the programmer needs to embed nodes into the data, thereby avoiding pointer indirections, and allowing data to be part of several data structures.

In this paper we address the challenge of specifying and verifying intrusive data structures using separation logic. We aim for modular verification, where we first specify and verify the operations on the nodes (without the data) and then use these specifications to verify clients that attach data. We achieve this by employing a representation predicate that separates the data structure's node structure from the data that is attached to it. We apply our methodology to singly-linked lists – from which we build cyclic and doubly-linked lists – and binary trees – from which we build binary search trees. All verifications are conducted using the Coq proof assistant, making use of the Iris framework for separation logic.

**2012 ACM Subject Classification** Theory of computation → Separation logic; Theory of computation → Hoare logic; Theory of computation → Programming logic; Theory of computation → Data structures design and analysis

**Keywords and phrases** Separation Logic, Program Verification, Data Structures, Iris, Coq

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2024.19

**Supplementary Material** *Software (Code)*: `https://doi.org/10.5281/zenodo.12575047` [10]

**Acknowledgements** We thank Derek Dreyer, Laila Elbeheiry, Deepak Garg, Jules Jacobs, Ike Mulder and Michael Sammler for discussions, and the anonymous reviewers for their feedback. This research was supported in part by a generous award from Google Android Security's ASPIRE program.

## 1 Introduction

Linked data structures such as lists and trees are pervasive in imperative programming and serve as the implementation for various abstract data types such as queues, stacks, deques, sets and maps. Verification of these data structures therefore received a considerable amount of attention in the literature – *e.g.* the seminal papers on separation logic [30, 28] use linked lists and trees as their key examples, and many papers on verification tools use linked data structures as case studies [3, 4, 8, 9, 12, 27]. Yet, there is an unfortunate discrepancy between the way linked data structures are studied in the literature and the way they are implemented in systems programming, *e.g.* the Linux kernel [16, 23] and the Boost C++ library [20].

Let us first review the naive way to represent singly-linked lists in C that are "generic" in the element type:

```
struct list {
  void* data;
  struct list* next;
};
```


15th International Conference on Interactive Theorem Proving (ITP 2024).
Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 19; pp. 19:1–19:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Linked-list nodes contain the `data` and a pointer to the `next` node (the `NULL` pointer is used to represent the empty list). To avoid fixing the element type, the field `data` is a `void` pointer, meaning it could point to data of arbitrary type. This naive way of representing lists has a number of drawbacks in systems programming where efficiency is a key concern.

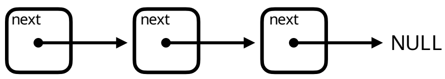
**Motivation #1.** The use of a pointer indirection for `data` requires additional storage and incurs a run-time cost on every read. This is in contrast to the "nongeneric" version where the data is stored directly in the struct:

```
struct int_list {
  int data;
  struct int_list* next;
};
```

Defining a specific version of linked lists for each element type is clearly undesirable – it means one has to duplicate all methods from the list API for each element type. Modern programming languages such as C++ and Rust offer generics and monomorphization to address this problem. It also possible to obtain efficient data types with generic elements in plain C, which we will illustrate in the following.

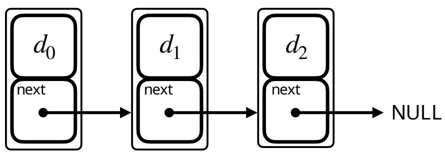
The Linux kernel uses the **intrusive** approach to linked lists, allowing for lists that can be re-used for different data types, while avoiding the overhead caused by pointer indirections. This is achieved by separating the meta data (*i.e.* the pointers to node) from the data. One first defines a `node` structure which is used to achieve the necessary linking:

```
struct node {
  struct node* next;
};
```




The list API can now be developed for the `node` structure, independently of the data. These nodes can then be incorporated as fields in other structures to create lists with values of any desired element type. Here we show the instantiation with integer values:

```
struct intrusive_int_list {
  int data;
  struct node node;
};
```




Compared to the naive approach, the functions implemented for the `node` API can be used to operate on lists with attached data, no matter the type of the data. To further illustrate this, we consider the function `replace_at`, which replaces the $n$-th element of an `intrusive_int_list` with a new integer value. In the implementation, we first define a function `get_pos` which yields a pointer to the $n$-th position in the `node` structure, and then use this function in `replace_at` to make the replacement at the correct position:

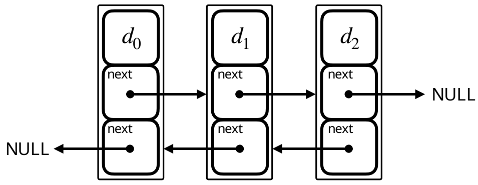
```
struct node* get_pos(int n, struct node* v) {
  if (n == 0 || v == NULL) return v;
  return get_pos(n-1, v->next);
}

#define container_of(ptr, container, field)
  (container*)((unsigned char*)(ptr) - offsetof(container,field))

void replace_at(int n, struct intrusive_int_list* l, data a) {
  struct node* pos = get_pos(n, &l->node);
  if (pos == NULL) return;
  struct intrusive_int_list* lp =
```

```
    container_of(pos, struct intrusive_int_list, node);
  lp->data = a;
}
```

In `replace_at`, we first create a pointer to the `node` field. If the call to `get_pos` returns a non-`NULL` node pointer `pos`, we make use of the `container_of` macro [24] to recover a pointer to its encompassing `intrusive_int_list` structure, in which we can then change the data field.

**Motivation #2.** The intrusive representation is particularly useful when elements are part of multiple data structures. For example, elements might be part of multiple search trees and a priority queue so that they can be retrieved efficiently in different orders. Let us consider a simpler example where a structure contains multiple singly-linked list nodes:

```
struct intrusive_int_list_2 {
  int data;
  struct node node_left;
  struct node node_right;
};
```




Here, we use `node_left` to keep track of the elements in left-to-right order, and `node_right` to keep track of the elements in right-to-left order. In other words, using two singly-linked intrusive lists we have constructed a doubly-linked list.

We emphasize that intrusive data structures also provide benefits in terms of allocation. When creating a new element, one allocates a new `intrusive_int_list_2`, which readily contains the list structures. This is in contrast to ordinary non-intrusive data structures, where one has to allocate the element, and insert (a pointer to) the element into both lists. Inserting an element into a list involves allocating a node, so this means there are three allocations in total, while the intrusive version needs just one allocation. Note that this means that allocation and deallocation are not handled by the `node` API, but that clients are put in charge of these tasks.

**Goal of the paper.** Formally specifying and verifying the correctness of intrusive data structures poses some interesting challenges. To illustrate this, let us consider the function `replace_at`. First, this function makes use of some involved pointer arithmetic by its use of the `container_of` macro. Secondly, the structures and functions are defined in a modular way, and it is desirable for the verification to follow this pattern. We say that the structures and functions are modular because instantiations (such as `intrusive_int_list`) use the `node` structure in an abstract way. They do not interact directly with the fields of `node`, and only use functions (such as `get_pos`) that operate on `node`. Similarly, we first want to specify `node` and verify `get_pos`, and then use these ingredients to specify `intrusive_int_list` and verify `replace`. In other words, the proof for `replace` should simply be about straight-line code, whereas the reasoning about the recursion should be done in the proof of `get_pos`.

Regarding the first point – to verify programs operating on pointers – we make use of separation logic [30, 28]. Since the exact programming language is not the issue we want to focus on, we will use a simpler language that has dynamic allocation, arrays and pointer arithmetic, but has none of the orthogonal challenges of C such as fixed-size integers, byte representations and alignment.

