Foundations of Software Verification

John Wickerson

30 November 2023

Today's lecture

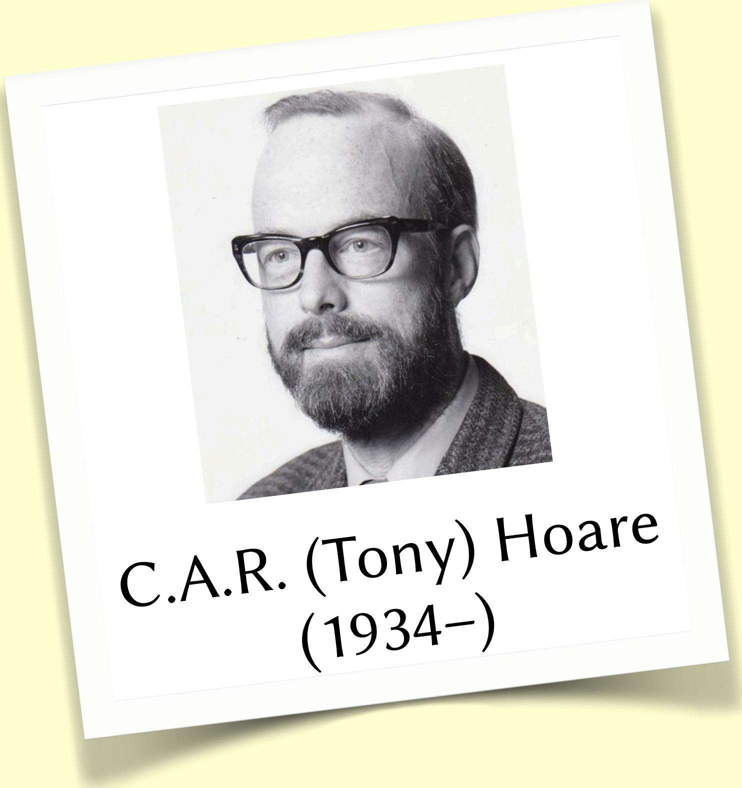
- Some underlying principles of software verification.

- A brief history of the field from around 1969 (Hoare Logic) to around 2001 (Separation Logic).

Hoare Logic

- Invented by Tony Hoare in 1969.

- A mathematical system based on annotating program code with **assertions** that must hold whenever execution reaches that point.



C.A.R. (Tony) Hoare
(1934–)

- The basic unit is the Hoare triple, written { p } C { q }.

Hoare triples

- What does { p } C { q } mean?
  - It's more-or-less equivalent to the following Dafny code:

    ```
    assume p;
    C;
    assert q;
    ```

  - If C begins execution in a state satisfying p, then any final state it reaches will satisfy q.
  - C can be an entire program, or just a single instruction.

Rules of Hoare logic

$$\frac{\{\,p\,\}\ C_1\ \{\,q\,\} \qquad \{\,q\,\}\ C_2\ \{\,r\,\}}{\{\,p\,\}\ C_1\ ;\ C_2\ \{\,r\,\}}\ \text{SEQ}$$

$$\frac{\{\,p \wedge b\,\}\ C_1\ \{\,q\,\} \qquad \{\,p \wedge \neg b\,\}\ C_2\ \{\,q\,\}}{\{\,p\,\}\ \text{if } b \text{ then } C_1 \text{ else } C_2\ \{\,q\,\}}\ \text{IF}$$

$$\frac{p \Rightarrow q[e/x]}{\{\,p\,\}\ x := e\ \{\,q\,\}}\ \text{ASS}$$

DEMO

$$\frac{p \Rightarrow I \qquad \{\,I \wedge b\,\}\ C\ \{\,I\,\} \qquad I \wedge \neg b \Rightarrow q}{\{\,p\,\}\ \text{while } b \text{ do } C\ \{\,q\,\}}\ \text{WHILE}$$

$$
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{c}\{\ \mathrm{true}\ \}\\ x := 0\\ \{x=0\}\end{array}}\;\text{ASS}
\quad
\dfrac{}{\begin{array}{c}\{x=0\}\\ y := 0\\ \{\ x=0 \wedge y=0\ \}\end{array}}\;\text{ASS}
}{\begin{array}{c}\{\ \mathrm{true}\ \}\\ x := 0;\ y := 0\\ \{x=0 \wedge y=0\}\end{array}}\;\text{SEQ}
\qquad
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{c}\{\ 2x=y \wedge x\leq 10 \wedge x<10\ \}\\ x := x+1\\ \{2(x\text{-}1)=y \wedge x\text{-}1<10\}\end{array}}\;\text{ASS}
\quad
\dfrac{}{\begin{array}{c}\{2(x\text{-}1)=y \wedge x\text{-}1<10\}\\ y := y+2\\ \{\ 2x=y \wedge x\leq 10\ \}\end{array}}\;\text{ASS}
}{\begin{array}{c}\{\ 2x=y \wedge x\leq 10 \wedge x<10\ \}\\ x := x+1;\ y := y+2\\ \{\ 2x=y \wedge x\leq 10\ \}\end{array}}\;\text{SEQ}
}{\begin{array}{c}\{x=0 \wedge y=0\}\\ \mathrm{while}\ x<10\ \mathrm{do}\ (x := x+1;\ y := y+2)\\ \{\ x=10 \wedge y=20\ \}\end{array}}\;\text{WHILE}
}{\begin{array}{c}\{\ \mathrm{true}\ \}\\ x := 0;\ y := 0;\\ \mathrm{while}\ x<10\ \mathrm{do}\ (x := x+1;\ y := y+2)\\ \{\ x=10 \wedge y=20\ \}\end{array}}\;\text{SEQ}
$$

