When Separation Arithmetic is Enough

Jean-Christophe Filliâtre[1(✉)], Andrei Paskevich[1], and Olivier Danvy[2]

[1] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria,
Laboratoire Méthodes Formelles, Gif-sur-Yvette 91190, France
jean-christophe.filliatre@cnrs.fr,
andrei.paskevich@universite-paris-saclay.fr
[2] School of Computing, National University of Singapore, Singapore, Singapore
danvy@acm.org

Abstract. In the practice of deductive program verification, it is desirable to make proofs as automated as possible. Today, the best tools for that are SMT solvers, which are able to handle both first-order logic and linear arithmetic. However, SMT solvers are not well suited for inductive reasoning, which is often needed to deal with recursive data structures such as linked lists or trees. In this paper, we propose a technique for specifying and proving imperative programs manipulating pointer-based recursive data structures, which stays within reach of first-order provers. The idea is to map a recursive structure onto a flat integer-indexed sequence, in such a way that separation and frame properties can be expressed using only simple arithmetic relations. We illustrate this approach with two examples: an original variant of list reversal and Morris's algorithm for constant-space traversal of a binary tree.

1 Introduction

Program verification, and deductive verification specifically, benefits enormously from proof automation, as provided by SMT solvers and their powerful combination of first-order logic and linear arithmetic. Today we get fully automated proofs for programs which would in the past require copious amounts of interactive reasoning.

Yet automated provers are far from a silver bullet. They are highly sensitive to the size and shape of their tasks, so that adding a new premise or even slightly reformulating the existing ones may cause a solver to no more being able to prove a goal that was easily proved before. They struggle with tasks that require lots of premise instantiation, especially when combined with interpreted theories like arithmetic. And, finally, they are not made for inductive reasoning.

This last limitation is especially felt when we verify pointer-based data structures like lists or trees, where well-formedness and other important properties are most naturally introduced with recursive definitions. Here we find ourselves between two seemingly conflicting principles:

This research was supported by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030".

F. Damiani and M. Farrell (Eds.): iFM 2025, LNCS 16194, pp. 23–37, 2026.
https://doi.org/10.1007/978-3-032-10794-7_2

The good specification is the one that is easy to understand.

The good invariant is the one that is easy to verify.

For our specification to be clear and convincing for those who will read it later, we would want to stay with recursive definitions. For our invariants to be within reach of SMT solvers, we would rather switch to a non-recursive formulation.

In this paper, we introduce an approach that allows us to reconcile, in certain cases, these two principles, achieving mostly automated proofs for pointer-based data structures. The key idea is to organize the elements of the manipulated data structure into a flat integer-indexed sequence, where the separation and frame properties can be expressed as simple linear inequalities, the comfort zone of SMT solvers. This sequence is only used for the intermediate stages of the proof, where it is created, as a ghost object, from the original recursive specification. The soundness of this translation is relatively easy to verify. Indeed, it amounts to a simple conversion between two mathematical models of the same data structure, done entirely in ghost code and without any modification of the actual program memory.

In what follows, we introduce and illustrate our approach through a series of examples. We chose OCaml as the implementation language, as this is the language that Why3 [9], our verification tool of choice, is based upon, making it easier for the reader to link the specification to the implementation. However, the same ideas can be applied when verifying code written in a language without algebraic data types and/or with actual pointers, like C. Source code and proofs are available at https://doi.org/10.5281/zenodo.17225346.

We start with the classic list reversal on linked lists (Sect. 2) as a way to introduce all necessary notions through an easy and well-known case. We then consider a variant of list reversal, where instead of reversing the "next" pointers, we rearrange the values in the list cells (Sect. 3). This program runs in constant space by reversing and then restoring the "next" pointers, using the list structure itself to temporarily store information. Our next example in Sect. 4 is the in-order traversal of a binary tree in constant space using Morris's algorithm [16]. Like in the previous example, Morris's algorithm temporarily mutates a data structure in order to avoid memory allocation. Finally, we cite in Sect. 5 the second challenge of the 9th VerifyThis verification competition [1]—an algorithm converting a doubly-linked list into a binary tree—which was the original inspiration for the present work. We conclude with a brief survey of related work and a discussion of the scope and limitations of the proposed method.

2 Warm Up: Classic List Reversal

To introduce our approach, let us revisit the classic list reversal. An OCaml type for mutable linked lists can be defined as follows:

```
type lst =
  | Nil
  | Cons of { mutable car: elt; mutable cdr: lst }
```

```
let rec aux r l = match l with
  | Nil    -> r
  | Cons c -> let n = c.cdr in
              c.cdr <- r;
              aux l n

let list_reversal (l: lst) : lst =
  aux Nil l
```

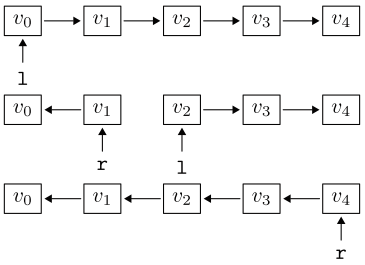


Fig. 1. Classic List Reversal.