To formally verify the above code in separation logic, we need to formally describe the involved list structure. Below is the canonical way to specify `int_list` using a *representation predicate*, which associates a value $v$ with a mathematical list of integer data values $D$:

$$\begin{array}{ll} \texttt{IntList} : \texttt{val} \to \texttt{list int} \to \texttt{iProp} & \\ \texttt{IntList}\, v\, [\,] & \triangleq v = \mathsf{None} \\ \texttt{IntList}\, v\, (d :: D) & \triangleq \exists (l : \texttt{loc}).\, v = \mathsf{Some}\, l * \exists v'.\, l \mapsto [\, d, v' \,] \, * \, \texttt{IntList}\, v'\, D \end{array}$$

Here, the notation $l \mapsto [\, d, v' \,]$, or more generally $l \mapsto [\, v_0, \ldots, v_n \,]$, is an abbreviation for $l \mapsto v_0 * \ldots * l + n \mapsto v_n$, which provides unique ownership of an array storing values $v_0, \ldots, v_n$ at the locations $l$ to $l + n$. The connective $*$ is the separating conjunction, which is used to describe the disjointness of the resources. NULL pointers are modeled using the constructors None and Some of the option type.

Using the representation predicate, the specification for a function `replace_int_at` on `int_list` can be expressed in terms of a Hoare triple. We will use $\langle n := d \rangle D$ to denote the mathematical list obtained from $D$ by replacing its $n$-th element with $d$. If $n$ is larger than the length of the list, $D$ remains unchanged.

$$\{\, \texttt{IntList}\, v\, D \,\}\, \texttt{replace\_int\_at}\; n\; v\; d\, \{\, \texttt{IntList}\, v\, (\langle n := d \rangle D) \,\}$$

While representation predicates are commonly used to describe non-intrusive structures like `int_list`, our goal is to formulate and prove similar specifications for intrusive data structures like `intrusive_int_list`. Additionally, we would like to achieve this in a way that also formally captures and makes use of the modularity which underlies the definition of `intrusive_int_list` and the implementation of `replace_at`.

To modularly verify `replace_at`, we should state and verify a specification for `get_pos`. This brings us to the key observation about giving a specification to intrusive structures: We should closely follow the definition of `intrusive_int_list`, and first isolate the intrusive `node` part of the structure:

$$\begin{array}{ll} \texttt{Node} : \texttt{val} \to \texttt{list loc} \to \texttt{iProp} & \\ \texttt{Node}\; v\; [\,] & \triangleq v = \mathsf{None} \\ \texttt{Node}\; v\; (l :: L) & \triangleq v = \mathsf{Some}\, l * \exists v'.\, l \mapsto v' * \texttt{Node}\; v'\, L, \end{array}$$

In contrast to `IntList`, the above does not make reference to any list of data values and is instead linking locations taken from the list $L$. However, this now allows us to define a representation predicate for the intrusive structure `intrusive_int_list` in a straightforward yet novel way:

$$\texttt{IntrusiveIntList}\; v\; D \triangleq \exists\, L.\, \texttt{Node}\; v\; L * \mathop{\Huge *}_{l,d \in L,D}\, (l - 1) \mapsto d$$

In the above, the big separating conjunction of the form $\mathop{\Large *}_{l,d \in [\, l_0,\ldots,l_n \,],[\, d_0,\ldots,d_n \,]}$ runs over the pairs $(l_0, d_0), \ldots, (l_n, d_n)$, implicitly requiring the two appearing lists to be of equal length. The definition of `IntrusiveIntList` clearly exposes the underlying intrusive structure in the form of the `Node` predicate. We can now easily give a specification for `get_pos`, which we can then use in the verification of `replace_at`:

$$\{\, \texttt{Node}\, v\, L \,\}\, \texttt{get\_pos}\; n\; v\, \{\, x.\, x = \texttt{nth}\; n\; L * \texttt{Node}\, v\, L \,\}$$
$$\{\, \texttt{IntrusiveIntList}\, v\, D \,\}\, \texttt{replace\_at}\; n\; v\; d\, \{\, \texttt{IntrusiveIntList}\, v\, (\langle n := d \rangle D) \,\}$$

In the above specification, `nth` $n$ $L$ returns Some $l$ if $l$ is the $n$-th element of the list $L$, and returns None if there is no $n$-th element. Given our definition of `IntrusiveIntList` the verification of `replace_at` is both straightforward and modular. We can easily make use of the specification of `get_pos`, since its precondition `Node` $v$ $L$ conveniently appears as the left part of the separating conjunction in the definition of `IntrusiveIntList`.

**Summary of key idea.** The example illustrates the key idea we want to push forward in this paper: A separation of concerns when specifying intrusively implemented data types. One concern is the "shape" in which data is supposed to be stored, here given by the `node` structure. The second concern is the actual data itself, which we can think of as being attached to the shape. The underlying shape is then used for navigation on the data. Functions that have been implemented on the shape can, and should, then be used in a modular way when implementing functions that operate on the data. The verification of the specifications should then turn out to be modular as well.

We demonstrate this methodology through two examples: intrusive lists (Section 2) and intrusive binary search trees (Section 3). In both cases, we give representation predicates to specify the intrusive structure and then show how to use them to specify mutable data structures that carry data. In the case of trees, we will implement a similar function to the above `get_pos`, to locate a specific key in the tree, and to obtain a pointer to the corresponding node. Since this function leaves us with a partially traversed tree, we need to deal with the orthogonal challenge of specifying partial trees (Section 3.3), for which we employ 'the magic-wand as frame' approach by Cao et al. [7] (which has previously only been applied to ordinary data structures, not intrusive ones).

To summarize, the main contributions of this paper are:

- We introduce a specification for Linux-like intrusive singly-linked lists and sequences in separation logic. We use the specification to modularly build up intrusive singly-linked cyclic lists (Section 2.3) and doubly-linked cyclic lists (Section 2.4), and use them to implement data structures that carry data with them (Section 2.5).
- We apply our approach by verifying locate and insertion operations of binary search trees, which are based on intrusive binary trees (Section 3). In extension to what is done by Cao et al. [7], we consider intrusive data structures, and our definition of partial trees incorporates the invariant of a binary search tree.

All of the included structures, their operations and specifications have been defined and verified [10] in the Coq proof assistant, by making use of the Iris framework [13, 14, 15, 17, 18, 19] for separation logic.

## 2 Intrusive List Structures

In this section, we will gradually and modularly build intrusive list data structures. These structures link together locations in the heap, and do not carry any data with them. Their API allows a user to attach new locations as nodes to the list. This means that the user is responsible for the allocation and deallocation of the nodes, and the list structure is only used to manage the nodes. This can then be used by the user to form lists keeping track of data, which we will showcase by implementing a deque data structure.

After covering some preliminaries concerning the programming language and separation logic (Section 2.1), we start with the implementation of simple intrusive sequences (Section 2.2), and give specifications for some standard operations. We then continue by modularly using sequences to build intrusive cyclic lists (Section 2.3) and doubly-linked cyclic lists (Section 2.4). Lastly, we illustrate how the intrusive doubly-linked cyclic lists can be complemented with data values to implement a deque data structure (Section 2.5).