Verification conditions:

$$\mathrm{true} \Longrightarrow 0=0$$

$$x=0 \Longrightarrow x=0 \wedge 0=0$$

$$x=0 \wedge y=0 \Longrightarrow 2x=y \wedge x\leq 10$$

$$2x=y \wedge x\leq 10 \wedge \neg(x<10) \Longrightarrow x=10 \wedge y=20$$

$$2x=y \wedge x\leq 10 \wedge x<10 \Longrightarrow 2((x+1)\text{-}1)=y \wedge (x+1)\text{-}1<10$$

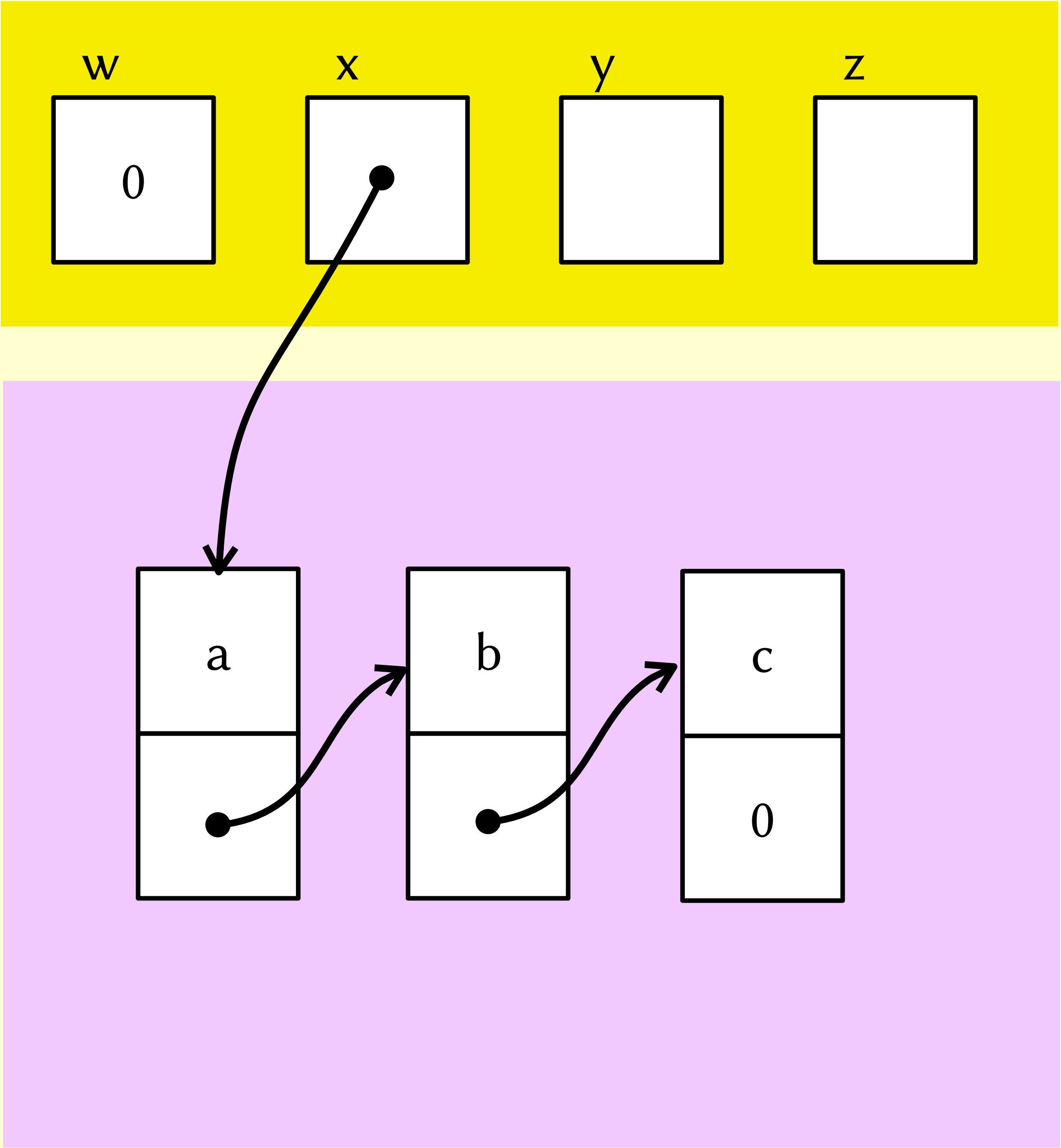
$$2(x\text{-}1)=y \wedge x\text{-}1<10 \Longrightarrow 2x=(y+2) \wedge x\leq 10$$

