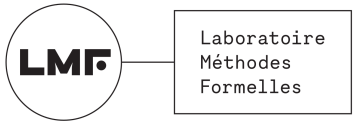
L'arithmétique de séparation

Jean-Christophe Filliâtre et Andrei Paskevich

JFLA 2023

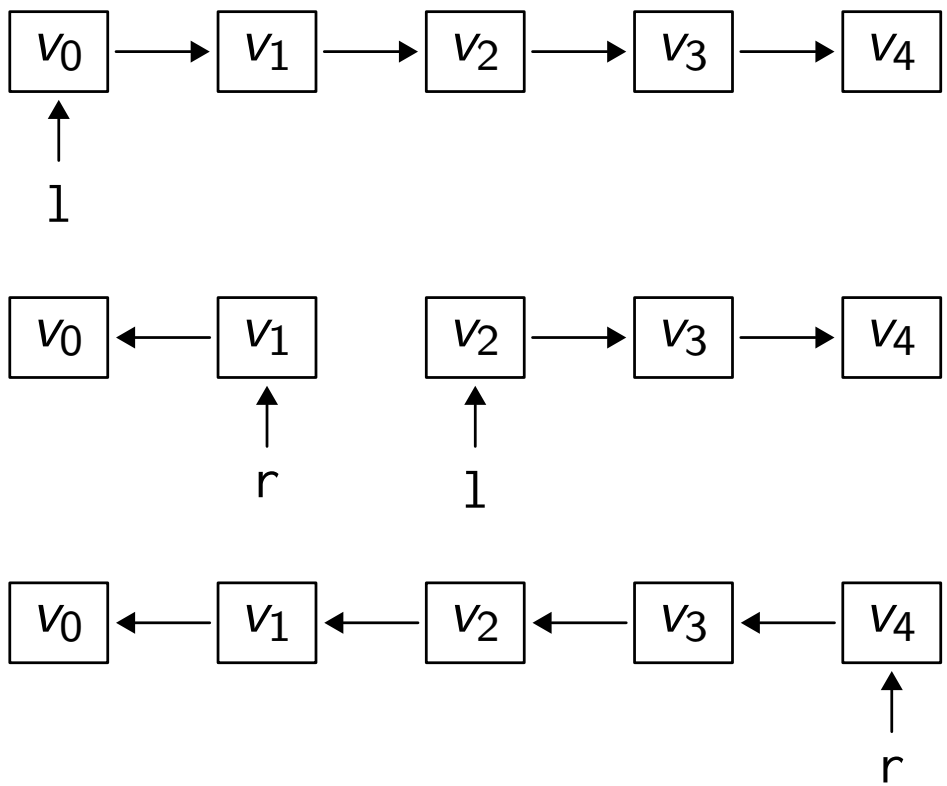
1er février 2023



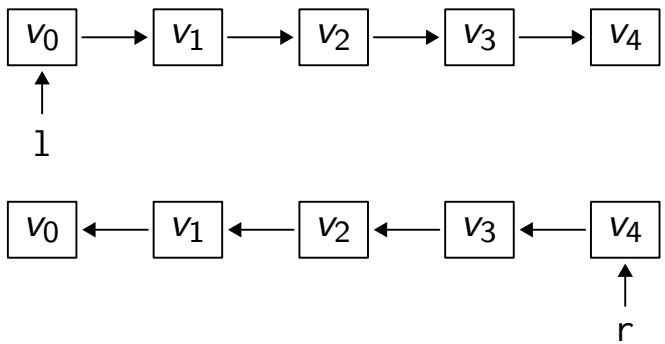
La bonne spécification, c'est celle qui est facile à comprendre.

Le bon invariant, c'est celui qui est facile à prouver.