### 2.1 Preliminaries

We briefly go over some specifics of the programming language and program logic that will subsequently be used. We work in a $\lambda$-calculus which includes let expressions, if-then-else expressions, matching on terms, equality checks and mutable arrays:

| | |
|---|---|
| `alloc` $n\ v$ | Allocates $n$ successive locations in memory, initializes all of them with the value $v$, and returns the starting location of this array. |
| `free` $n\ l$ | Deallocates $n$ successive memory locations beginning from $l$. |
| $l \leftarrow v$ | Assigns the value $v$ to the location $l$. |
| $!\,l$ | Retrieves the value at memory location $l$. |

To reason about these operations, we utilize separation logic [30, 28], a variant of Hoare logic designed for imperative programs with pointers. Separation logic introduces the primitive $l \mapsto v$, separating conjunction $*$, and separating implication $-\!\!*$, which can be used to form assertions that are interpreted to describe fragments of the heap.

| | |
|---|---|
| `emp` | An empty heap fragment. |
| $l \mapsto v$ | Describes a memory fragment in which location $l$ contains the value $v$. |
| $P * Q$ | Disjoint union of the fragments described by $P$ and $Q$. |
| $P -\!\!* Q$ | Describes a heap fragment which satisfies $Q$ once it is combined with a disjoint fragment for which $P$ holds. |

We let `iProp` denote the set of separation logic assertions, differentiating it from the set of assertions `Prop` of the meta-logic (Coq). We allow pointer arithmetic on locations, meaning $l + n$ denotes the location $n$ steps away from $l$. This allows us to describe arrays of values $v_0, \ldots, v_n$ in the heap, by the formula $l \mapsto [\, v_0, \ldots, v_n \,] \triangleq l + 0 \mapsto v_0 * \ldots * l + n \mapsto v_n$. We often encounter situations where we want to make the assumption that a certain location is non-empty. This is expressed by $\exists v.\, l \mapsto v$, which we abbreviate as $l \mapsto \_$, and likewise extend the usage of the wildcard symbol "_" to arrays, as for example in $l \mapsto [\, \_, \_, v \,]$.

A key element in reasoning about program correctness within the framework of separation logic are Hoare triples. A Hoare triple is of the form $\{\, P \,\}\ e\ \{\, v.\, \Phi(v) \,\}$, where:

| | |
|---|---|
| $P$ | Assertion specifying the state of the heap before the execution of $e$. |
| $e$ | An expression in the programming language being analyzed. |
| $\Phi(v)$ | A predicate making an assertion about the return value $v$ and describing the state of the heap after the execution of $e$. |

The semantics of a Hoare triple is that if the initial state satisfies the precondition $P$, then the program $e$ will not crash, and if it finishes executing, the return value and final state will satisfy the postcondition $\Phi$. If the postcondition does not bind a return value, we simply write $\{P\}\, e\, \{Q\}$.

## 2.2 Sequences

A sequence [30] starting at location $l$ : `loc` links together a list $D$ : `list val` of values and stores a given default pointer $e$ : `loc` in the final node, giving its predicate the type signature `loc` → `list val` → `loc` → `iProp`. A standard definition of lists is similar to this, but would restrict it by demanding the last pointer to be a NULL pointer. Since our goal is to specify intrusive structures, we decouple the values from the shape of the sequence and define a representation predicate `Seq.pred` with signature `loc` → `list loc` → `loc` → `iProp`, which only links together a list of locations:

$$\texttt{Seq.pred}\, s\, [\,]\, e \triangleq s \mapsto e \qquad\qquad \texttt{Seq.pred}\, s\, (l :: L)\, e \triangleq s \mapsto l * \texttt{Seq.pred}\ l\ L\ e.$$

Our API for sequences includes a function to create a new intrusive sequence, push a new location to the front of the sequence, pop the first location, and to retrieve the next location in the sequence:

$$\begin{aligned}
\texttt{Seq.new}\ \mathit{start}\ \mathit{end} &:= \mathit{start} \leftarrow \mathit{end}\\
\texttt{Seq.push}\ \mathit{start}\ \mathit{new} &:= \mathit{new} \leftarrow !\,\mathit{start};\ \mathit{start} \leftarrow \mathit{new}\\
\texttt{Seq.pop}\ \mathit{start} &:= \texttt{let}\ \mathit{rem} = !\,\mathit{start}\ \texttt{in}\ \mathit{start} \leftarrow !\,\mathit{rem}\ ;\ \mathit{rem}\\
\texttt{Seq.next}\ \mathit{start} &:= !\,\mathit{start}
\end{aligned}$$

As alluded to by the naming, the predicate and functions are enclosed in a module named `Seq`. If the context makes it clear enough, we abbreviate the representation predicate `Seq.pred` by `Seq`, and drop the module name in the function names. We do so for all data structures that follow. The specifications for the operations on sequences are as follows:

$$\begin{array}{llll}
\{\, s \mapsto _ \,\} & \texttt{new}\ s\ e & \{\, \texttt{Seq}\ s\ [\,]\ e \,\}\\
\{\, l \mapsto _ * \texttt{Seq}\ s\ L\ e \,\} & \texttt{push}\ s\ l & \{\, \texttt{Seq}\ s\ (l :: L)\ e \,\}\\
\{\, \texttt{Seq}\ s\ (l :: L)\ e \,\} & \texttt{pop}\ s & \{\, p.\ p = l * \texttt{Seq}\ s\ L\ e * p \mapsto _ \,\}\\
\{\, \texttt{Seq}\ s\ L\ e \,\} & \texttt{next}\ s & \{\, p.\ p = \texttt{nth}\ 0\ (L +\!\!+ [e]) * \texttt{Seq}\ s\ L\ e \,\}
\end{array}$$

Recall that $\texttt{nth}\ n\ L$ returns the $n$-th element of a list or None if there is no such element. The specification clarifies that to create a new sequence at location $s$, the caller needs to own the location $s$ and provide a pointer $e$ to be stored inside $s$. The function `push` likewise requires the caller to have ownership of the location that is added to the sequence, and `pop` will return ownership of the popped location. This means that the operations do not perform allocation or deallocation of nodes, but are used to manage locations of the sequence.

We want to highlight the specification of `next`, as its postcondition, maybe unexpectedly, seems to return the sequence unchanged. In a non-empty cycle $\texttt{Seq}\ s\ (l :: L)\ e$ the call $\texttt{next}\ s$ returns the location $l$. Intuitively, the caller would now continue to operate on the sequence $\texttt{Seq}\ l\ L\ e$. Formally, this is done using the following "splitting" property:

$$\texttt{Seq}\ s_1\ (L_1 +\!\!+ s_2 :: L_2) \dashv\vdash \texttt{Seq}\ s_1\ L_1\ s_2 * \texttt{Seq}\ s_2\ L_2\ e \tag{1}$$

Here, $\dashv\vdash$ expresses interderivability of separation logic assertions, making it possible to use the rule in both directions. This means that one can use the rule in left-to-right direction to obtain a `Seq` predicate for any position in the list, then `push` or `pop` elements there, and finally use the right-to-left direction to reattain the `Seq` predicate for the whole sequence.

## 2.3 Cyclic Lists

We can now use the previously defined sequences to define intrusive cyclic lists. For this, we simply make use of the fact that we can freely choose the pointer stored at the end of a sequence and use it to point back to the start:

$$\texttt{Cycle.pred}\ c\ L := \texttt{Seq}\ c\ L\ c$$

The `Cycle` predicate satisfies the following "rotation" property, derived from the "splitting" property for `Seq` (1), reflecting its cyclic structure:

$$\texttt{Cycle}\ c\ (c' :: L) \dashv\vdash \texttt{Cycle}\ c'\ (L +\!\!+ [c]) \tag{2}$$

The API for cyclic lists is similar to the one for sequences, but includes a function that checks if a list is empty. This is done by comparing the starting location of the cycle to the location it points to. If the two are equal, the cycle is considered empty. Once we attach data to the intrusive cycle, the starting location takes the special role of a *sentinel node*.

$$
\begin{aligned}
\texttt{Cycle.new}\ \mathit{start} &:= \texttt{Seq.new}\ \mathit{start}\ \mathit{start}\\
\texttt{Cycle.is\_empty}\ \mathit{start} &:= !\,\mathit{start} = \mathit{start}\\
\texttt{Cycle.insert} &:= \texttt{Seq.push}\\
\texttt{Cycle.remove} &:= \texttt{Seq.pop}\\
\texttt{Cycle.next} &:= \texttt{Seq.next}
\end{aligned}
$$

Many of the functions are directly defined in terms of the sequence functions. The specifications are as follows, where `is_nil` returns a boolean reflecting if a list is empty:

$$
\begin{array}{lll}
\{\, c \mapsto \_ \,\} & \texttt{new}\ c & \{\, \texttt{Cycle}\, c\, [\,] \,\}\\
\{\, \texttt{Cycle}\, c\, L \,\} & \texttt{is\_empty}\ c & \{\, b.\ b = \texttt{is\_nil}\ L * \texttt{Cycle}\, c\, L \,\}\\
\{\, \texttt{Cycle}\, c\, L * l \mapsto \_ \,\} & \texttt{insert}\ c\ l & \{\, \texttt{Cycle}\, c\, (l :: L) \,\}\\
\{\, \texttt{Cycle}\, c\, (l :: L) \,\} & \texttt{remove}\ c & \{\, p.\ p = l * \texttt{Cycle}\, c\, L * p \mapsto \_ \,\}\\
\{\, \texttt{Cycle}\, c\, L \,\} & \texttt{next}\ c & \{\, p.\ p = \texttt{nth}\ 0\ (L +\!\!+ [c]) * \texttt{Cycle}\, c\, L \,\}
\end{array}
$$

Similar to sequences, we can use the specification of **`next`** in combination with the "rotation" property (2) to insertion and remove elements at arbitrary positions in the cycle.