A challenge for Hoare

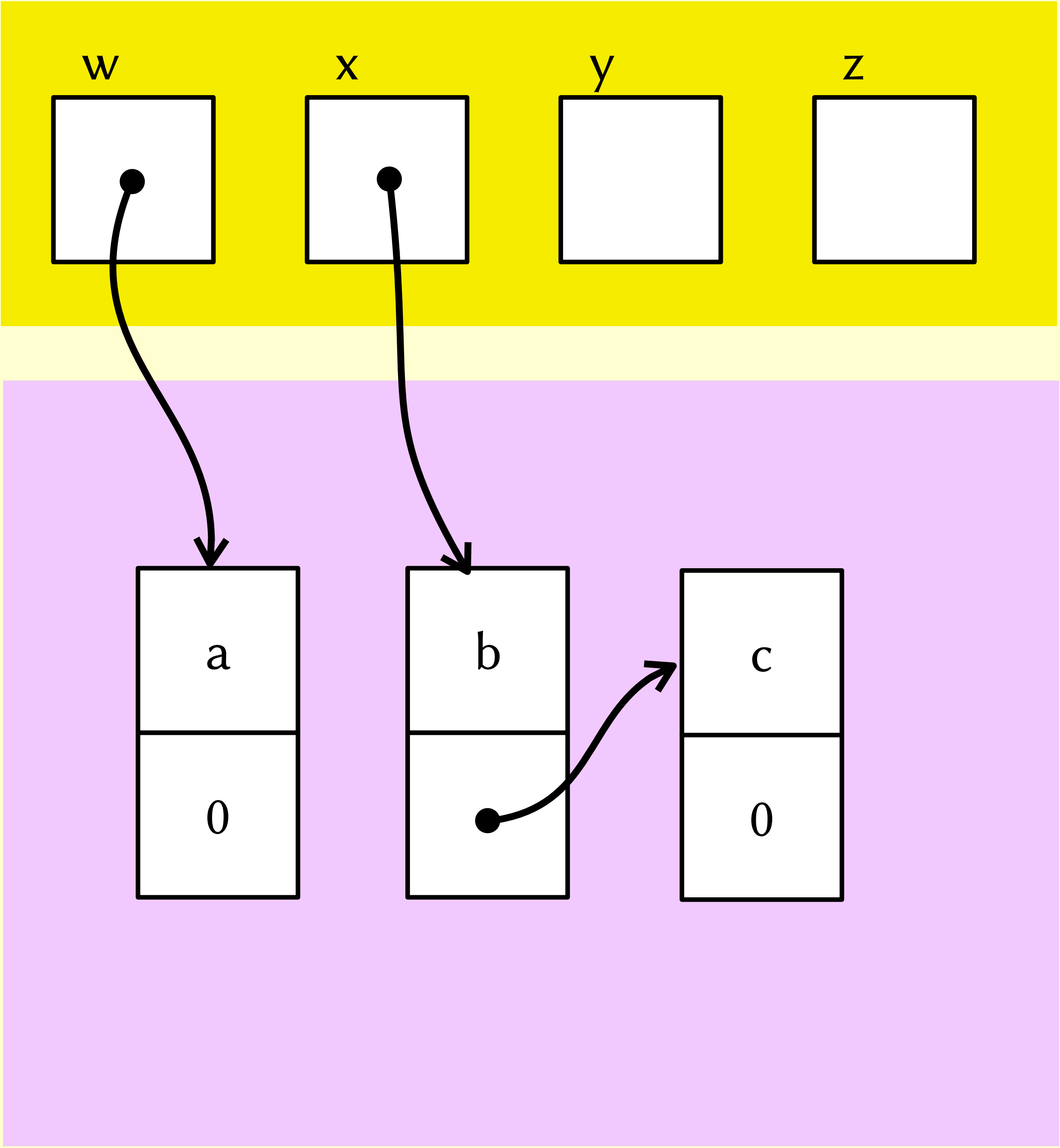
- Hoare logic struggles to reason about heap-allocated data structures like linked lists.

List reverse

```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



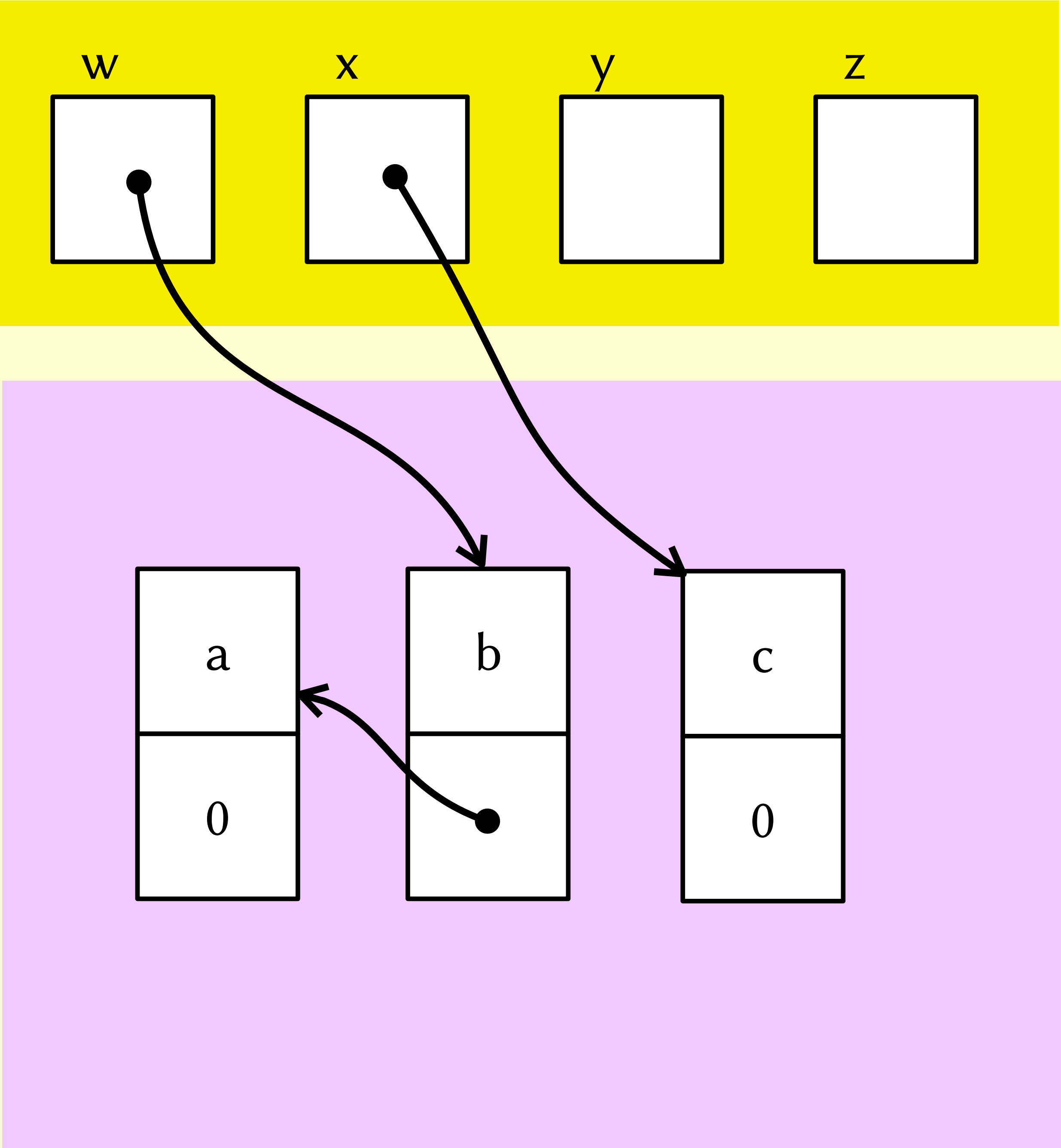
List reverse



