Imperative List Reverse in Separation Logic

Andrew W. Appel

Verified
Software
Toolchain

Software Foundations
Volume 5: Verifiable C

These slides illustrate
the `reverse.c` program
and its verification in `Verif_reverse.v`

Separation Logic

If state satisfies the precondition

then it's safe to run the command

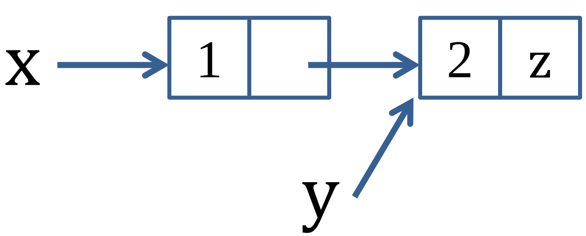
and the state after will satisfy the postcondition

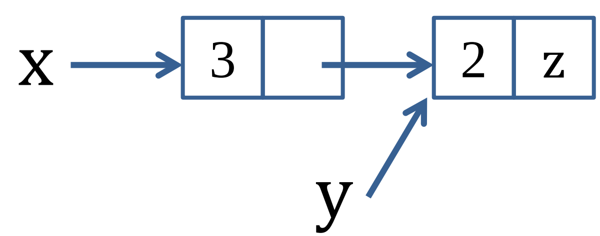
$$\{\text{Pre}\}\ \ \text{command}\ \ \{\text{Post}\}$$

separating conjunction

$$P * Q \qquad\qquad P' * Q$$

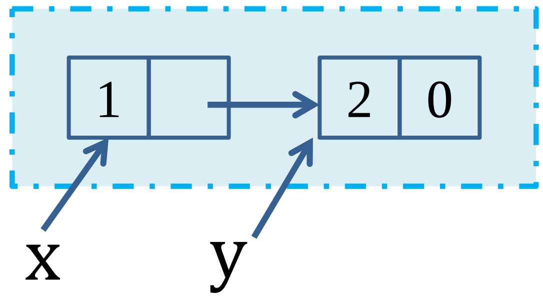
$$\{\ x \mapsto (1,y) * y \mapsto (2,z)\ \}\quad x.data=3;\quad \{\ x \mapsto (3,y) * y \mapsto (2,z)\ \}$$