## 2.4 Doubly-linked cyclic Lists

We now come to the more interesting example of intrusive doubly-linked cyclic lists, or *dcycles* for brevity. Conceptually, a dcycle is composed of a cyclically arranged nodes, each node containing two pointers, one to the next node, and one to the previous node. Another way to split a dcycle into two parts, is to collect all the forward pointers in a cycle, and all the backward pointers in a separate cycle.

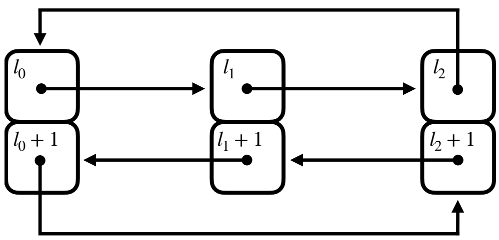



**Figure 1** Each node of the dcycle consists of two pointers, one pointing to the next, and one pointing to the previous node.

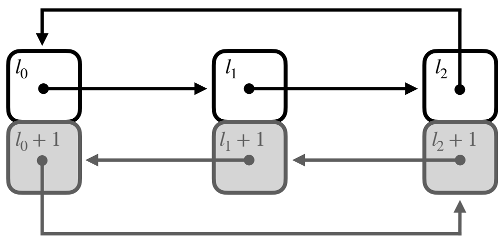



**Figure 2** The dcycle can be decomposed into two cycles, one containing all the `next` pointers (white), and one with all the `prev` pointers (grey).

Both combined make up the doubly-linked cyclic list. It is this latter view that motivates our choice for the representation predicate for dcycles, since it allows us to re-use the previous cycle specification in a straightforward way:

$$\texttt{DCycle.pred}\ c\ L := \texttt{Cycle}\ c\ L * \texttt{Cycle}\ (c+1)\ (\texttt{rev\_add\_1}\ L)$$

The function `rev_add_1` reverses the list of locations and adds 1 to every location, generating the cycle of backward pointers. By this definition, a node of the dcycle `DCycle.pred` $c$ $L$ at location $l \in L$ owns the resources $l \mapsto \texttt{next} * (l+1) \mapsto \texttt{prev}$, where `next` points to the next node in the dcycle and is owned by the underlying cycle `Cycle` $c$ $L$, and `prev` points to the previous one, and is owned by the cycle $\texttt{Cycle}\ (c+1)\ (\texttt{rev\_add\_1}\ L)$ (Figure 2). Basic operations on dcycles are implemented in a way that leverages the functions already provided by the underlying cycles. For readability, we introduce some notation for operations used in accessing the next and previous pointer of a node: given a location $l$ in a dcycle, we write $l.\texttt{next}$ for $l+0$ and $l.\texttt{prev}$ for $l+1$.

$$\begin{aligned}
\texttt{DCycle.new}\ c &:= \texttt{Cycle.new}\ c.\texttt{next}\ ;\ \texttt{Cycle.new}\ c.\texttt{prev}\\
\texttt{DCycle.is_empty}\ c &:= \texttt{Cycle.is_empty}\ c.\texttt{next}\\
\texttt{DCycle.next}\ c &:= \texttt{Cycle.next}\ c.\texttt{next}\\
\texttt{DCycle.prev}\ c &:= (\texttt{Cycle.next}\ c.\texttt{prev}) + (-1)\\
\texttt{DCycle.insert}_0\ c\ \mathit{new} &:= \textbf{let}\ \mathit{next} = \texttt{DCycle.next}\ c\ \textbf{in}\\
&\phantom{:=}\ \ \texttt{Cycle.insert}\ \mathit{next}.\texttt{prev}\ \mathit{new}.\texttt{prev}\ ;\\
&\phantom{:=}\ \ \texttt{Cycle.insert}\ c.\texttt{next}\ \mathit{new}.\texttt{next}\\
\texttt{DCycle.remove}_0\ c &:= \textbf{let}\ \mathit{nn} = \texttt{DCycle.next}\ (\texttt{DCycle.next}\ c)\ \textbf{in}\\
&\phantom{:=}\ \ \textbf{let}\ l_0 = \texttt{Cycle.remove}\ c.\texttt{next}\ \textbf{in}\\
&\phantom{:=}\ \ \texttt{Cycle.remove}\ \mathit{nn}.\texttt{prev}\ ;\ l_0
\end{aligned}$$

The function $\texttt{insert}_0$ and $\texttt{remove}_0$ are used to insertion and remove the node that comes after the current node. We can also define functions $\texttt{insert}_1$ and $\texttt{remove}_1$, which do the same operations in the other direction of the dcycle. We omit these here, but they can be found in the Coq mechanization. The specifications of the dcycle functions are:

$$\begin{array}{lll}
\{\, c \mapsto [_,_] \,\} & \texttt{new}\ c & \{\, \texttt{DCycle}\ c\ [\,] \,\}\\
\{\, \texttt{DCycle}\ c\ L \,\} & \texttt{is_empty}\ c & \{\, b.\ b = \texttt{is_nil}\ L * \texttt{DCycle}\ c\ L \,\}\\
\{\, \texttt{DCycle}\ c\ L \,\} & \texttt{next}\ c & \{\, p.\ p = \texttt{nth}\ 0\ (L \mathbin{+\!\!+} [c]) * \texttt{DCycle}\ c\ L \,\}\\
\{\, \texttt{DCycle}\ c\ L \,\} & \texttt{prev}\ c & \{\, p.\ p = \texttt{nth}\ 0\ (\texttt{rev}\ L \mathbin{+\!\!+} [c]) * \texttt{DCycle}\ c\ L \,\}\\
\{\, \texttt{DCycle}\ c\ L * l \mapsto [_,_] \,\} & \texttt{insert}_0\ c\ l & \{\, \texttt{DCycle}\ c\ (l :: L) \,\}\\
\{\, \texttt{DCycle}\ c\ (l :: L) \,\} & \texttt{remove}_0\ c & \{\, p.\ p = l * \texttt{DCycle}\ c\ L * p \mapsto [_,_] \,\}
\end{array}$$

We often need to make use of a "rotation" property, which analogously to the similar property for cycles (2), allows us to cyclically rotate the locations of the dcycle:

$$\texttt{DCycle}\ c\ (c' :: L) \dashv\vdash \texttt{DCycle}\ c'\ (L \mathbin{+\!\!+} [c]) \tag{3}$$

To illustrate in how far the chosen definition of the dcycle representation predicate allows for modular reasoning, let us consider what happens during the verification of $\texttt{DCycle.remove}_0$. Its specification as given above is:

$$\{\, \texttt{DCycle}\ c\ (l :: L) \,\}\ \texttt{DCycle.remove}_0\ c\ \{\, p.\ p = l * \texttt{DCycle}\ c\ L * p \mapsto [\_,\_] \,\}$$

The function $\texttt{DCycle.remove}_0$ first makes two calls to `DCycle.next`. The specification of `DCycle.next` keeps the initial `DCycle` predicate unaltered, so we make use of the rotation property. Next, there are two calls to the function `Cycle.remove`, each separately effecting one of the two underlying cycles of the dcycle. At this point, we would like to use the already verified specification of `Cycle.remove` from Section 2.3. Fortunately, we can achieve this by unfolding the definition of the `DCycle` predicate:

$$\begin{aligned}
&\texttt{DCycle}\ c\ (l :: L)\\
\dashv\vdash\ &\texttt{Cycle}\ c\ (l :: L) * \texttt{Cycle}\ (c+1)\ (\texttt{rev_add_1}\ (l :: L))
\end{aligned}$$

This then allows us to reason on the separate cycles and make use of the `Cycle.remove` specification twice.