This type is compiled in a natural and efficient way: a list is either the empty list `Nil`, implemented as a scalar, or a pointer to a heap-allocated `Cons` block with two mutable fields `car` and `cdr`. For simplicity, we consider here monomorphic lists that carry values of some fixed type `elt`.

The following diagram represents a list of five elements stored in a variable `l`. We make a slight abstraction, by showing only `car` values and representing `cdr` pointers as horizontal arrows.

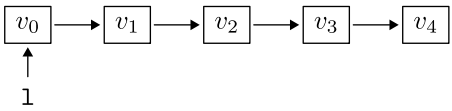


Figure 1 shows OCaml code for a function `list_reversal` that takes a list argument, performs in-place reversal, and returns the reversed list. The input list may be empty, in which case the output is also the empty list. When the input list is not empty, the returned value is the head of the reversed list, which is the last cell of the input list.

The heart of the algorithm is a recursive function `aux` with two list parameters `r` and `l`. The list `r` is the head of the already reversed prefix of the input, and the list `l` is the head of the still-to-be-reversed suffix of the input. The pictures on the right illustrate the initial state (when `aux` is called with `r` equal to `Nil`), an intermediate state, and the final state (when `l` is `Nil` and `r` is returned).

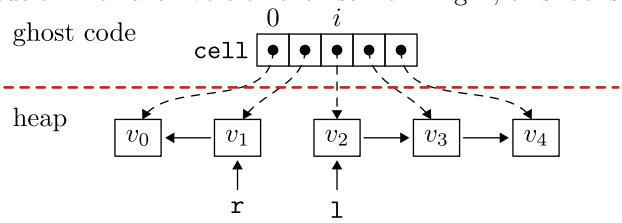
To verify the code in Fig. 1, we introduce a Why3 model of OCaml lists as follows. An abstract Why3 type `lst` models the values of OCaml lists, with a constant `nil` representing the empty list[1]. The program memory is modeled using Burstall's component-as-array principle [5]: a mutable `lst`-indexed map for each field of the data structure:

```
type mem = { mutable mcar: lst -> elt;
             mutable mcdr: lst -> lst }
```

The main idea of our "separation arithmetic" approach is to map the recursive structure of a list onto a ghost sequence of elements of type `lst`. This

[1] As `lst` is not a sum type in the Why3 model, we cannot reuse the `Nil` constructor.

sequence complements the unstructured heap model and provides the necessary support to describe intermediate structures and shapes that appear during the program execution. For the five element list from Fig. 1, this looks as follows:



On the bottom side, we have the OCaml heap and the five list cells. On the top side, we have a ghost sequence `cell` that contains, in order, the list elements. This way, the specification and the proof may conveniently use sequence indices to refer to list elements and list segments. For instance, a ghost variable i can be used to track the list cell pointed to by the program variable `l`. Then we can state that `r` is the list comprising the cells from $i - 1$ to 0, and similarly that `l` is the list comprising the cells from i to the end of `cell`.

We introduce a new Why3 type `list_shape` for this ghost sequence. It is composed of an integer-indexed function and a length:

```
type list_shape = { cell : int -> lst; size : int }
```

We equip this type with several invariants, which state that the sequence elements are non-`nil` and pairwise distinct:

```
invariant { 0 <= size }
invariant { forall i. 0 <= i <  size -> cell[i] <> nil }
invariant { forall i. 0 <= i <  size ->
            forall j. 0 <= j <= size -> i <> j -> cell[i] <> cell[j] }
```

Note that the index `j` in the last invariant goes up to `size` inclusive. In this way, `cell[size]` serves as a sentinel, which can be `nil`, for a `Nil`-terminated list, or non-`nil`, for a segment of a larger list. This is not required for the classic list reversal, but will prove handy for the algorithm in the next section.