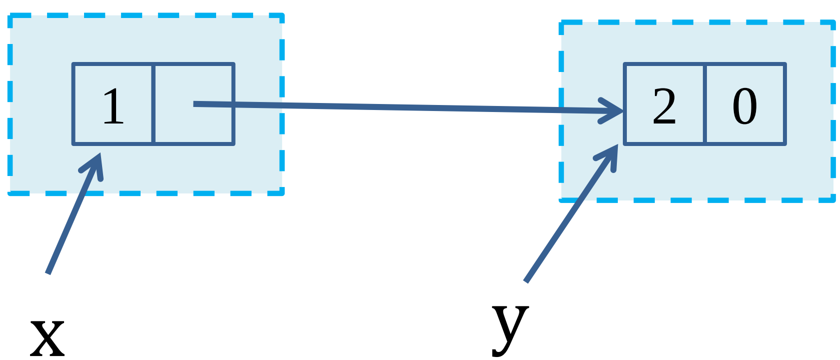


Heaplets in Separation Logic

$x \mapsto (1, y) * y \mapsto (2, NULL)$

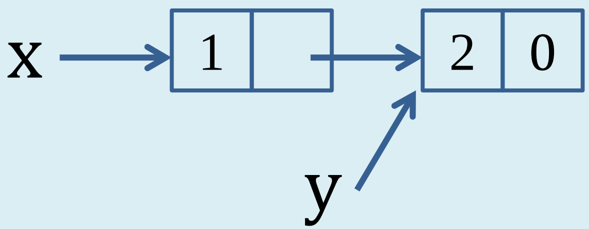


$x \mapsto (1, y) \quad * \quad y \mapsto (2, NULL)$

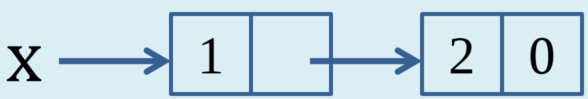


Quantifiers in Separation Logic

$$x \mapsto (1, y) * y \mapsto (2, \mathrm{NULL})$$



$$\exists y.\ x \mapsto (1, y) * y \mapsto (2, \mathrm{NULL})$$

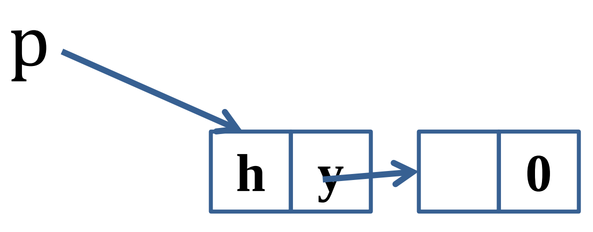


Description of linked lists in sep.log.

```
Fixpoint listrep (σ: list val) (p: val) : mpred :=
 match σ with
 | h :: σ′  ⇒  EX y, data_at ⊤ t_struct_list (h,y) p ∗ listrep σ′ y
 | nil ⇒  !! (p = null) && emp
 end.
```

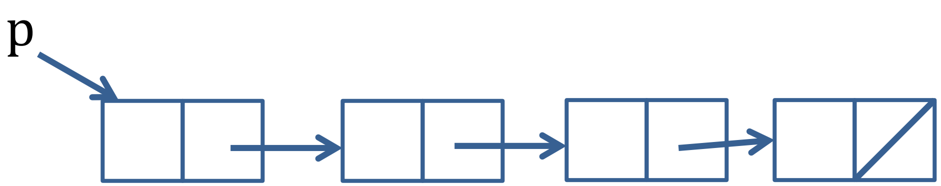
$$p \rightsquigarrow^{\sigma} \;=\; p=0 \land \mathrm{emp} \;\lor\; \exists h,\sigma',y.\ \sigma=h::\sigma' \land p\mapsto(h,y) * y \rightsquigarrow^{\sigma'}$$

$p=\mathbf{0}$



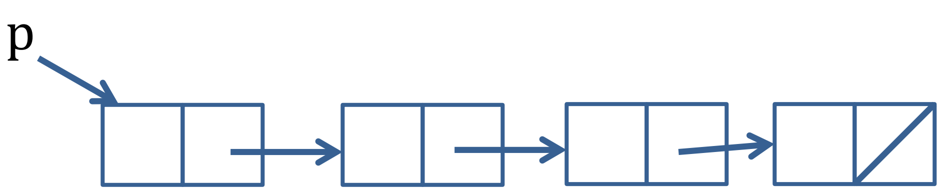
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



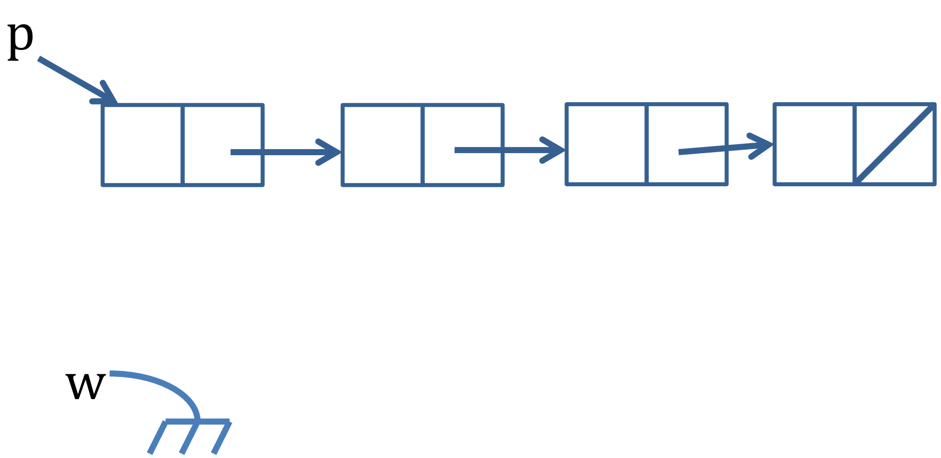
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