```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```

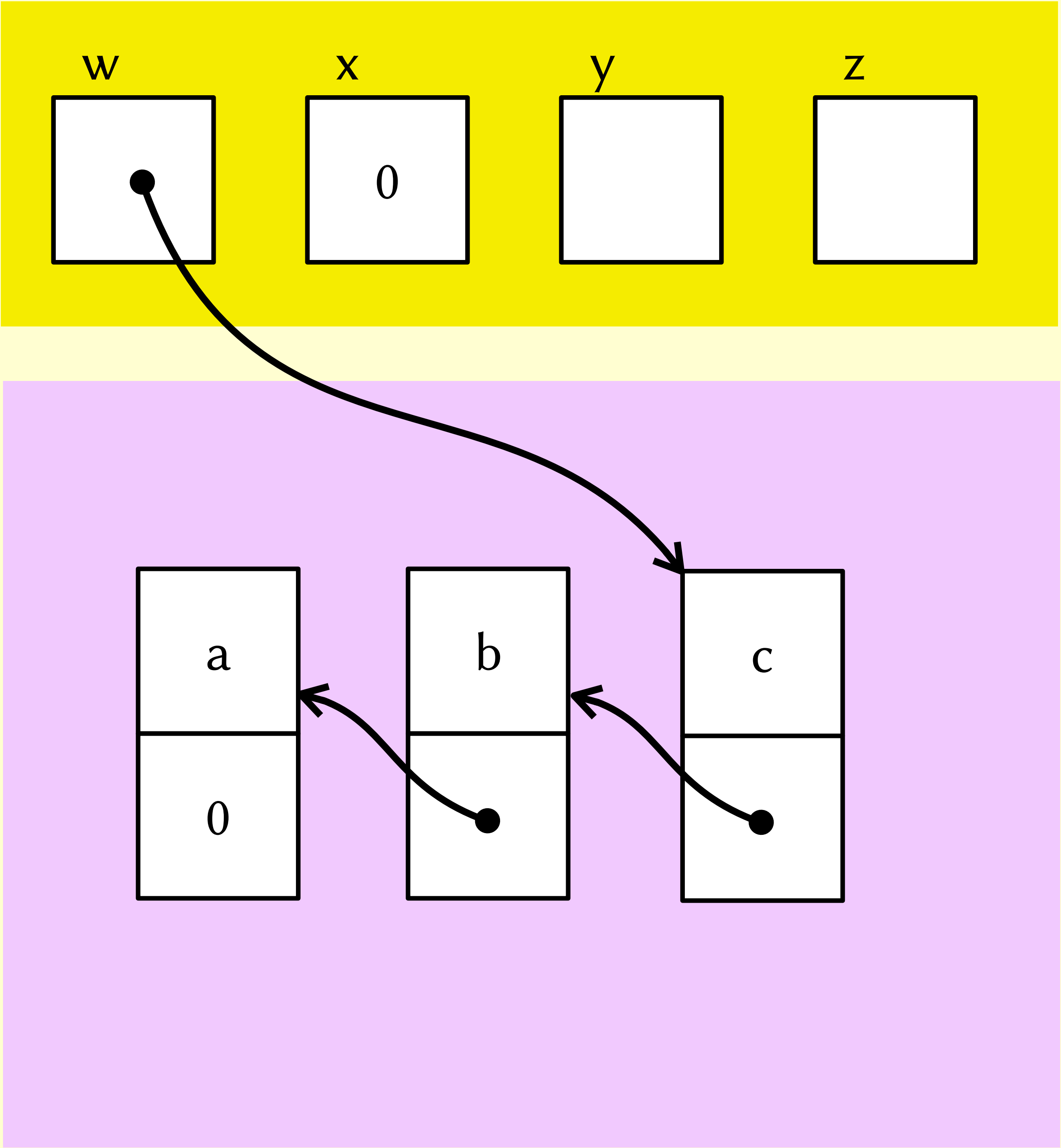
List reverse

```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



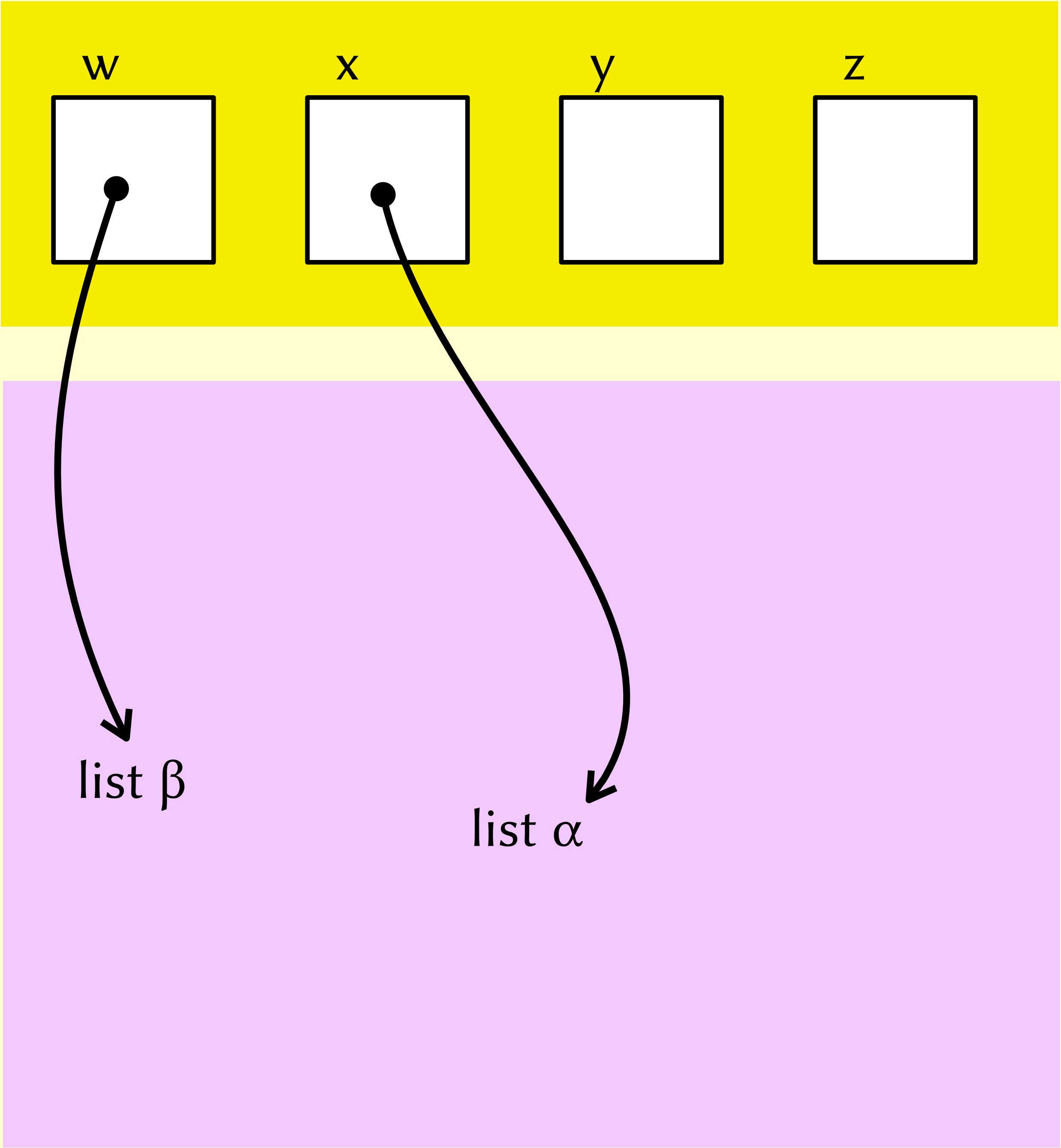
List reverse