We can now introduce predicates that relate a given list shape to the contents of the heap. One such predicate states that a list segment from `p` to `q` is composed of the list shape cells from `lo` to `hi` inclusive:

```
predicate listLR (ls: list_shape) (m: mem) (p: lst) (lo hi: int) (q: lst) =
  0 <= lo <= hi <= ls.size /\ p = ls.cell[lo] /\ q = ls.cell[hi] /\
  forall i. lo <= i < hi -> m.mcdr[ls.cell[i]] = ls.cell[i+1]
```

Notice how the rightmost element `q` is stated to be equal to the sequence element at the index `hi`. In a similar fashion, we define a predicate that describes a reversed list segment. This time we consider the sequence elements from right to left: from, but not including, the index `k` and down to 0.

```
predicate listRL (ls: list_shape) (m: mem) (p: lst) (k: int) =
  (k = 0 /\ p = nil) \/
  (0 < k <= ls.size /\ p = ls.cell[k-1] /\ m.mcdr[ls.cell[0]] = nil /\
    forall i. 0 < i < k -> m.mcdr[ls.cell[i]] = ls.cell[i-1])
```

This predicate is more complex than the previous one, because the invariants of `list_shape` do not require `cell[-1]` to be `nil`.

Lastly, we introduce a frame predicate stating that memory is not modified outside of the list shape:

```
predicate frame (ls: list_shape) (m1 m2: mem) =
  forall p. (forall i. 0 <= i < ls.size -> p <> ls.cell[i]) ->
    m1.mcar[p] = m2.mcar[p] && m1.mcdr[p] = m2.mcdr[p]
```

Here, the condition `forall i. 0 <= i < ls.size -> p <> ls.cell[i]` means that pointer `p` does not occur anywhere in the list shape `ls`. The `frame` predicate appears in the postcondition of `list_reversal`, where it is applied to the pre-state and the post-state of the program memory, ensuring that all memory changes during the reversal could only have happened inside the list shape.

The key point here is that the definitions of these predicates are not recursive. Instead, they are all defined using universal quantifiers and linear arithmetic.

Taking a list shape and the current position as ghost parameters, the contract of the `aux` function is rather straightforward:

```
let rec aux (ghost ls: list_shape) (ghost i: int) (r l: lst) : lst
  requires { listLR ls mem l i ls.size nil }
  requires { listRL ls mem r i }
  variant  { ls.size - i }
  writes   { mem.mcdr }
  ensures  { listRL ls mem result ls.size  }
  ensures  { frame  ls mem (old mem) }
```

Here, `mem` is a global variable that represents the current state of the heap. The preconditions relate `l` and `r` to the list shape using the ghost index `i`. The initial and final states are nicely captured by `i` = 0 and `i` = `ls.size`, respectively. The `writes` clause states that only `cdr` fields have been modified. The postconditions express the reversal and frame properties. Finally, the termination of `aux` is justified by the variant `ls.size - i`. The verification condition is easily discharged by SMT solvers (e.g., Z3 proves it in a fraction of a second). The main function `list_reversal`, which is merely a call to function `aux`, is also readily verified:

```
let list_reversal (ghost ls: list_shape) (p: lst) : (r: lst)
  requires { listLR ls mem p 0 ls.size nil }
  writes   { mem.mcdr }
  ensures  { listRL ls mem r ls.size }
  ensures  { frame  ls mem (old mem) }
= aux (ghost ls) (ghost 0) nil p
```

To complete the verification, we must link our easy-to-prove shape-based invariant to a final, easy-to-understand specification. To do that, we must first build a suitable list shape and prove the corresponding instance of `listLR`. We can do this with a ghost function, as follows:

```
let ghost shape_of_list (p q: lst) : (ls: list_shape)
  requires { linked_list mem p q }
  ensures  { listLR ls mem p 0 ls.size q }
```

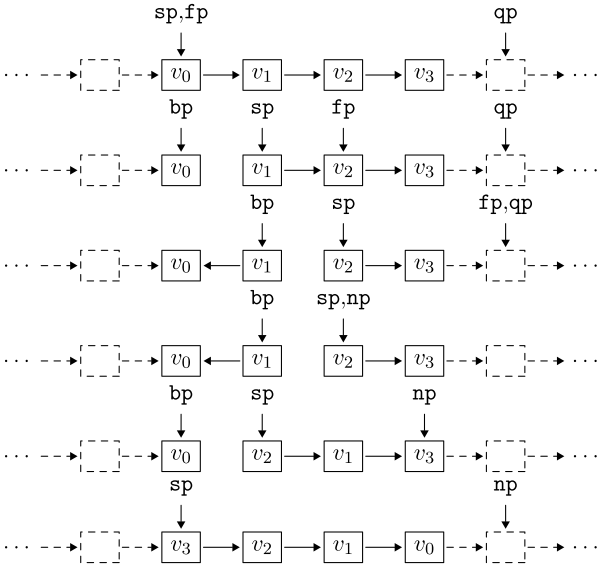


Fig. 2. Reversing values in constant space.

The precondition posits that the heap contains a list segment going from `p` to `q`. We intentionally do not provide a definition for the `linked_list` predicate in the paper. Indeed, it can be defined in any number of ways: recursively or as an inductive predicate, or as an existence of a particular mathematical model mapped to the program memory, maybe even as an existence of a suitable `list_shape`. In our implementation, we opted for a traditional recursive definition.