## 2.5 Deques

We now illustrate how the intrusively treated dcycles can be used to implement and verify a deque data structure. A deque represents a linearly arranged list of elements, supporting push and pop operations for the addition and removal of elements at both the front and the end of the list. The cyclic nature of dcycles makes it convenient to implement a deque, since we can use the first node as a sentinel. Using a dcycle as the underlying structure, defining the representation predicate is also rather simple: we associate the nodes of the dcycle with the data that is supposed to be stored next to it:

$$\texttt{Deque.pred}\ c\ D := \exists L.\ \texttt{DCycle}\ c\ L\ * \mathop{\huge *}_{l,d \in L,D} (l-1) \mapsto d$$

Since a node of the underlying dcycle at location $l$ stores two pointers, the data is stored at location $l-1$. From the entry-point $c$ of the dcycle (which does not get any data), we can then make changes at the head and tail of the deque. Creation, push and pop operations for the deque are defined as follows:

$$\begin{aligned}
\texttt{Deque.new}\ () &:= \texttt{let}\ l = \texttt{alloc}\ 2\ \textsf{None}\ \texttt{in}\ \texttt{DCycle.new}\ l\ ;\ l \\
\texttt{Deque.push_front}\ x\ c &:= \texttt{let}\ l_x = \texttt{alloc}\ 3\ x\ \texttt{in}\ \texttt{DCycle.insert}_0\ c\ (l_x + 1) \\
\texttt{Deque.push_back}\ x\ c &:= \texttt{let}\ l_x = \texttt{alloc}\ 3\ x\ \texttt{in}\ \texttt{DCycle.insert}_1\ c\ (l_x + 1) \\
\texttt{Deque.pop_front}\ c &:= \texttt{if}\ \texttt{DCycle.is_empty}\ c\ \texttt{then}\ \textsf{None}\ \texttt{else} \\
&\phantom{:=}\ \texttt{let}\ \mathit{next} = \texttt{DCycle.next}\ c\ \texttt{in} \\
&\phantom{:=}\ \texttt{let}\ x = !\,\mathit{next}.\texttt{data}\ \texttt{in} \\
&\phantom{:=}\ \texttt{let}\ \mathit{rem} = \texttt{DCycle.remove}_0\ c\ \texttt{in} \\
&\phantom{:=}\ \texttt{free}\ 3\ (\mathit{rem} - 1)\ ;\ \textsf{Some}\ x \\
\texttt{Deque.pop_back}\ c &:= \texttt{if}\ \texttt{DCycle.is_empty}\ c\ \texttt{then}\ \textsf{None}\ \texttt{else} \\
&\phantom{:=}\ \texttt{let}\ \mathit{prev} = \texttt{DCycle.prev}\ c\ \texttt{in} \\
&\phantom{:=}\ \texttt{let}\ x = !\,\mathit{prev}.\texttt{data}\ \texttt{in} \\
&\phantom{:=}\ \texttt{let}\ \mathit{rem} = \texttt{DCycle.remove}_1\ c\ \texttt{in} \\
&\phantom{:=}\ \texttt{free}\ 3\ (\mathit{rem} - 1)\ ;\ \textsf{Some}\ x
\end{aligned}$$

Here, $l.\texttt{data}$ is notation for $l-1$. Note that `Deque.new` only makes an allocation for an array of length 2, since it only creates the sentinel node of the underlying dcycle, which does not get to hold any data.

Thanks to the already verified specifications for the operations on dcycles, available lemmas about the big separating conjunction in the library of Iris, and the usage of the proof automation framework Diaframe [26], verifying the specifications of the deque API surmounts to less than 30 lines of proof in Coq.

# 3 Intrusive Binary Search Trees

In this section, we discuss and specify intrusive trees (Section 3.1), akin to those found in the Linux kernel [32, 22], where they are used for the implementation of the red-black trees. In the introduction (Section 1) we gave an example where the replacement of an element in a list was split into two parts: first, finding the right position in the intrusive list and returning a pointer to that location, and second, using this pointer to make the replacement in the corresponding data field. In the case of binary search trees, our implementation of the insertion operation will likewise be done in two steps. It first uses a function `locate` to find the correct position for the insertion – making use of the intrusive structure for navigation – and then carry out the insertion in this position. Again, it will be our goal to show that the verification of the final insertion operation can be done modularly (Section 3.3).

Since `locate` searches for – and potentially stops – at an arbitrary position inside a tree, we need to deal with partially traversed binary trees. In Section 3.3 we will show how we can deal with this by using the "magic wand as frame" approach [7], which we have also adapted (Section 3.2) to deal with the verification of properties of the functional `locate` function.

## 3.1 Representation Predicates

To illustrate a use of binary search trees, our overall goal will be to use them to implement a map data structure, which keeps track of key-value pairs by making use of an underlying binary search tree.

To start, we specify the intrusive tree structure, which relates a tree $t$ : `tree loc` labeled with locations – each one the location of a node of the heap representation of the tree – to the root-location $l$ : `loc` of the tree.

$$\begin{aligned}
&\texttt{Tree.pred} : \texttt{loc} \to \texttt{tree loc} \to \texttt{iProp}\\
&\texttt{Tree.pred}\ l\ \mathsf{Leaf} := l \mapsto \mathsf{None}\\
&\texttt{Tree.pred}\ l\ (\texttt{Node}\ p\ t_1\ t_2) := l \mapsto \mathsf{Some}\ p\ *\ \texttt{Tree.pred}\ p\ t_1\ *\ \texttt{Tree.pred}\ (p+1)\ t_2
\end{aligned}$$

In the above, we make use of polymorphic typed trees in Coq, since we will use them with different types for the labels.

$$\texttt{tree}\ (A : \mathsf{Type}) ::= \texttt{Leaf} : \texttt{tree}\ A \mid \texttt{Node} : A \to \texttt{tree}\ A \to \texttt{tree}\ A \to \texttt{tree}\ A$$

We also define standard `map` and `inorder` functions on those trees. So far we have only specified the intrusive binary tree shape. To get binary search trees, our next step is to add a key of type $K$ to every node in the tree, changing the signature to take trees of type `tree (K * loc)`, and to enforce the binary search tree invariant. We restrict our attention to binary search trees that use natural numbers as their keys (*i.e.* `K` = `nat`), and we rely on a Coq predicate `BST_inv_nat` : `tree nat` → `Prop` to describe binary search trees on the level of Coq. Combining the above, the desired heap predicate for binary search trees is:

$$\begin{aligned}
&\texttt{BST.pred} : \texttt{loc} \to \texttt{tree (K * loc)} \to \texttt{iProp}\\
&\texttt{BST.pred}\ l\ t := \texttt{Tree.pred}\ l\ (\pi_1\ t)\ *\ \texttt{BST_inv_nat}\ (\pi_2\ t)\ *\ \mathop{\Large *}_{(k,l) \in \texttt{inorder}\ t}\ (l+2) \mapsto k
\end{aligned}$$

The predicate is a separating conjunction of three parts:

- The shape of the tree, which must hold for the tree of locations that we get from the first projection $\pi_1\ t = \texttt{map fst}\ t$ of the tree $t$ : `tree (K * loc)` of key-location pairs.
- The binary search tree invariant, which must hold for the tree of natural numbers that we get from the second projection $\pi_2\ t = \texttt{map snd}\ t$ of the tree.
- A big separating conjunction over every key-location pair $(k, l)$ in the tree $t$, and indicating where the key is stored. Notably, the structure of the tree plays no role here.