```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



List reverse

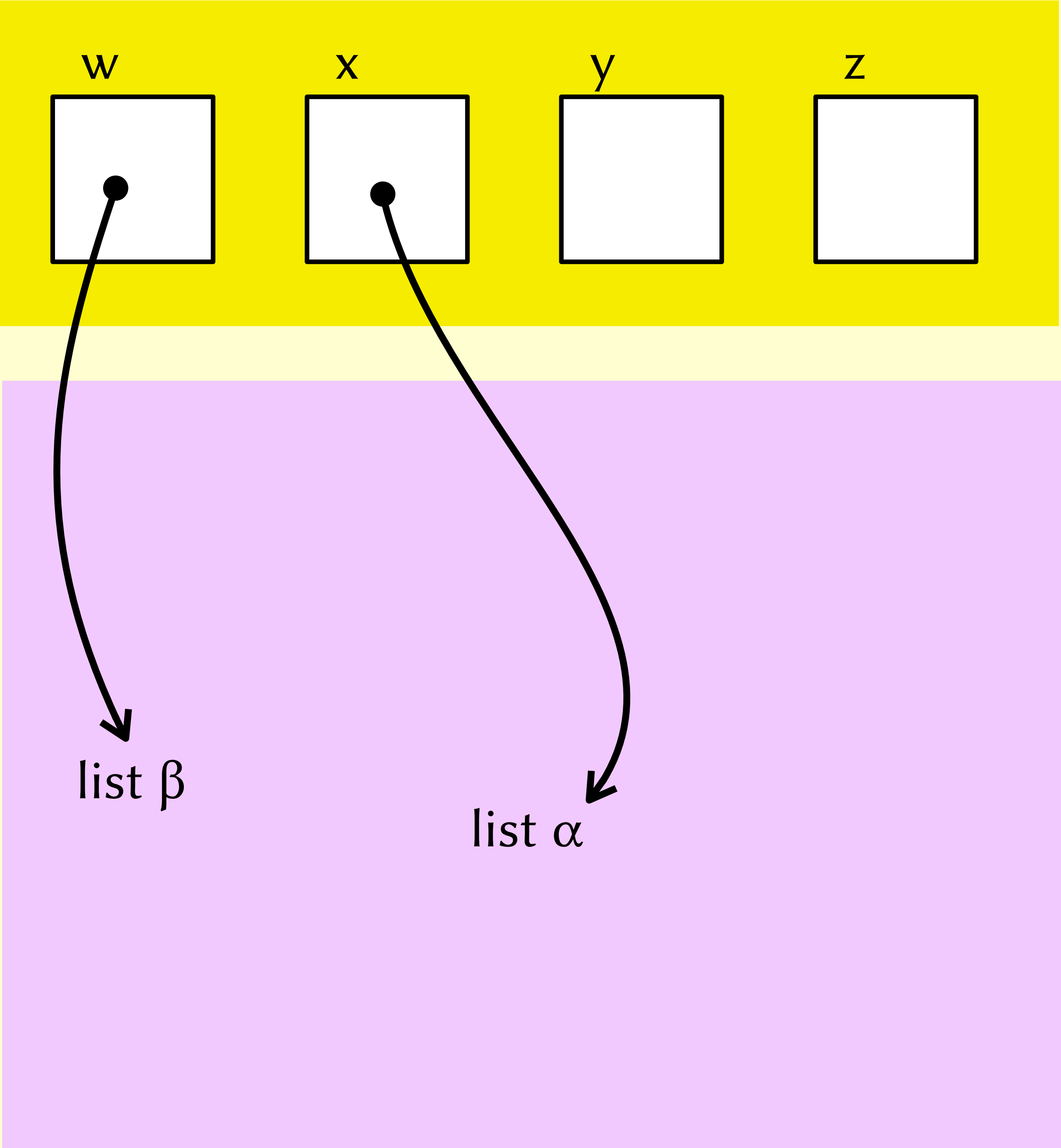
```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



$\delta = -\beta \cdot \alpha$

List reverse

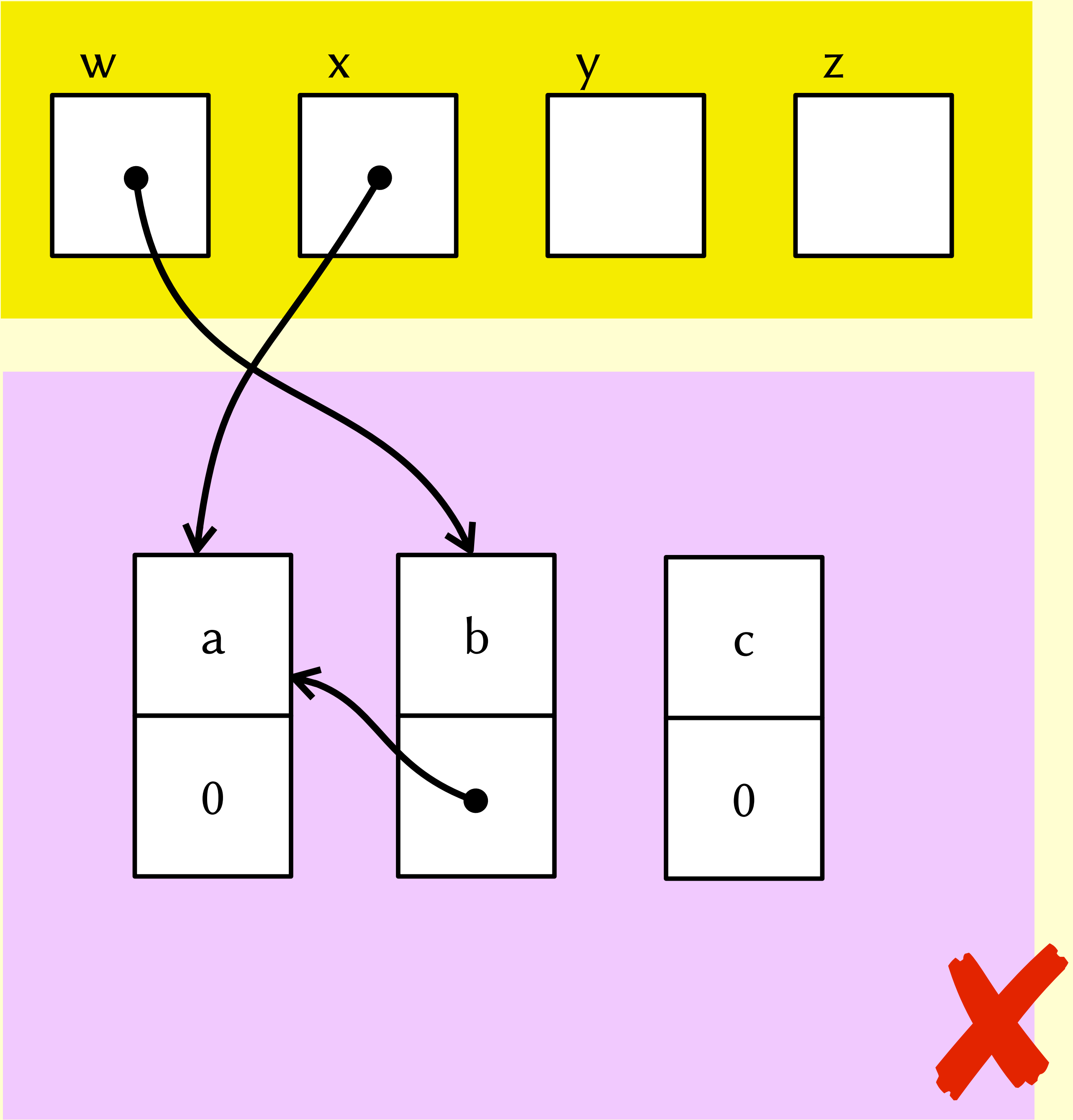
```
{list δ x}
w := 0;
{∃α,β. list α x ∧ list β w ∧ δ = -β·α}
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