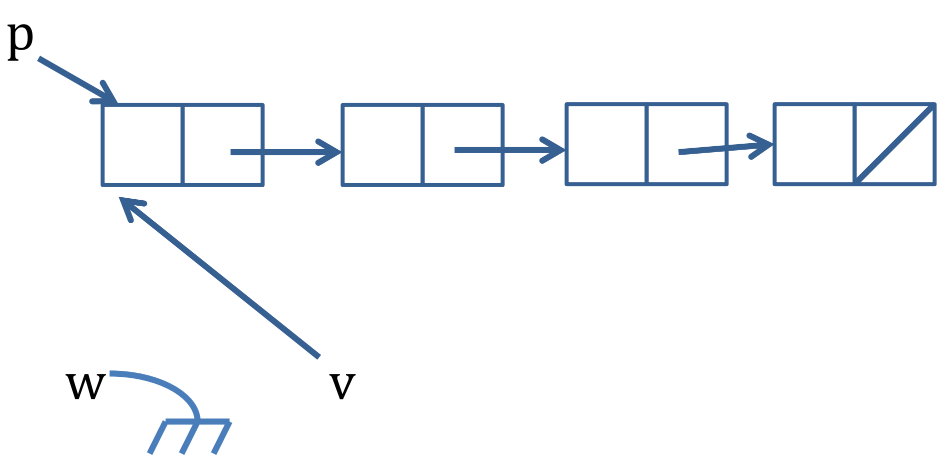
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



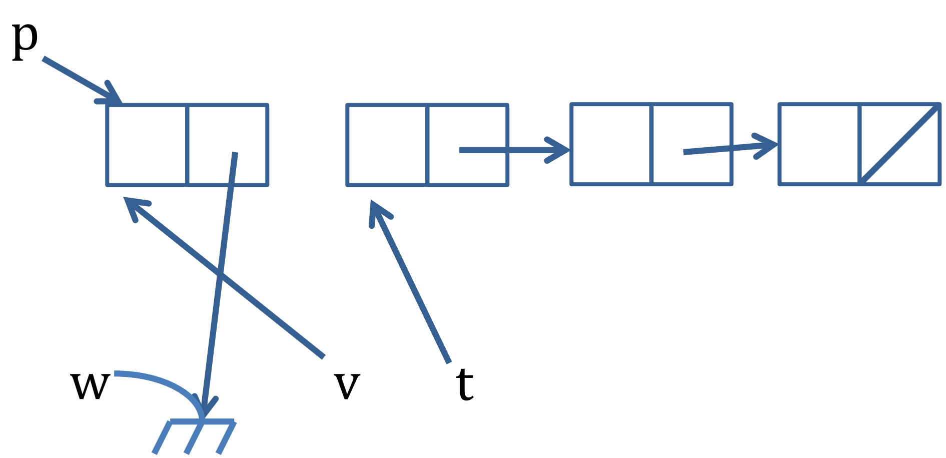
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



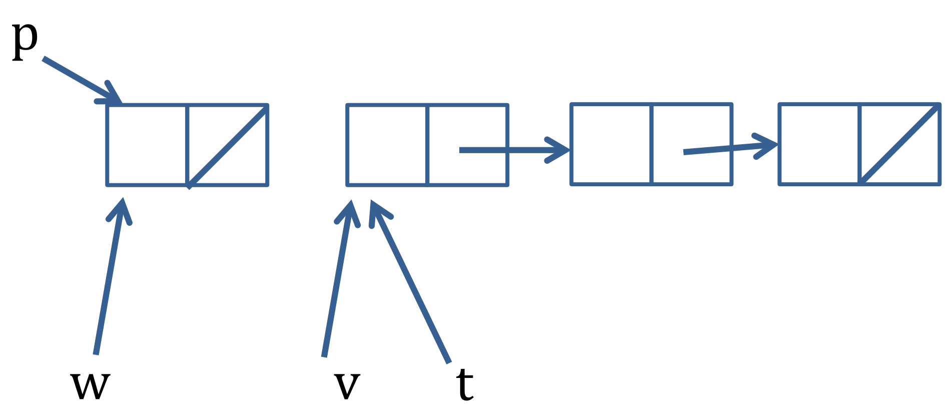
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