In the following sections, we omit the discussion of this translation between two specification styles, and concentrate on the internal proof-oriented specification.

3 Reversing Values in Constant Space

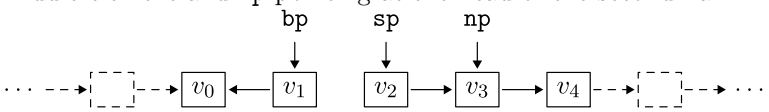
To demonstrate the benefits of our approach on a more complex example, we introduce a variant of the classic list reversal where the list is reversed by swapping the contents of the `car` fields instead of reversing the direction of the `cdr` fields. To run in constant space, the algorithm proceeds as follows. First, it uses the Tortoise and Hare algorithm (TH for short) [13, ex. 6, p. 7] [7] to reach the middle of the list, while simultaneously performing the *list reversal* of the first half. Second, it simultaneously traverses the two halves of the list, swapping the contents of the `car` fields and restoring the `cdr` fields of the first half.

```
1  let rec back_again bp sp np = match bp, np with
2    | Cons bc, Cons nc ->
3        let tmp = bc.car in bc.car <- nc.car; nc.car <- tmp;
4        let nbp = bc.cdr in bc.cdr <- sp;
5        back_again nbp bp nc.cdr
6    | _ -> ()
7
8  let rec tortoise_hare bp sp fp qp = match sp, fp with
9    | _ when fp == qp ->
10       back_again bp sp sp
11   | Cons sc, Cons {cdr = nfp} when nfp == qp ->
12       back_again bp sp sc.cdr
13   | Cons sc, Cons {cdr = Cons {cdr = nfp}} ->
14       let nsp = sc.cdr in sc.cdr <- bp;
15       tortoise_hare sp nsp nfp qp
16
17 let value_reverse (sp: lst) (qp: lst) : unit =
18   tortoise_hare Nil sp sp qp
```

Fig. 3. Reversing values in constant space.

Figure 2 illustrates the algorithm on a 4-element list. In the picture, `qp` stands for the terminator of the list, which can be either `Nil` or—if we want to reverse the values inside a list segment—the first cell after the end of the segment. The TH algorithm traverses the list at speed 1 (the tortoise) and 2 (the hare), using two pointers `sp` (slow pointer) and `fp` (fast pointer), respectively. With four elements in the list, the fast pointer reaches `qp` in two steps (third line). At this point, the slow pointer `sp` is at the first element of the second half, while the reversed first half is accessible via the "back pointer" `bp`, which lingers one step behind `sp`. Now we start the second part of the algorithm. We traverse the first half with `bp` and the second half with `np`. We swap the values at `bp` and `np`, we restore the next pointer of `bp` (line 5), and we move `bp` and `np` further in the lists. Note how `sp` is maintained so that we can restore the first half of the list.

An OCaml implementation of the algorithm is given in Fig. 3. Function `tortoise_hare` implements the first part of the algorithm (TH + list reversal). Lines 9–10 handle lists with an even number of elements (as in our example), and lines 11–12 handle lists with odd length. In the latter case, we have `sp` pointing at the middle element and `np` pointing at the head of the second half:



Function `back_again` implements the second part (swaps + list reversal). Note that all recursive calls in the two functions are tail calls, which are optimized by the OCaml compiler, ensuring that the program runs in constant space. Alternatively, the same algorithm can be easily implemented with loops.

```
let rec warp p (N q) =
  if q.right == E then (q.right <- p; true ) else
  if q.right == p then (q.right <- E; false) else
  warp p q.right

let rec morris visit (N {left; dat; right} as p) =
  if left != E && warp p left then
    morris visit left
  else (
    visit dat;
    if right != E then morris visit right )

let traversal (visit: elt -> unit) (p: tree) : unit =
  if p != E then morris visit p
```

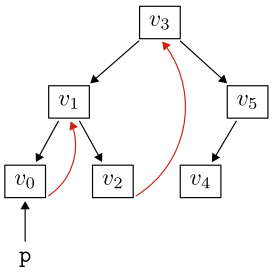


Fig. 4. Morris's algorithm.

