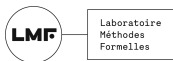


When Separation Arithmetic is Enough

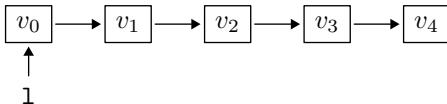
Jean-Christophe Filiâtre
joint work with Andrei Paskevich and Olivier Danvy

ANTIQUÉ Seminar
Paris, May 2026



- ① warm up: classic list reversal
- ② reversing values in constant space
- ③ binary tree traversal using Morris's algorithm
- ④ other examples
- ⑤ discussion

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



```

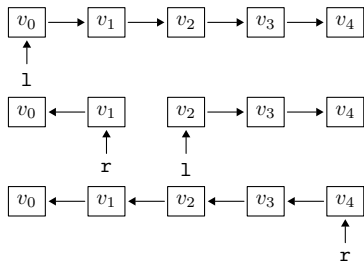
let rec aux r l = match l with
| Nil      -> r
| Cons c -> let n = c.cdr in
              c.cdr <- r;
              aux l n

```

```

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

```



```
type elt
type lst

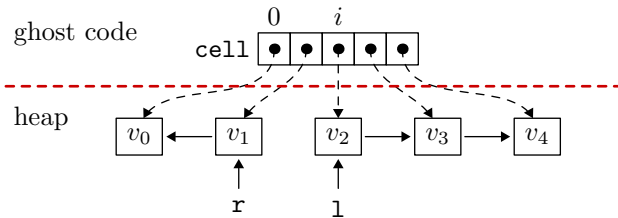
constant nil: lst

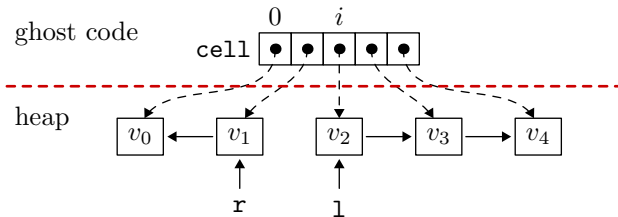
type mem = { mutable mcar: lst -> elt;
             mutable mcdr: lst -> lst }

val mem: mem
```

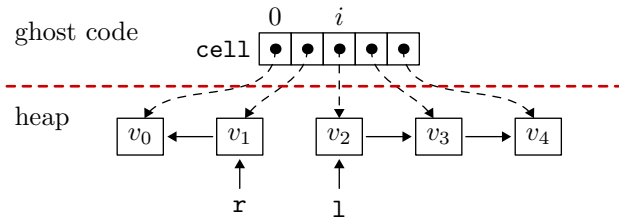
this is Burstall's **component-as-array** principle

R. M. Burstall, Some techniques for proving correctness of programs which alter data structures. Machine intelligence 1972





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



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

```
let list_reversal (ghost ls: list_shape)
  (l: lst) : (r: lst)
= ...
```

the cells are non-nil and pairwise distinct:

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

they relate a given list shape to the contents of the heap

e.g. there is a list from p to q corresponding to `cell[lo..hi]`

```

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]

```

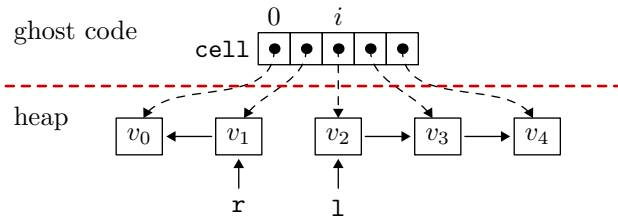
the key: **this is not recursive!**

similarly for `listRL`

```

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 }
  ensures  { listRL ls mem result ls.size }

```



the verification is fully automatic

no lemma, no assertion in the code, no user interaction

if you really insist in using a traditional specification involving a recursive predicate (say, over algebraic logical lists),

it is possible to prove the equivalence between this and the combination `list_shape + listLR`