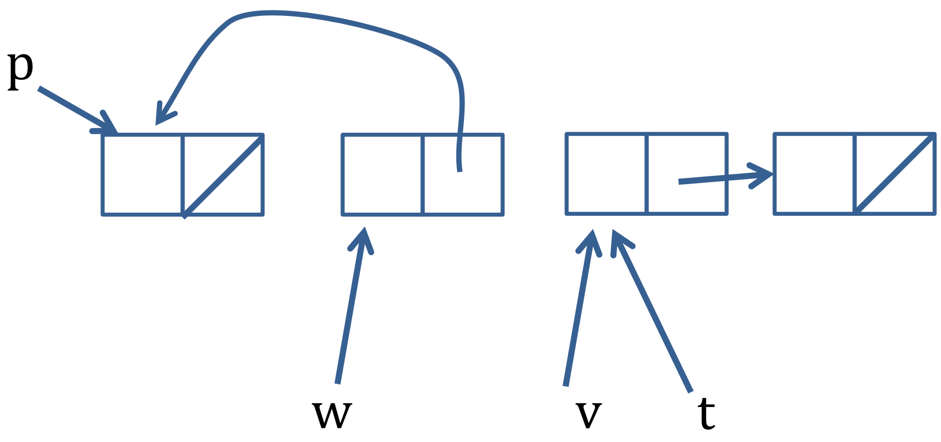
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



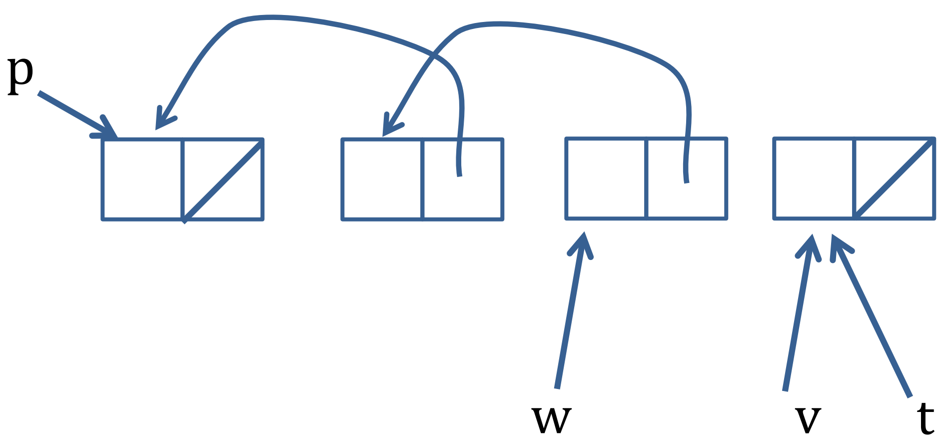
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



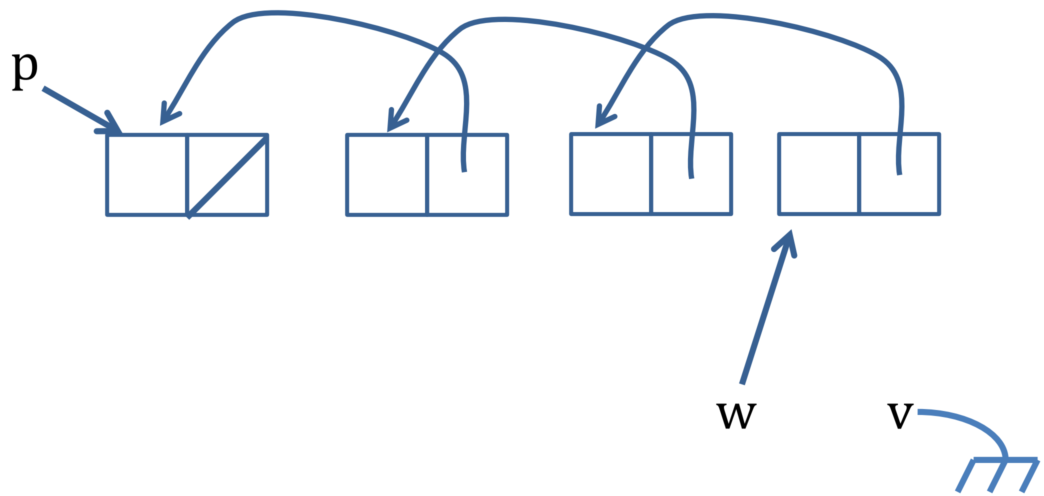
Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