To verify the three functions from Fig. 3, we reuse the `list_shape` type and the predicates from the previous section. For instance, the main function is given the following contract:

```
let value_reverse (ghost ls: list_shape) (sp qp: lst) : unit
  requires { listLR ls mem sp 0 ls.size qp }
  ensures  { forall i. 0 <= i < ls.size ->
             mem.mcar[ls.cell[i]] = old mem.mcar[ls.cell[ls.size-1-i]]
          /\ mem.mcdr[ls.cell[i]] = old mem.mcdr[ls.cell[i]] }
  ensures  { frame ls mem (old mem) }
```

Auxiliary functions `back_again` and `tortoise_hare` are specified in a similar way, with another ghost integer parameter to relate the pointer parameters to the sequence, as we did for the classic list reversal. Again, the verification using SMT solvers is straightforward, without need for user assistance of any kind.

4 Binary Tree Traversal Using Morris's Algorithm

Let us now go from linked lists to binary trees. Mutable binary trees can be defined in OCaml as follows:

```
type tree =
  | E
  | N of { mutable left: tree; mutable dat: elt; mutable right: tree }
```

As before, we consider that tree nodes carry values of some fixed type `elt`.

Morris's algorithm accomplishes in-order tree traversal in constant space by adding, temporarily, backward edges that cycle from the final rightmost node of each left subtree to the parent node of that subtree. An OCaml implementation of the algorithm is given in Fig. 4. It takes as parameters the binary tree to traverse and the visitor function. This function should be applied, in order, to

each element stored in the tree: first, the elements from the left subtree, then the parent node, and then the elements from the right subtree.

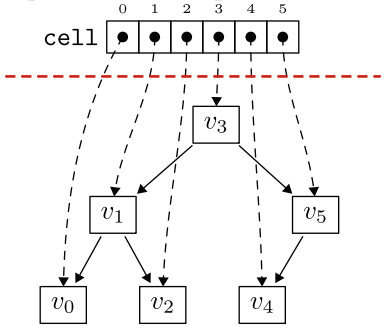
When the main function, `morris`, is applied to a non-empty node p, it checks whether the left subtree L is empty. If it is, then p can be visited right away, and the traversal continues down the right subtree. If L is non-empty, `morris` calls an auxiliary function `warp` which descends to the rightmost node of L; this node would be the final node in the in-order traversal of L. If, during the descent, `warp` reaches an empty right subtree, then it adds an edge from that node to p, creating a cycle, and returns `true` to inform `morris` that L is now "warped". If, however, `warp` reaches p, this means that L has already been warped, and we actually just finished traversing it. In this case, `warp` removes the backward edge and returns `false`, so that `morris` does not go through L for the second time.

Notice that all recursive function calls in the program are tail calls, leading to a fixed stack size during execution. Alternatively, the same algorithm can be implemented using nested loops.

In order to verify Morris's algorithm, we project, once again, the recursive structure of a tree onto a flat ghost sequence. We do it in the expected order of the traversal: the elements of a left subtree appear in the sequence before the parent node and the elements of the right subtree. In addition to the sequence itself, we want to store the structural relations between the elements of the sequence. Specifically, for each tree node in the sequence we want to know the positions of its left and right children, as well as the positions of its leftmost and rightmost descendants. This leads us to the following Why3 type:

```
type tree_shape = {
  cell : int -> tree; (* tree nodes *)
  lchd : int -> int;  (* left child *)
  rchd : int -> int;  (* right child *)
  lmdt : int -> int;  (* leftmost descendant *)
  rmdt : int -> int;  (* rightmost descendant *)
  size : int }
```

We only store non-empty tree nodes in the sequence. When a node at index i has an empty left subtree, we put i in the `lchd` and `lmdt` fields. When a node at index i has an empty right subtree, we put i in the `rchd` and `rmdt` fields. Here is an example of a tree shape for a six-node tree:



	0	1	2	3	4	5
lchd	0	0	2	1	4	4
rchd	0	2	2	5	4	5
lmdt	0	0	2	0	4	4
rmdt	0	2	2	5	4	5

The `tree_shape` type is equipped with a number of invariants:

```
  invariant { 0 <= size }
  invariant { forall i. 0 <= i < size -> cell[i] <> empty }
  invariant { forall i. 0 <= i < size ->
              forall j. 0 <= j < size -> i <> j -> cell[i] <> cell[j] }
  invariant { forall i. 0 <= i < size ->
    (0 <= lmdt[i] = lmdt[lchd[i]] <= lchd[i] <= i) /\
    (i <= rchd[i] <= rmdt[rchd[i]] = rmdt[i] < size) }
  invariant { forall i. 0 <= i < size ->
    (if lchd[i] = i then lmdt[i] = i else rmdt[lchd[i]] = i-1) /\
    (if rchd[i] = i then rmdt[i] = i else lmdt[rchd[i]] = i+1) }
```

The first three invariants state that cells are non-empty and pairwise distinct. These are identical to the invariants for `list_shape` in Sect. 2. We use a constant symbol `empty` to represent the empty tree `E`, similarly to `nil` standing for the empty list in the Why3 model. The fourth invariant states that cells are listed according to the in-order traversal. The last invariant states that there are no spurious cells: if a node i has a left subtree, then the rightmost descendant of that tree appears right before i in the sequence, and similarly for the right subtree.