δ = -β · α

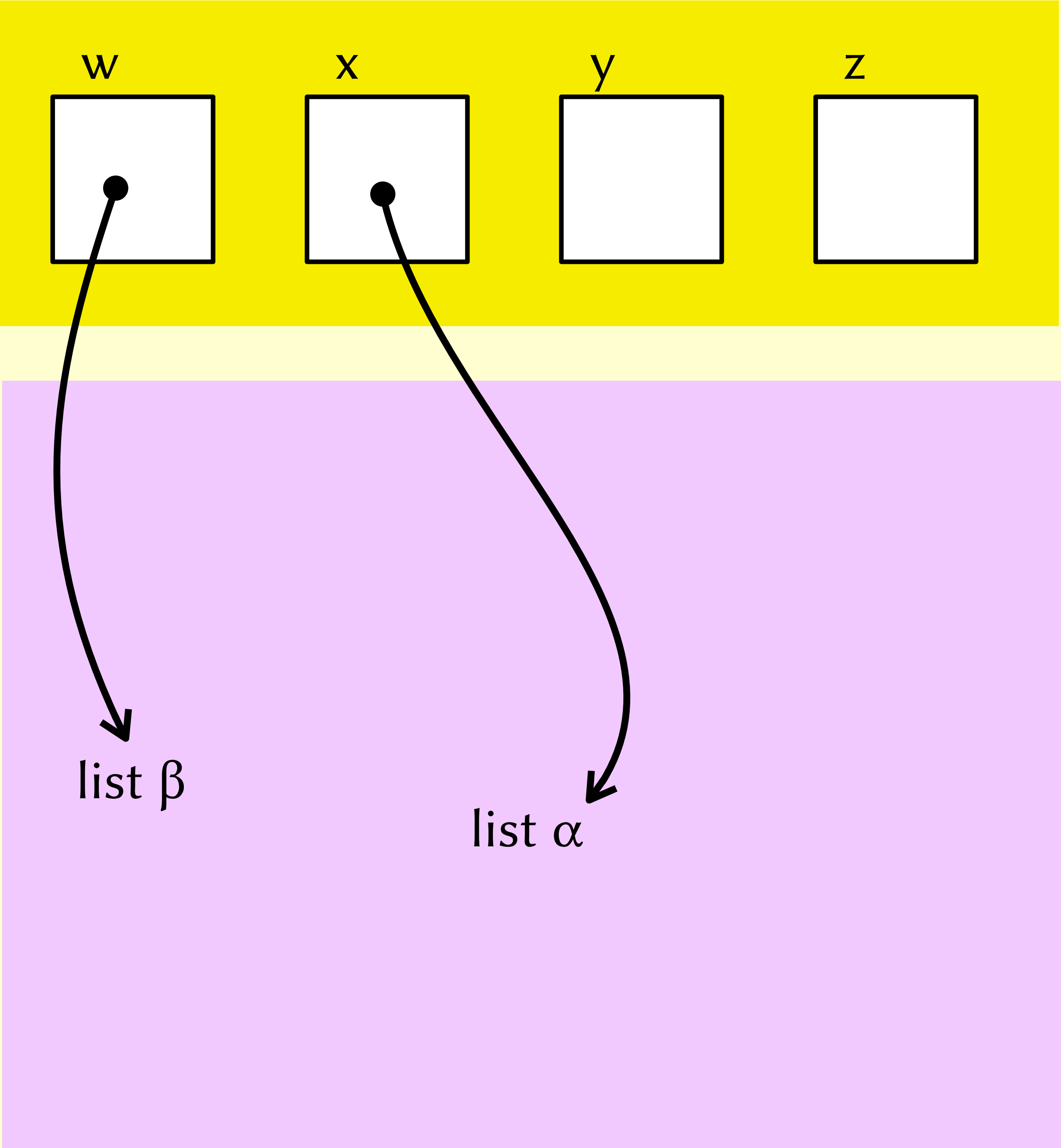
List reverse

```
{list δ x}
w := 0;
{∃α,β. list α x ∧ list β w ∧ δ = -β·α}
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



List reverse

```
$\{\text{list } \delta\ x\}$
w := 0;
$\{\exists\alpha,\beta.\ \text{list } \alpha\ x \wedge \text{list } \beta\ w \wedge \delta = -\beta\cdot\alpha$
$\wedge\ (\forall z.\ \text{reach}(x,z) \wedge \text{reach}(w,z) \Rightarrow z=0)\}$
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
$\{\text{list } -\delta\ w\}$
```



$\delta = -\beta \cdot \alpha$

List reverse