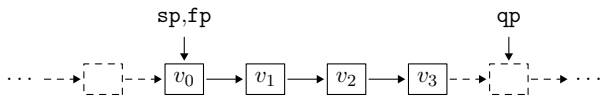
(we have done this in our Why3 proofs)

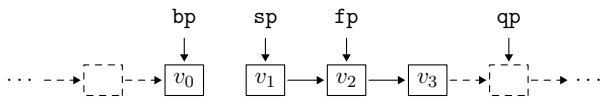
reversing values in constant space

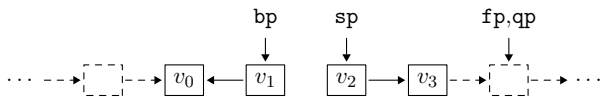
what if we want to reverse a list **by swapping the car fields** instead of reversing the cdr fields?

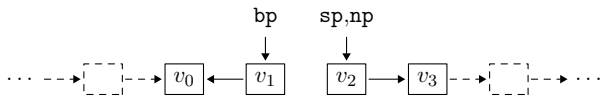
can we do this in linear time and constant space?

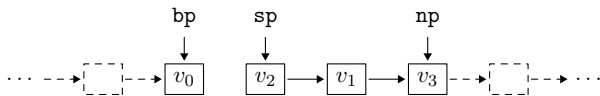
- 1 traverse the list at speed 1 and 2 (tortoise and hare) to find its middle
 - reverse the first half as you go
- 2 traverse the two halves simultaneously, swapping the cars
 - restore the first half as you go

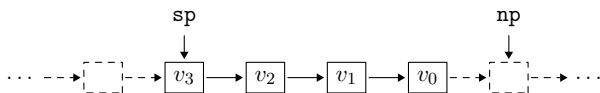


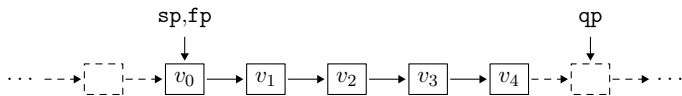


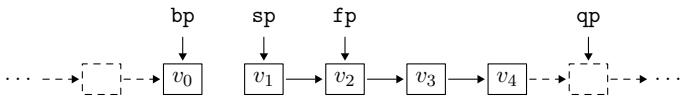


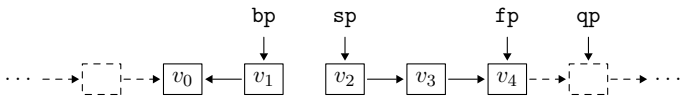


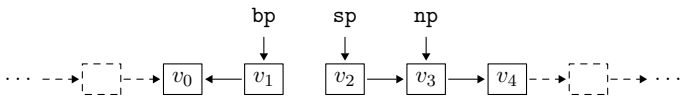


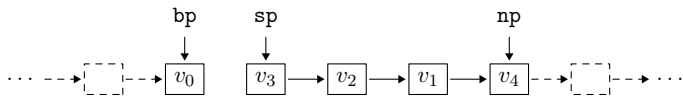


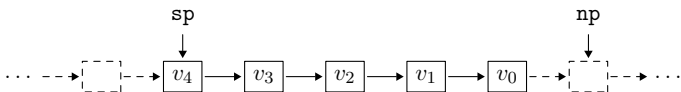




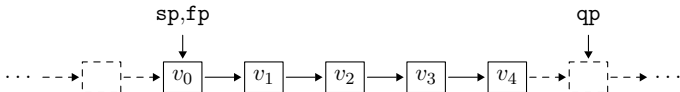




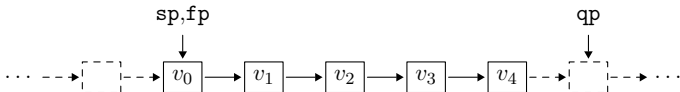




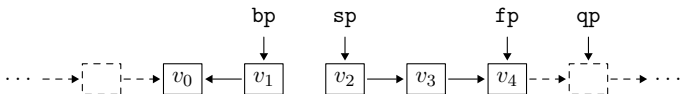
```
let value_reverse (sp: lst) (qp: lst) : unit =  
  tortoise_hare Nil sp sp qp
```



```
let value_reverse (sp: lst) (qp: lst) : unit =
  tortoise_hare Nil sp sp qp
```