The next ingredients are the heap model and the predicates that relate a tree shape to the contents of the heap:

```
type mem = { mutable mleft:  tree -> tree;
             mutable mdat:   tree -> elt;
             mutable mright: tree -> tree }

predicate wf_lhs (t: tree_shape) (m: mem) (lo hi: int) =
  forall i. 0 <= lo <= i < hi <= t.size ->
   (t.lmdt[i] = t.lchd[i] = i /\ m.mleft[t.cell[i]] = empty) \/
   (t.lchd[i] <= t.rmdt[t.lchd[i]] = i-1 /\
                                m.mleft[t.cell[i]] = t.cell[t.lchd[i]])

predicate wf_rhs (t: tree_shape) (m: mem) (lo hi: int) =
  forall i. 0 <= lo <= i < hi <= t.size ->
   (i = t.rchd[i] = t.rmdt[i] /\ m.mright[t.cell[i]] = empty) \/
   (i+1 = t.lmdt[t.rchd[i]] <= t.rchd[i] /\
                                m.mright[t.cell[i]] = t.cell[t.rchd[i]])
```

These predicates state, separately for the left-hand side and the right-hand side, the well-formedness of a tree object stored in memory, by showing that the pointer structure is reflected by the given tree shape. Once again, we can define them without recursion, using only universal quantifiers and linear arithmetic.

Using these predicates, we can write the specification of `traversal`:

```
let traversal (ghost t: tree_shape) (ghost k: int) (p: loc) : unit
  requires { trace.length = 0 }
  requires { 0 <= k <= t.size }
  requires { if t.size = 0 then p = empty
             else k < t.size /\ t.cell[k] = p /\
                  t.lmdt[k] = 0 /\ t.rmdt[k] = t.size-1 }
```

```
  requires { wf_lhs t mem 0 t.size }
  requires { wf_rhs t mem 0 t.size }
  writes   { mem.mright, trace }
  ensures  { forall q. mem.mright q = old mem.mright q }
  ensures  { trace.length = t.size }
  ensures  { forall i. 0 <= i < t.size -> trace[i] = t.cell[i] }
```

The tree shape and the index of the root node in the sequence are passed as ghost arguments. Instead of a `visit` function parameter, we store visited nodes in a global mutable sequence, called `trace`. The last two postconditions state that the final trace coincides with the node sequence in the shape. The `writes` clause says that only the right child pointers and the trace sequence are modified, and the first postcondition states that all right child pointers at the end of `traversal` are restored to their initial values.

The predicates above are suitable for the initial and the final state of the algorithm. However, we also need to describe the intermediate states, where some left subtrees in the tree are warped and cycle back to their respective parent nodes. This is expressed by the following `warped_rhs` predicate:

```
predicate warped_rhs (t: tree_shape) (m: mem) (lo hi: int) =
  forall i. 0 <= lo <= i < hi <= t.size ->
   (i = t.rchd[i] = t.rmdt[i] /\ (
    ((i < t.size-1 /\ t.lmdt[i+1] <= lo /\
                                  m.mright[t.cell[i]] = t.cell[i+1]) \/
     (i < t.size-1 /\ lo < t.lmdt[i+1] /\ m.mright[t.cell[i]] = empty) \/
     (i = t.size-1 /\ m.mright[t.cell[i]] = empty))) \/
   (i+1 = t.lmdt[t.rchd[i]] <= t.rchd[i] /\
                               m.mright[t.cell[i]] = t.cell[t.rchd[i]])
```

This extends the `wf_rhs` predicate by saying that a node at the index `i` whose right child is empty according to the tree shape (`i = t.rchd[i] = t.rmdt[i]`), may nonetheless have the `right` field pointing to the next node in the sequence: `m.mright[t.cell[i]] = t.cell[i+1]`. The node `t.cell[i+1]` is thus the parent of a warped left subtree, whose rightmost node is `t.cell[i]`. Moreover, only those left subtrees that contain index `lo` are warped: `t.lmdt[i+1] <= lo <= i`. In the precondition of `morris`, we set `lo` to the index of the current node `p`.

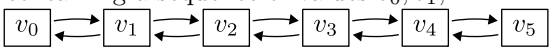
In order to verify `morris` and `warp`, we need an intermediate lemma that states that for any two subtrees, either one is contained in the other, or they are disjoint and there is at least one tree node between them in the in-order traversal. This lemma is proved separately, using a lemma-function.

We also provide a ghost function that computes a suitable tree shape from a traditional recursive description of a binary tree in the heap.