```
{list δ x}
list_reverse(x,w)
{list -δ w}
```

List reverse

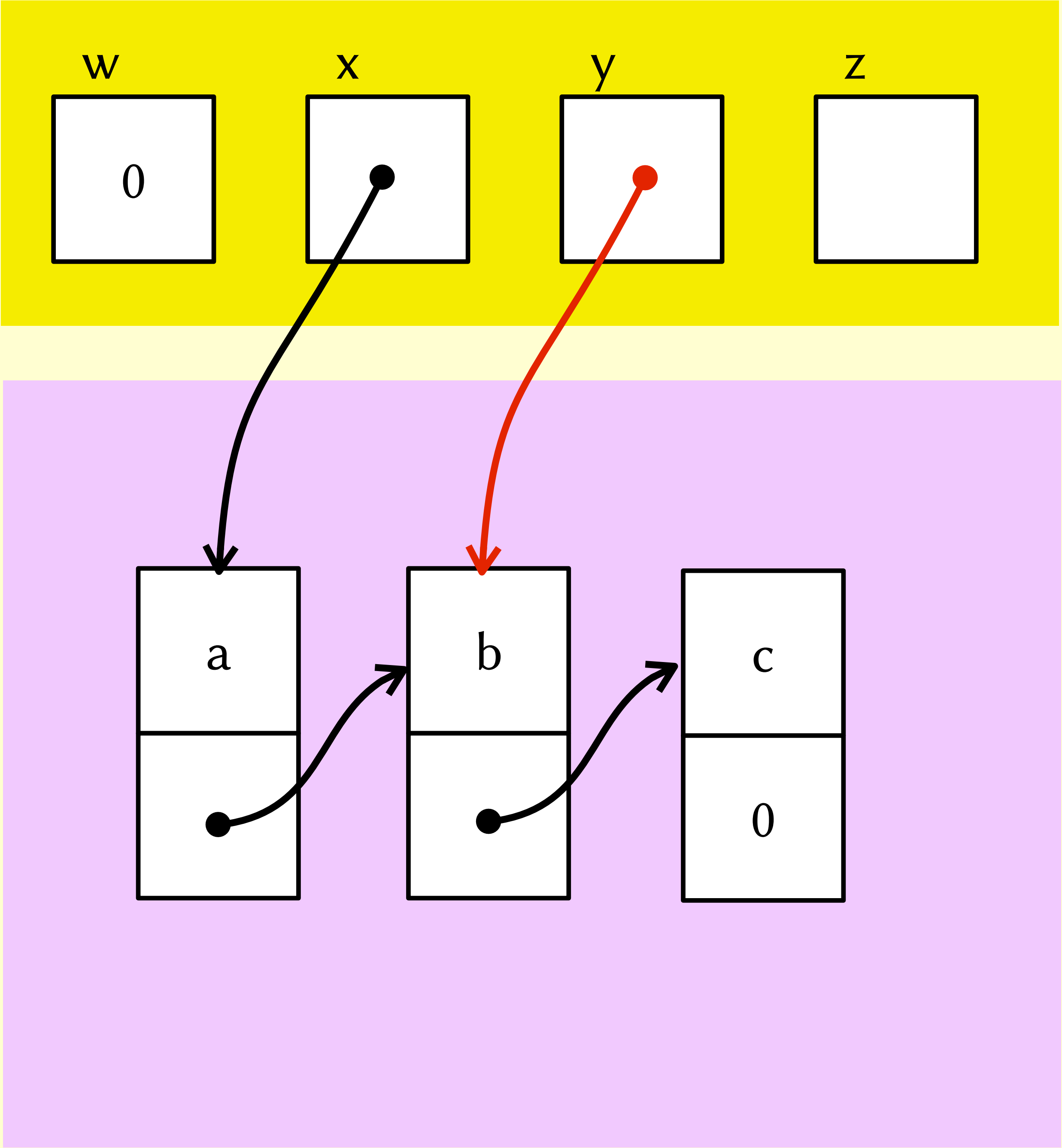
$\{\mathsf{list}\ \delta\ x \wedge \mathsf{list}\ \varepsilon\ y\}$
list_reverse(x,w)
$\{\mathsf{list}\ {-\delta}\ w\}$