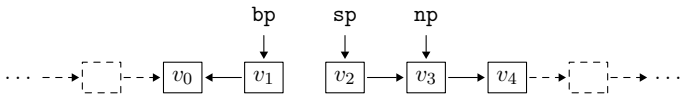
```
let rec tortoise_hare bp sp fp qp = match sp, fp with
| _ when fp == qp ->
  back_again bp sp sp
| Cons sc, Cons {cdr = nfp} when nfp == qp ->
  back_again bp sp sc.cdr
| Cons sc, Cons {cdr = Cons {cdr = nfp}} ->
  let nsp = sc.cdr in sc.cdr <- bp;
  tortoise_hare sp nsp nfp qp
```



```

let rec back_again bp sp np = match bp, np with
| Cons bc, Cons nc ->
    let tmp = bc.car in
    bc.car <- nc.car; nc.car <- tmp;
    let nbp = bc.cdr in bc.cdr <- sp;
    back_again nbp bp nc.cdr
| _ -> ()

```



(the name echoes *There and Back Again*, Danvy&Goldberg 2005)

time complexity is linear (two traversals)

all calls are tail calls, so the space complexity is constant
(could be written with loops)

we reuse the same idea, and the very same `list_shape`

```
let value_reverse (ghost ls: list_shape) (sp qp: lst)
  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]] }
```

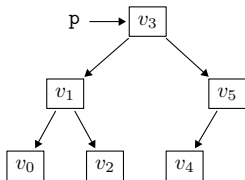
again, the verification is fully automatic

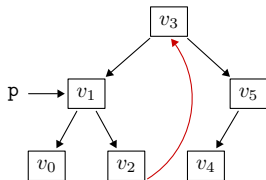
binary tree traversal using Morris's algorithm

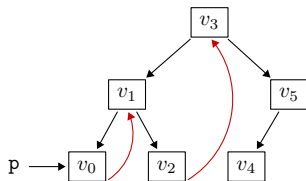
perform an in-order traversal of a binary tree **in constant space and linear time**

solution: mutate the tree as you go, to remember what to do next, and restore it before completion

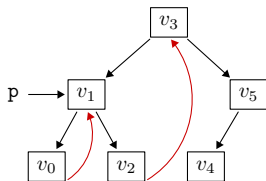
Joseph M. Morris, Traversing Binary Trees Simply and Cheaply.
Information Processing Letters 9(5), 197–200 (1979)



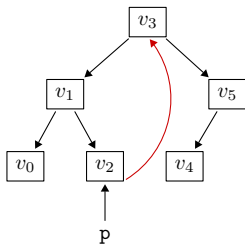




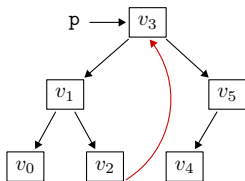
visit v_0



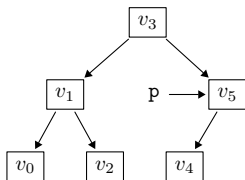
visit v_0



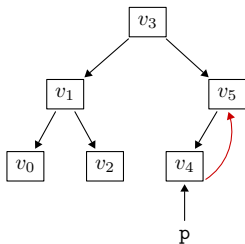
visit v_0 , visit v_1



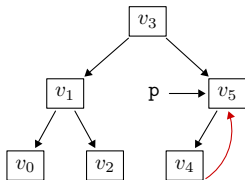
visit v_0 , visit v_1 , visit v_2



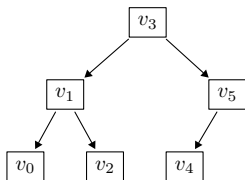
visit v_0 , visit v_1 , visit v_2 , visit v_3



visit v_0 , visit v_1 , visit v_2 , visit v_3



visit v_0 , visit v_1 , visit v_2 , visit v_3 , visit v_4



visit v_0 , visit v_1 , visit v_2 , visit v_3 , visit v_4 , visit v_5

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

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

```
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
```

```
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
```

the space complexity is **constant** (all calls are tail calls)