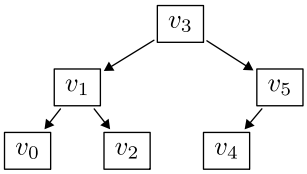
5 Bonus Example: VerifyThis 2021

The approach presented in this paper originates from a Why3 solution to the second challenge from the VerifyThis 2021 verification competition [10]. The task

was to verify an algorithm converting doubly linked lists into binary trees. We start with a list containing a sequence of values $v_0, v_1, \ldots$:

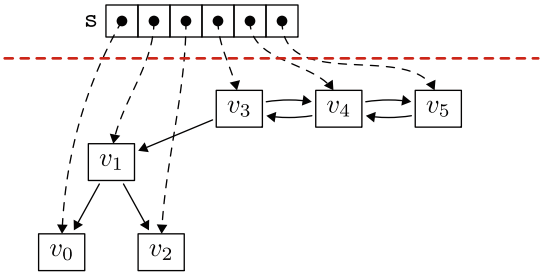


Each cell contains a pointer `prev` towards the previous cell (or `null` for the first cell) and a pointer `next` towards the next cell (or `null` for the last cell). The algorithm rearranges the `prev` and `next` pointers, creating a binary tree, where `prev` points to the left subtree and `next` to the right subtree.



The order of elements is preserved, meaning that an in-order traversal of the final tree enumerates the values $v_0, v_1, \ldots$ in the same order as in the initial list. This way, sorted lists become binary search trees. Furthermore, the result tree is balanced. The core of the algorithm is a recursive function that converts the list prefix of a given length and returns the root node of the resulting tree and the first non-consumed list cell.

In order to specify and verify the algorithm, we can again resort to separation arithmetic. As for the list reversal, we introduce a ghost sequence `s` of list cells:



Based on this sequence, we define a predicate `dll s p lo hi` which states that `p` is the first cell of a doubly-linked list formed by the cells in `s` between the indices `lo` and `hi`. We also define a predicate `tree s p lo hi` which states that `p` is the root of a binary tree formed, in order, by the cells in `s` between the indices `lo` and `hi`. Similarly to other well-formedness predicates in this paper, the `dll` predicate is non-recursive, which allows SMT solvers to easily discharge frame-related proof obligations. A full Why3 proof of the program is available online [10].

6 Related Work

The classic list reversal via pointer reversal is the introductory example in Reynolds's seminal paper introducing Separation Logic [18][2]. Since then, it has

[2] And long before that, it was a proof example in the aforementioned 1972 paper by Burstall introducing the component-as-array principle [5].

been reused as an example in most flavors and implementations of separation logic. The proof is based on a recursively-defined predicate *list* ℓ p that states the existence in the heap of a list ℓ starting at pointer p. The definition of such a predicate looks like

$$\begin{aligned} \mathit{list}\ \varepsilon\ p &\stackrel{\text{def}}{=} p = \mathit{null} \\ \mathit{list}\ (v \cdot \ell)\ p &\stackrel{\text{def}}{=} \exists q.\ p \mapsto (v, q) \star \mathit{list}\ \ell\ q \end{aligned}$$

where $\star$ denotes the separating conjunction, meaning that the head cell of the list is disjoint from the cells composing its tail. This is similar to the third invariant on our type `list_shape` (see page 4), where we state that all cells are pairwise distinct. However, the proofs in separation logic require a great deal of user interaction, via lemmas and manual opening/closing of the recursive definition above. This is particularly true in incarnations of separation logic within proof assistants, for instance in Coq [6, chapter 3] or Isabelle [20, section 5]. Even when separation logic tools are based on SMT solvers, such as Jacobs's VeriFast [12], reasoning on mutable lists still requires a serious amount of user interaction; see for instance the `doubly_linked_list.c` example in VeriFast sources.

Blanchard et al. [2] describe the verification with Frama-C/WP of the C implementation of linked lists that is part of the operating system Contiki. For the purpose of specification, an inductive predicate relates C linked lists with an algebraic list data type, in a way similar to that of the *list* predicate above. Some lemmas then require proofs by induction, unsurprisingly; those are out of reach of SMT solvers, and performed with the Rocq Prover. Section 5 of the paper compares this approach to a previous verification effort using ghost arrays, which would be closer to our work. Their conclusion leans towards ghost lists rather than ghost arrays, but mostly for technical reasons. Indeed, Frama-C does not ensure by default the separation of program data and ghost data, which incurs extra annotations and proof obligations. It is worth pointing out that both verification approaches, either with ghost lists or with ghost arrays, require some lemmas and assertions to be discharged interactively within the Rocq Prover.