Using binary search trees, we can now specify finite maps $\texttt{K} \xrightarrow{\text{fin}} \texttt{val}$ of key-value pairs. This is first done by specifying a finite map connecting keys and locations, which can then be used to attach data to the locations.

$$\begin{aligned}
&\texttt{MapNode.pred} : \texttt{loc} \to (\texttt{K} \xrightarrow{\text{fin}} \texttt{loc}) \to \texttt{iProp}\\
&\texttt{MapNode.pred}\ l\ m := \exists t.\ \texttt{BST.pred}\ l\ t\ *\ m = \texttt{to_map}\ t
\end{aligned}$$

$$\begin{aligned}
&\texttt{Map.pred} : \texttt{loc} \to (\texttt{K} \xrightarrow{\text{fin}} \texttt{val}) \to \texttt{iProp}\\
&\texttt{Map.pred}\ l\ m := \exists m'.\ \texttt{MapNode.pred}\ l\ m'\ *\ \mathop{\Large *}_{(l,v) \in m', m}\ (l-1) \mapsto v
\end{aligned}$$

The specification of the insertion operation for intrusive maps is then:

$$\{\, \mathit{new} \mapsto [\,\mathsf{None}, \mathsf{None}, k\,] * \texttt{MapNode}\ l\ m \,\}$$
$$\texttt{MapNode.insert}\ l\ \mathit{new}$$
$$\{\, ml.\ \texttt{MapNode}\ l\ (\langle k := \mathit{new}\rangle m)$$
$$*\ \texttt{if}\ m(k) = \mathsf{Some}\ l'\ \texttt{then}\ ml = \mathsf{Some}\ l' * l' \mapsto [\_, \_, k]\ \texttt{else}\ ml = \mathsf{None} \,\}$$

Here, $\langle k := \mathit{new}\rangle m$ is the map $m$ extended with the key-value pair $(k, \mathit{new})$ (the value of $k$ is overwritten if it already exists). Note that by definition of `Tree.pred` the location $l$ is always the entry point of the tree and not subject to any changes the function `insert` makes.

To verify the above, we make immediate use of the specification of insertion for trees, which will be discussed in Section 3.3. After using it we are left with proof obligations about the mathematical trees, which can be resolved by previously established lemmas, and lemmas from the library. Finishing the example, we can then use the above to verify the insertion operation on maps that have values attached:

$$\{\texttt{Map}\ l\ m\}\ \texttt{Map.insert}\ l\ k\ v\ \{\texttt{Map}\ l\ (\langle k := v\rangle m)\}$$

In the next two subsections we cover the definition and verification of the locate and insertion functions on binary search trees, first as functional versions on the level of the meta-theory (Section 3.2), and then implemented in the imperative object language (Section 3.3).

## 3.2 Functional Implementation of Tree Functions

To specify representation predicates for binary search trees we made use of polymorphic trees on the level of Coq. To give specifications for the `locate` and `insert` operations, we also need to implement functional versions of them. Our choice of implementing `insert` by making use of `locate`, instead of the more standard recursive definition, will lead to an interesting challenge when it comes to verifying that `insert` preserves the binary search tree invariant. Dealing with this is the main technical aspect we would like to highlight in this section.

We again try to keep things polymorphic and assume an arbitrary type `K` of keys, and a boolean comparison function `p : K → K → bool`. Functional implementations of `locate` and `insert` are then given by the following Coq code:

```
Fixpoint locate (k : K) (Γ  : tree K → tree K) (t' : tree K) :  (tree K → tree K) ∗ tree K :=
  match t' with
  | Leaf        ⇒  (Γ , Leaf)
  | Node k' l r ⇒  if p k  k' then locate k (λ h, Γ (Node k' h r)) l
                   else if p k' k  then locate k (λ h, Γ (Node k' l h)) r
                   else (Γ , t')
  end.

Definition insert (k : K) (t : tree K) : tree K :=
  match locate k id t with
  | (Γ ', Leaf)        ⇒ Γ ' (Node k Leaf Leaf)
  | (Γ ', Node _ l r) ⇒ Γ ' (Node k l r)
  end.
```

The argument $\Gamma$ : `tree K → tree K` of `locate` is used as a form a functional zipper [11] or *context* – it keeps track of the tree that is left behind, as we traverse down in search of the key. We can also think of the function $\Gamma$ as a partial tree with one hole, waiting for a tree as input in order to be completed to a full tree. Since not all functions correspond to partial trees that result from traversing down a tree (*e.g.* λt, Node t t ), we define a predicate ctx to define properly formed contexts. The constructors capture the ways in which locate enlarges the context during a recursive call.

```
Inductive ctx : (tree K → tree K) → Prop :=
| ctx_id : ctx id
| ctx_ht k t Γ : ctx Γ → ctx (λ h, Γ (Node k h t))
| ctx_th k t Γ : ctx Γ → ctx (λ h, Γ (Node k t h)).
```

Making use of the comparing function p we define the binary search tree invariant BST for trees over K, and make the assumption that p is asymmetric and antisymmetric to ensure that the invariant has the expected properties.

```
Inductive BST_inv : tree K →Prop :=
| BST_Leaf : BST_inv Leaf
| BST_Node x l r : BST_inv l → (∀ y, y ∈ to_set l → p y x) →
                   BST_inv r → (∀ y, y ∈ to_set r → p x y) →
                   BST_inv (Node x l r).
```

The above is the invariant which we specialized to natural numbers (BST_inv_nat) in Section 3.1. Next we want to prove that insert preserves the BST invariant, since this is a property that will be needed in the verification of the imperative implementation of insert.

Compared to the recursive implementation of insert, the usage of locate complicates this verification. A specification for locate needs to faithfully capture that the function can potentially stop in the middle of a tree, leaving behind a partially traversed tree and the root of a tree that has the sought after key.

To formally capture partially traversed trees, we define the predicate BST_ctx.

```
Definition BST_ctx (Γ : tree K → tree K) C C' :=
  forall t, BST_inv t → to_set t ⊆ C → BST_inv (Γ t) ∧ to_set (Γ t) ⊆ C'.

Lemma BST_ctx_locate_spec {k Γ t Γ ' t'} C :
  locate k Γ t = (Γ ', t') →
  BST_inv t ∧ BST_ctx Γ  ({k} ∪ to_set t) ({k} ∪ C) →
  BST_inv t' ∧ BST_ctx Γ ' ({k} ∪ to_set t') ({k} ∪ C).
```

The assertion BST_ctx Γ C C' expresses that for every BST $t$ with keys in the set $C$, passing it to Γ will yield another binary search tree $\Gamma\, t$ whose keys are a subset of $C'$.