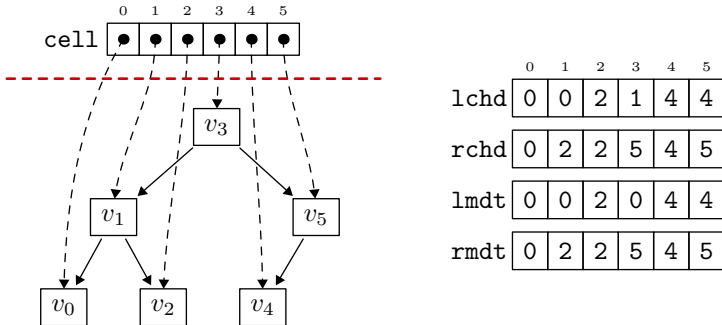
the time complexity is **linear!** (exercise: show it)

let us verify this with Why3, using separation arithmetic

let us verify this with Why3, using separation arithmetic

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

let us verify this with Why3, using separation arithmetic



cells are non-empty and pairwise distinct

```
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] }
```

(as for list_shape)

cells are listed according to the in-order traversal

invariant

```
{ forall i. 0 <= i < size ->  
  0 <= lmdt[i] = lmdt[lchd[i]] <= lchd[i] <= i  
  /\ i <= rchd[i] <= rmdt[rchd[i]] = rmdt[i] < size }
```

finally, there are no spurious cells

```
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) }
```

as for lists, we model the heap

```
type mem = ...
```

and we related the heap and the tree shape with predicates

```
predicate wf_lhs (t: tree_shape) (m: mem) (lo hi: int)
  = ...
predicate wf_rhs (t: tree_shape) (m: mem) (lo hi: int)
  = ...
predicate warped (t: tree_shape) (m: mem) (lo hi: int)
  = ...
```

no recursion, only universal quantifiers and linear arithmetic

visited nodes are appended to a ghost trace

and we show that it coincides with cell

```
let traversal (ghost t: tree_shape)
              (ghost k: int) (p: loc) : unit
...
ensures { trace.length = t.size }
ensures { forall i. 0 <= i < t.size ->
          trace[i] = t.cell[i] }
```

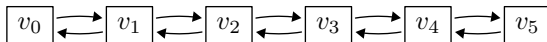
this time, the verification is slightly more complex

- need for one lemma
- a little bit of user interaction

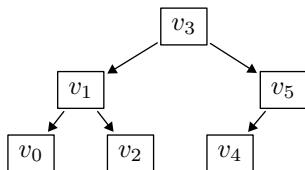
yet mostly automatic

other examples

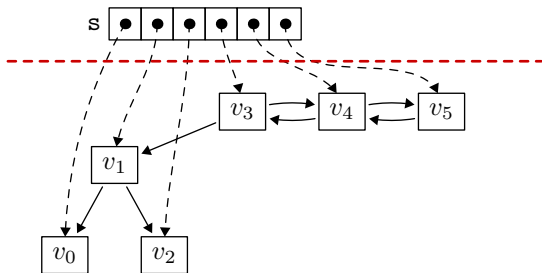
verify a program that turns a doubly-linked list



into a balanced binary tree



(prev and next pointers become left and right pointers)



we believe that separation arithmetic could also be used on

- VerifyThis 2012, challenge 3: remove the node with the minimal key in a binary search tree
- Schorr-Waite in-place graph-marking algorithm

yet to be done, though

discussion

of course, Separation Logic is a great tool to reason about pointer-based data structures

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

separation arithmetic cannot be compared, but is a powerful tool
when applicable

the idea of using local invariants based on universal quantification rather than recursive definitions is folklore

e.g.

JCF, Simpler Proofs with Decentralized Invariants.

Journal of Logical and Algebraic Methods in Programming, 2021

<https://usr.lmf.cnrs.fr/~jcf/spdi/>

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

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

thank you!



<https://usr.lmf.cnrs.fr/~jcf/sep-arith/>