The idea of using local invariants based on universal quantification rather than recursive definitions is not new, and is folklore in program verification [8]. Examples include reachability predicates (e.g., paths in graphs) and structural properties (e.g., a binary tree is a heap). The contribution of this work is using them to express separation properties.

Schorr-Waite graph marking algorithm [19] is another instance of pointer reversal algorithm, which actually predates Morris's algorithm. Morris himself tackled the verification of the algorithm with a pen-and-paper proof [17]. Later it was turned into a formal verification by Bornat [3] and then became a classic in mechanized verification [4,11,14,15]. Even if Morris's algorithm may seem to be a simple instance of Schorr-Waite algorithm, the two are quite different, as Schorr-Waite requires extra space within nodes to store information needed by the algorithm, whereas Morris's algorithm only uses the existing left and right

pointers. For our approach, Schorr-Waite presents an additional challenge since the traversal sequence is not apparent in the initial graph. In order to construct an appropriate shape structure, we would have to simulate the same traversal in ghost code, before starting the actual computation, which would then handle the elements of the graph exactly as they are arranged in the shape.

7 Conclusion

Above, we described a method of verifying imperative programs which manipulate pointer-based recursive data structures. This technique eschews recursive specifications and relies instead on flat integer-indexed sequences to provide a logical model of the data structure. Then separation and frame properties that arise during the proof can be stated using simple arithmetic and discharged automatically by an SMT solver. Such sequences can be derived from a traditional recursively-defined model via auxiliary ghost functions, allowing us to preserve the user-facing specification and to use separation arithmetic only for proof.

To take advantage of this approach, one has to come up with a suitable flat model for the manipulated data structure. We expect this to be straightforward for lists and trees, as demonstrated by the examples in the paper. The task is more difficult for dags and graphs, where multiple paths can lead to the same cell. Our next challenge, therefore, is to show how to apply separation arithmetic in these cases, starting with the classical Schorr-Waite algorithm.

References

1. The VerifyThis program verification competition (2011). https://www.pm.inf.ethz.ch/research/verifythis.html
2. Blanchard, A., Kosmatov, N., Loulergue, F.: Logic against ghosts: comparison of two proof approaches for a list module. In: Proceedings of the 34th ACM/SIGAPP Symposium On Applied Computing (SAC 2019), pp. 2186–2195. ACM (2019)
3. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) Mathematics of Program Construction, pp. 102–126. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/10722010_8
4. Bubel, R.: The Schorr-Waite algorithm. Verification of Object-Oriented Software. The KeY Approach, pp. 569–587 (2007)
5. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. Mach. Intell. **7**(3), 23–50 (1972)
6. Charguéraud, A.: Separation Logic Foundations, Software Foundations, vol. 6. Electronic textbook (2025). https://softwarefoundations.cis.upenn.edu
7. Danvy, O.: The tortoise and the hare algorithm for finite lists, compositionally. ACM Trans. Program. Lang. Syst. **45**(1) (2023). https://doi.org/10.1145/3564619
8. Filliâtre, J.C.: Simpler proofs with decentralized invariants. J. Logical Algebraic Methods Program. **121** (2021). https://doi.org/10.1016/j.jlamp.2021.100645, https://usr.lmf.cnrs.fr/~jcf/spdi/
9. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. vol. 7792, pp. 125–128 (2013)

10. Filliâtre, J.C., Paskevich, A.: Solution for Challenge 2 of the 9th VerifyThis program verification competition (2021). https://toccata.gitlabpages.inria.fr/toccata/gallery/verifythis_2021_dll_to_bst.en.html
11. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), pp. 190–199 (2005). https://doi.org/10.1109/SEFM.2005.1
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
13. Knuth, D.E.: The Art of Computer Programming, volume 2 (3rd ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc. (1997). https://doi.org/10.5555/270146
14. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
15. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 121–135. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45085-6_10
16. Morris, J.M.: Traversing binary trees simply and cheaply. Inf. Process. Lett. **9**(5), 197–200 (1979). http://dblp.uni-trier.de/db/journals/ipl/ipl9.html#Morris79a
17. Morris, J.M.: A proof of the Schorr-Waite algorithm. In: Broy, M., Schmidt, G. (eds) Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School, directed by FL Bauer, EW Dijkstra and CAR Hoare, pp. 43–51. Springer (1982). https://doi.org/10.1007/978-94-009-7893-5_5
18. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, p. 55–74. LICS 2002, IEEE Computer Society, USA (2002)
19. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM **10**(8), 501–506 (1967). https://doi.org/10.1145/363534.363554
20. Weber, T.: Towards mechanized program verification with separation logic. In: Marcinkowski, J., Tarlecki, A. (eds.) Comput. Sci. Logic, pp. 250–264. Springer, Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30124-0_21