The above specification of locate can now be shown by induction on the BST invariant of the input tree $t$ and making sure that the induction hypothesis generalizes over Γ. It also makes use of some properties of BST_ctx, which establish base cases and compositionality, and readily follow from the definition:

- $C \subseteq C' \to$ BST_ctx id $C\ C'$
- $(\forall y.\, y \in C \to p\, y\, k) \to$ BST_inv $t \to$ BST_ctx $(\lambda h,$ Node $k\, h\, t)$ $C$ $(\{k\} \cup C \cup$ to_set $t)$
- $(\forall y.\, y \in C \to p\, k\, y) \to$ BST_inv $t \to$ BST_ctx $(\lambda h,$ Node $k\, t\, h)$ $C$ $(\{k\} \cup C \cup$ to_set $t)$
- BST_ctx Γ $A\ B \to$ BST_ctx $\Gamma'$ $B\ C \to$ BST_ctx $(\Gamma' \circ \Gamma)$ $A\ C$

To prove that insert preserves the BST invariant, we only need the special case of the above lemma, where Γ = id and C = to_set t This gives us:

```
locate k id t = (Γ ', t') → BST_inv t →
BST_inv t' ∧ BST_ctx Γ ' ({k} ∪ to_set t') ({k} ∪ to_set t).
```

By case analysis on the tree $t$, we can then show the desired preservation property of insert:

```
Lemma BST_inv_insert k t : BST_inv t → BST_inv (insert k t).
```

### 3.3 Locate and Insertion on Binary Search Trees

We come to a minimal API for binary search tree, only including a function to create a new tree and one to insert a new element. We again introduce notations for accessing the fields of a node: $l.\texttt{left}$ for $l+0$, $l.\texttt{right}$ for $l+1$, and $l.\texttt{key}$ for $l+2$.

$$
\begin{aligned}
&\texttt{BST.new}\, l := l \leftarrow \mathsf{None}\\
&\texttt{BST.locate}\, pos\, k := \textbf{match}\, !\, pos\, \textbf{with}\\
&\quad \mathsf{None} \;\Rightarrow pos\\
&\quad |\ \mathsf{Some}\ l \Rightarrow \textbf{let}\, k' = !\, l.\texttt{key}\, \textbf{in}\\
&\qquad \textbf{if}\, k < k'\, \textbf{then}\, \texttt{BST.locate}\ l.\texttt{left}\ k\\
&\qquad \textbf{else if}\, k > k'\, \textbf{then}\, \texttt{BST.locate}\ l.\texttt{right}\ k\\
&\qquad \textbf{else}\, pos\\
&\quad \textbf{end}\\
&\texttt{BST.insert}\, root\, new := \textbf{let}\, new\_key = !\, new.\texttt{key}\, \textbf{in}\\
&\quad \textbf{let}\, pos = \texttt{BST.locate}\ root\ new\_key\, \textbf{in}\\
&\quad \textbf{match}\, !\, pos\, \textbf{with}\\
&\quad \mathsf{None} \;\Rightarrow pos \leftarrow new\ ;\ \mathsf{None}\\
&\quad |\ \mathsf{Some}\ l \Rightarrow new.\texttt{left} \leftarrow !\, l.\texttt{left}\ ;\\
&\qquad new.\texttt{right} \leftarrow !\, l.\texttt{right}\ ;\\
&\qquad pos \leftarrow new\ ;\ \mathsf{Some}\ l\\
&\quad \textbf{end}
\end{aligned}
$$

Running `locate` to find a key $k$ in the tree $t$, will return a value with the location of the node in $t$ that contains the key $k$, or None if no such node exists. Notably, if the key is found, this means we get a pointer to a node in the tree. The challenge is now to find a way to give a specification for `locate`, since it must somehow mention this pointer to an internal node of the tree. This is not yet possible with the tree predicate we have given so far. The specification of `locate` should assure that the function will always successfully run on a BST.

$$\{\, \texttt{BST}\ l\ t\, \}\ \texttt{locate}\ l\ k\ \{\, l'.\ \Phi\, l'\, \}$$

We still need to determine the predicate for the postcondition $\Phi$. On the one hand, it will need to express that the returned pointer $l'$ is the root of some subtree, which can be done by using the `BST` predicate. But apart from this subtree, $\Phi$ also has to account for the remainder of the initial tree $t$. Instead of coming up with a new data structure in Coq to describe these partial trees, we deal with this by following the "magic wand as frame" approach [7], which makes use of the separating implication $-\!\!*$ and functions $\Gamma$ to define partial trees.

$$
\begin{aligned}
&\texttt{part\_BST} : \texttt{loc} \to (\texttt{tree}\ (\texttt{nat} * \texttt{loc}) \to \texttt{tree}\ (\texttt{nat} * \texttt{loc})) \to \texttt{loc} \to \texttt{iProp}\\
&\texttt{part\_BST}\ l\ \Gamma\ p := \forall t'.\ \big(\texttt{BST.pred}\ p\ t' \ * \ \texttt{BST\_inv\_nat}\ (\pi_1(\Gamma\ t'))\big)\ -\!\!*\ \texttt{BST.pred}\ l\ (\Gamma\ t')
\end{aligned}
$$

In the predicate `part_BST l` $\Gamma$ `p`, the location $l$ is the root of the partial tree, $p$ is the location which is missing a subtree, and $\Gamma$ can intuitively be viewed as the remaining partial tree. Formally, during the proof, we require $\Gamma$ to correspond to proper contexts, so we use the `ctx` predicate to enforce this. We can now formulate the specification for `locate` as follows:

$$
\begin{aligned}
&\texttt{ctx}\ \Gamma \to \texttt{tree.locate}\ k\ s\ \texttt{id}\ t = (\Gamma', t') \to\\
&\{\, \texttt{BST}\ l\ t\, \}\ \texttt{locate}\ l\ k\ \{ l'.\ \texttt{part\_BST}\ l\ \Gamma'\ l' \ * \ \texttt{BST}\ l'\ t'\, \}
\end{aligned}
$$

Here, `tree.locate` is the functional implementation of `locate` in Coq (Section 3.2). The verification of the above specification is done by induction on the tree $t$. Since `locate` is running a loop, we will need a loop invariant. A first proof attempt quickly reveals that during a loop of `locate`, the precondition involving `BST` can usually not be restored, since the tree is only partially traversed. But we can generalize the precondition to fix this issue.

$$\texttt{ctx}\,\Gamma \to \texttt{tree.locate}\; k\; s\; \Gamma\; t = (\Gamma', t') \to$$
$$\{\,\texttt{part\_BST}\; l\; \Gamma\; p\; *\; \texttt{BST}\; p\; t\,\}\; \texttt{locate}\; p\; k\; \{p'.\; \texttt{part\_BST}\; l\; \Gamma'\; p'\; *\; \texttt{BST}\; p'\; t'\,\}$$

The above can then be proven by induction on the tree $t$ while making sure to generalize over all other parameters. The specification of `locate` can then be used to verify the insertion function, only requiring a case distinction on the tree, and no further proof by induction.

$$\{\, \mathit{new} \mapsto [\,\mathsf{None}, \mathsf{None}, k\,] * \texttt{BST}\; l\; t\,\}\; \texttt{insert}\; l\; \mathit{new}\; \{\,\texttt{BST}\; l\; (\texttt{tree.insert}\; k\; \mathit{new}\; t)\,\}$$

The proof makes use of the fact that the functional implementation of insertion preserves the BST invariant of the tree, which was discussed in Section 3.2.

## 4 Mechanization

All of the above presented data structures, specification and related proofs have been fully mechanized in the Coq proof assistant, making use of the Iris framework for separation logic. Apart from some notational short hands, the definitions and theorem statements in the paper directly reflect their counterparts in Coq.

In the verification, we additionally make use of Diaframe [26], which is a proof automation framework for Iris. It employs a goal-directed proof search strategy which can be extended by the user. The object programming language we use and study with Iris is HeapLang, which is the standard language provided by Iris, and is used without further adjustments. Regarding Diaframe, we added a few lines of code to enhance some simple handling of pointer arithmetic. These latter lines can be found in the `Setup.v` file.

The proofs remain rather simple for the sequences and cyclic lists, but start to get slightly more involved once we layer the intrusive lists in the case of the dcycles (Section 2.4). The representation predicate needs to be unfolded and its constituents manipulated in very deliberate ways, by making use of the rotation property of the cycle predicate. The same can be said about the verification in the case of the binary search trees. Here, the main formalization overhead (`Util.v`) was in relation to the underlying mathematical trees, which turns out to be significantly larger than the corresponding one for lists. This was mainly due to the choice of implementing insertion via locate, which required different proof approaches compared to the recursive version, but allowed us to discuss the problem of internal pointers in tree structures.