Example: imperative list reverse

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



Specification and proof

$$p \rightsquigarrow^{\sigma} \quad = \quad p{=}0 \wedge \mathrm{emp} \;\; \vee \;\; \exists h,\sigma',y.\; \sigma{=}h{::}\sigma' \wedge p{\mapsto}(h,y) * y \rightsquigarrow^{\sigma'}$$

```
struct list *reverse (struct list *p);
```

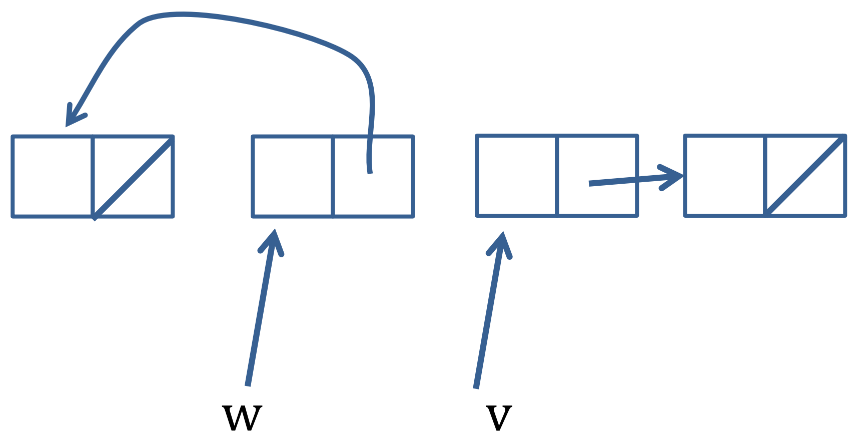
$\{\, p \rightsquigarrow^{\sigma} \,\}$ **`ret_val = reverse(p);`** $\{\, \mathrm{ret\_val} \rightsquigarrow^{\mathrm{rev}\ \sigma} \,\}$

While loops

$$\frac{P \vdash I \qquad \{I \wedge e\}\ c\ \{I\} \qquad I \wedge \neg e\ \vdash\ Q}{\{\ P\ \}\ \text{while}\ e\ \text{do}\ c\ \{\ Q\ \}}$$

Loop invariant

```
struct list {int head; struct list *tail;};

struct list *reverse (struct list *p) {
  struct list *w, *t, *v;
  w = NULL;
  v = p;
  while (v) {
    t = v→tail;
    v→tail = w;
    w = v;
    v = t;
  }
  return w;
}
```



$$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \ \wedge\ \mathrm{w} \rightsquigarrow^{\sigma 1} \ * \ \mathrm{v} \rightsquigarrow^{\sigma 2}$$

$\mathrm{rev}(1 \cdot 2 \cdot 3 \cdot 4) = 4 \cdot 3 \cdot 2 \cdot 1$

In Coq

$\{\, p \rightsquigarrow^{\sigma} \,\}$ **`ret_val = reverse(p);`** $\{\, \mathrm{ret\_val} \rightsquigarrow^{\mathrm{rev}\ \sigma} \,\}$

```
Definition reverse_spec :=
 DECLARE _reverse
  WITH sigma: list val, p: val
  PRE  [ (tptr t_struct_list) ]
     PROP ( )  PARAMS (p) SEP (listrep sigma p)
  POST [ (tptr t_struct_list) ]
    EX q:val,  PROP () RETURN(q) SEP (listrep(rev sigma) q).
```

$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1)\cdot\sigma_2\ \wedge\ w \rightsquigarrow^{\sigma 1}\ *\ v \rightsquigarrow^{\sigma 2}$

```
EX s1: list val, EX s2 : list val,  EX w: val, EX v: val,
    PROP (sigma= rev s1 ++ s2)
    LOCAL (temp _w w; temp _v v)
    SEP (listrep s1 w; listrep s2 v)
```