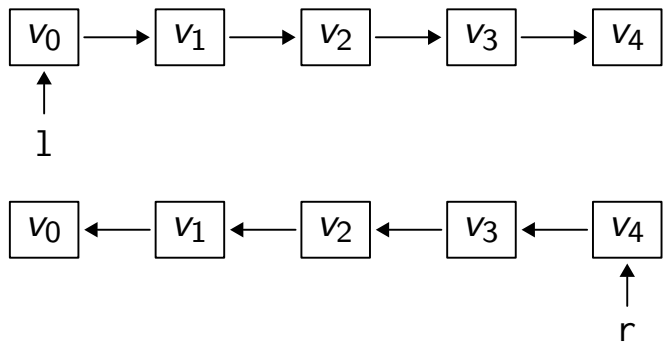
```
typedef struct Cell list;
struct Cell { int value; list *next; };

list *list_reversal(list *l) {
  list *r = NULL;
  while (l != NULL) {
    list *tmp = l;
    l = l->next;
    tmp->next = r;
    r = tmp;
  }
  return r;
}
```



```
type loc
type mem = loc -> { next: loc }

list_reversal l (ghost s) : r
  requires  list mem l s
  modifies  mem
  ensures   list mem r (rev s)
  ensures   forall p. not (mem p s) -> mem[p] = old mem[p]
```

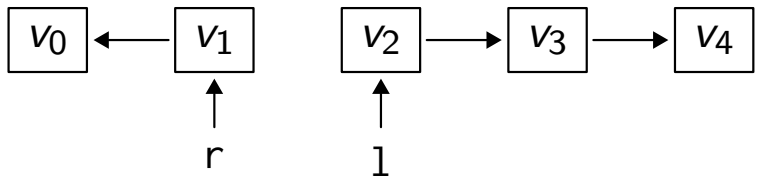


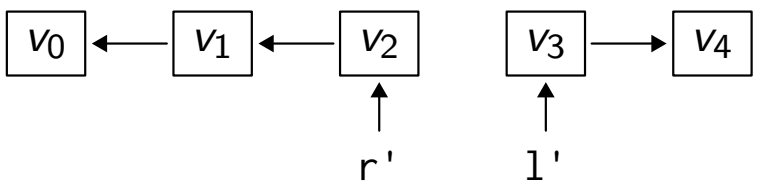
```
type loc
type mem = loc -> { next: loc }

list_reversal l (ghost s) : r
  requires  list mem l s
  modifies  mem
  ensures   list mem r (rev s)
  ensures   forall p. not (mem p s) -> mem[p] = old mem[p]

predicate list mem l s =
  if length s = 0 then l = NULL
  else l = s[0] <> NULL && list mem[l].next s[1:]
```



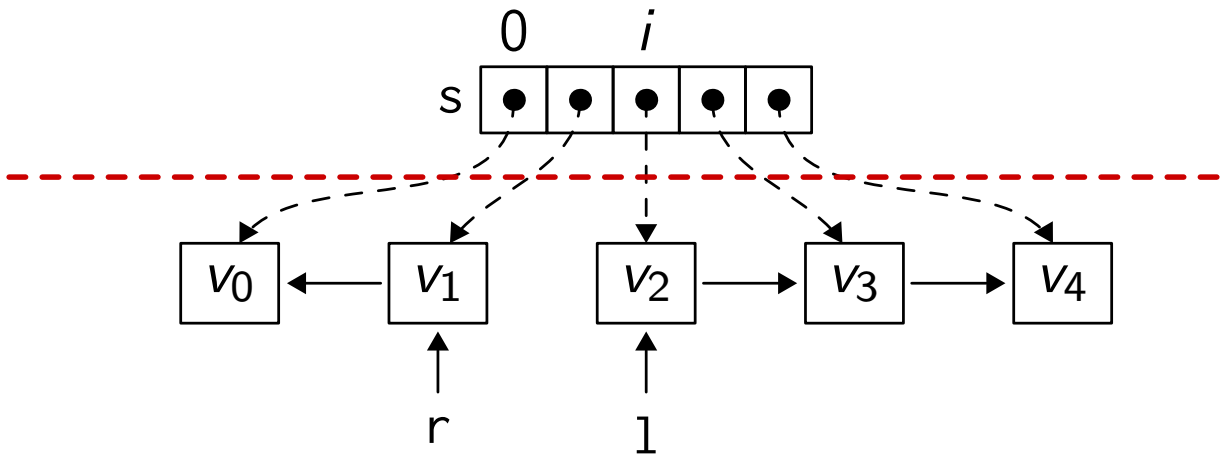
droit dans le mur

```
invariant  list mem l s[i:]
invariant  list mem r (rev s[:i])
```






```
goal  list mem' l' s[i+1:]
goal  list mem' r' (rev s[:i+1])
```




```
predicate valid_cells s =
  s contient des pointeurs non nuls et distincts

predicate suffix mem s l i =
  0 <= i <= length s &&
  if i = length s then l = null else
    l = s[i] && mem[s[length s - 1]].next = null &&
    forall k. i < k < length s -> mem[s[k-1]].next = s[k]

predicate prefix mem s r i = ...
```