## 5 Related Work

**Intrusive Data Structures.** As part of effort in verifying Google's pKVM hypervisor for Android, Pulte et al. [29] verify the buddy allocator [1] used in the hypervisor. The corresponding C code makes use of intrusive lists (`list_head`), which are specified as part of the main invariant in the verification. The intrusive data structure is however not identified and specified as an independent structure. Lee et al. [21] consider intrusive data structures in the context of Rust, where they focus the issue of type-checking intrusive structures in ownership type-systems.

**Linked List in Separation Logic.** Linked lists are a standard data structure covered in any introduction to separation logic [3, 5, 30]. They also serve as a natural target to benchmark verification tools and verifications can therefore for example be found in among others Bedrock [9], Charge [4], VST [3, 6], Viper [27] and Verifast [31, 12]. In all of these cases, lists are specified in a non-intrusive way, similar to `is_int_list` in the introduction (Section 1).

**Magic Wand as Frame.** Cao et al. [7] utilize the magic wand to describe partial data structures, which avoids the introduction of another recursively defined predicate to specify these kinds of data "frames". While we have adopted their approach for defining partial trees by usage of the magic wand, our definition still differs. They define a partial tree that only represents the partial tree shape. We however actually define a partial *binary search tree*, *i.e.* the values in our partial structure define are sorted according to the BST invariant.

**Higher-Order Representation Predicates in Separation Logic.** Charguéraud [8] showcases how higher-order representation predicates can be used to describe polymorphic mutable data structures. He covers a wide range of standard structures, which includes mutable lists, list segments, records, trees and arrays. Similarly to the example we have given in Section 1, he treats the example of a function to read the $n$-th cell of a mutable list, and uses a predicate designating a list segment to be able to formulate the specification. Likewise, he continues by treating trees and showing techniques of how to represent trees with holes, *i.e.* trees, where the ownership of possibly several subtrees has been detached. Charguéraud uses recursively defined predicates to describe the structures with holes, whereas we have made use of the "magic wand as frame" approach. Lists are treated non-intrusively, and he does not cover cyclic data structures or binary search trees.

## 6 Conclusion

In this paper, we have presented an approach to specify intrusive data structures by first separately specifying the underlying node structure before the addition of data. One key feature of intrusive data structures is that they can be combined: they can be used to keep track of data across multiple intrusive data structures. With our given approach, this can easily be captured, since we can combine specifications of intrusive data structures. We illustrated this by using two cyclic lists to track data, effectively giving us the implementation of a deque (Section 2.5).

Throughout the presented examples, we have chosen a modular approach when it came to specifying the list and tree data structures. This is particularly evident in the specifications of the cyclic and doubly-cyclic lists. But this was not only limited to specifications; whenever we implemented a new function on a data structure, we made sure to reuse operations provided by any underlying structure. As a consequence, verifying the specifications of the presented data structures also ended up decomposing in a modular way. Since our mechanization makes use of the Iris framework and Diaframe, most proofs get simplified to the point where the only proof obligations that were left were related to the logical representations of the data.

Looking ahead, it remains to be determined to which extent this modular approach can be applied to more complex graph-like structures. The Linux kernel makes use of a task structure [25], which contains multiple intrusive list node occurrences (`list_head`) and with the addition of other intrusive structures, and the buddy allocator in pKVM [1, 2] makes use of intrusive lists to keep track of free blocks. So there are natural examples of data structures in which many intrusive structures are embedded. Some scaling challenges will probably appear when tying the mathematical structures – which outline the shape and carry information about the location of nodes – to the heap representations of the intrusive structures. In the dcycle example (Section 2.4) this happened by the usage of the `rev_add_1` function. Many similar functions, or a complex relation, will most likely be needed in order to specify a structure that involves several embedded intrusive data structures.

## References


1 Android Open Source Project. Buddy allocator in pKVM, 2024. URL: `https://github.com/torvalds/linux/blob/master/arch/arm64/kvm/hyp/nvhe/page_alloc.c`.

2 Android Open Source Project. Source code of the `hyp_pool` structure used in the pKVM implementation of the buddy allocator, 2024. URL: `https://github.com/torvalds/linux/blob/bf3a69c6861ff4dc7892d895c87074af7bc1c400/arch/arm64/kvm/hyp/include/nvhe/gfp.h#L12`.

3 Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

4 Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! In *ITP*, pages 315–331, 2012. `doi:10.1007/978-3-642-32347-8_21`.

5 Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic, 2024. URL: `https://iris-project.org/tutorial-material.html`.

6 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *JAR*, 61(1):367–422, 2018. `doi:10.1007/s10817-018-9457-5`.

7 Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W Appel. Proof Pearl: Magic Wand as Frame, 2018.

8 Arthur Charguéraud. Higher-order representation predicates in separation logic. In *CPP*, pages 3–14, 2016. `doi:10.1145/2854065.2854068`.

9 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011. `doi:10.1145/1993498.1993526`.

10 Marc Hermes. Coq Mechanization of “Modular Verification of Intrusive List and Tree Data Structures in Separation Logic”, 2024. `doi:10.5281/zenodo.12575047`.

11 Gérard P. Huet. The zipper. *JFP*, 7(5):549–554, 1997. `doi:10.1017/S0956796897002864`.

12 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011. `doi:10.1007/978-3-642-20398-5_4`.

13 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016. `doi:10.1145/2951913.2951943`.

14 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 28:e20, 2018. `doi:10.1017/S0956796818000151`.

15 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*, pages 637–650, 2015. `doi:10.1145/2676726.2676980`.

16 Kernel Newbies. How does the kernel implements linked lists?, 2017. URL: `https://kernelnewbies.org/FAQ/LinkedLists`.

17 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):77:1–77:30, 2018. `doi:10.1145/3236772`.

18 Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*, pages 696–723, 2017. `doi:10.1007/978-3-662-54434-1_26`.

19 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017. doi:10.1145/3009837.3009855.

20 Olaf Krzikalla and Ion Gaztanaga. Boost.intrusive, part of Boost C++ documentation, 2024. Chapter 17, version 1.84.0. URL: https://www.boost.org/doc/libs/1_84_0/doc/html/intrusive.html.

21 Keunhong Lee, Jeehoon Kang, Wonsup Yoon, Joongi Kim, and Sue Moon. Enveloping Implicit Assumptions of Intrusive Data Structures within Ownership Type System. In *PLOS*, pages 16–22, 2019. doi:10.1145/3365137.3365403.

22 Linux Kernel. Source code of red-black trees, 2021. URL: https://github.com/torvalds/linux/blob/v6.8/include/linux/rbtree_types.h#L5.

23 Linux Kernel. Source code of linked lists library, 2023. URL: https://github.com/torvalds/linux/blob/v6.8/include/linux/list.h.

24 Linux Kernel. Source code of `container_of` macro, 2023. URL: https://github.com/torvalds/linux/blob/v6.8/include/linux/container_of.h.

25 Linux Kernel. Source code of `task_struct`, 2024. URL: https://github.com/torvalds/linux/blob/v6.8/include/linux/sched.h#L748.

26 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *PLDI*, pages 809–824, 2022. doi:10.1145/3519939.3523432.

27 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*, pages 41–62, 2016. doi:10.1007/978-3-662-49122-5_2.

28 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142, pages 1–19, 2001. doi:10.1007/3-540-44802-0_1.

29 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *PACMPL*, 7(POPL):1:1–1:32, 2023. doi:10.1145/3571194.

30 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

31 Jan Smans, Bart Jacobs, and Frank Piessens. VeriFast for Java: A Tutorial. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850, pages 407–442. Springer, 2013. doi:10.1007/978-3-642-36946-9_14.

32 The Linux Kernel. What are red-black trees, and what are they for?, 2007. URL: https://docs.kernel.org/core-api/rbtree.html.