nouveau contrat

```
list_reversal l (ghost s) : r
  requires valid_cells s
  requires suffix mem s l 0
  modifies mem
  ensures  prefix mem s r (length s)
  ensures  forall p. not (mem p s) -> mem[p] = old mem[p]
```

nouveaux invariants

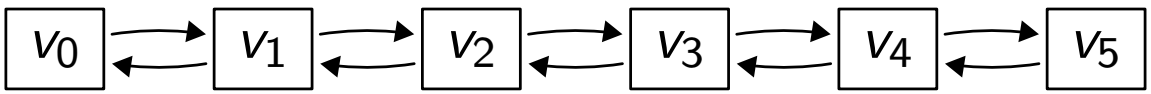
```
invariant suffix mem s l i
invariant prefix mem s r i
```

preuve maintenant entièrement automatique

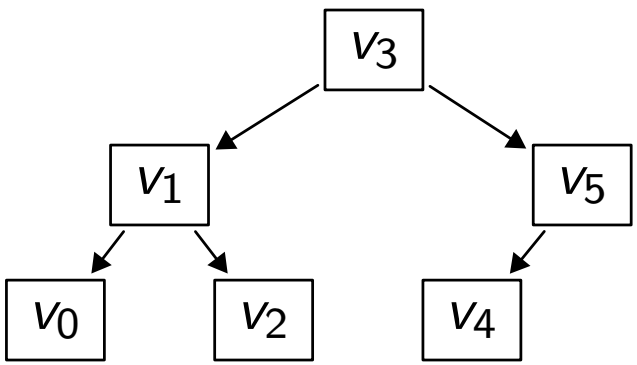
```
list_reversal_final l (ghost s) : r
  requires  list mem l s
  modifies  mem
  ensures   list mem r (rev s)
  ensures   forall p. not (mem p s) -> mem[p] = old mem[p]
= cells_of_list l s;
  let r = list_reversal l s in
  list_of_cells r s;
  r
```

```
list_reversal_final l (ghost s) : r
  requires  list mem l s
  modifies  mem
  ensures   list mem r (rev s)
  ensures   forall p. not (mem p s) -> mem[p] = old mem[p]
= cells_of_list l s;
  let r = list_reversal l s in
  list_of_cells r s;
  r

lemma cells_of_list l s
  requires list mem l s
  ensures  valid_cells s
  ensures  suffix mem s l 0
  variant  length s
= if s <> empty then cells_of_list mem[l].next s[1:]
```

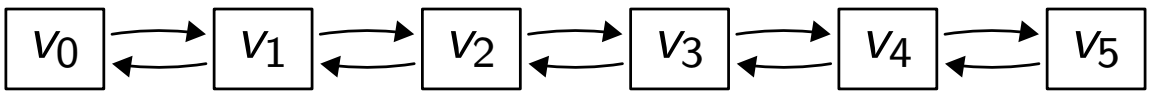
partant d'une liste
doublement chaînée



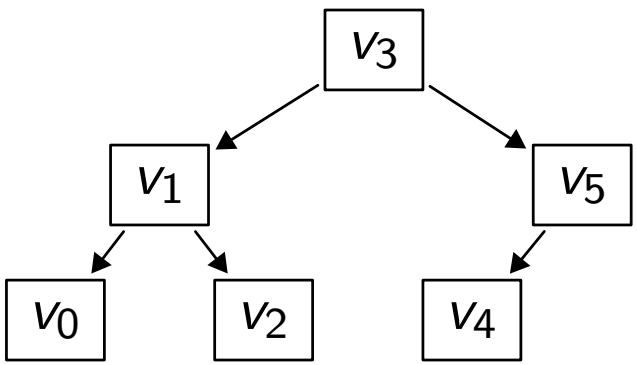
on veut construire un arbre
binaire équilibré



partant d'une liste
doublement chaînée



on veut construire un arbre
binaire équilibré



l'arithmétique de séparation
s'applique là aussi