$$\frac{\{\,p\,\}\ C\ \{\,q\,\}\quad (\dagger)}{\{\,p \wedge r\,\}\ C\ \{\,q \wedge r\,\}}$$

[†]provided r doesn’t mention any variable modified by C

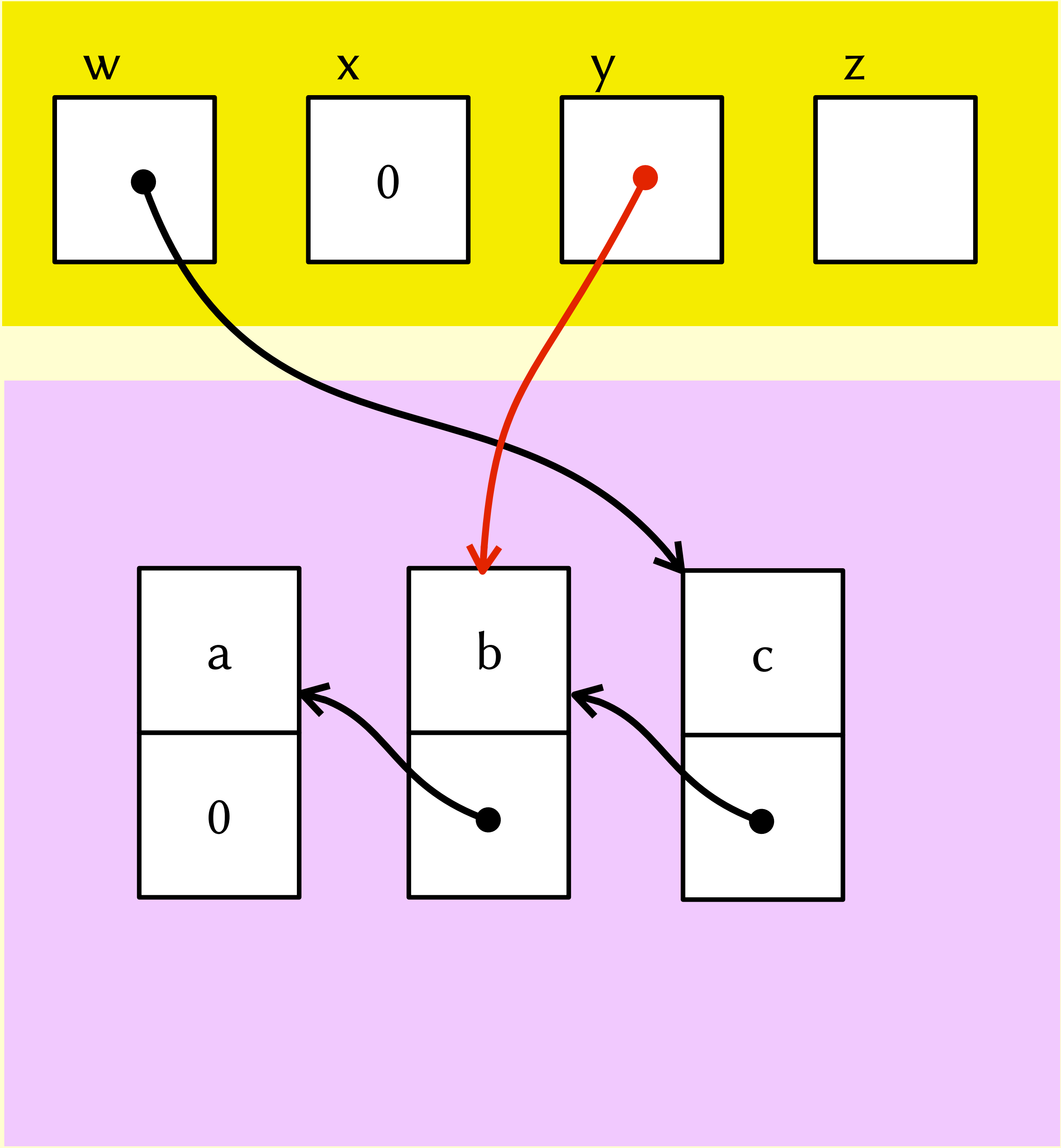
✗

List reverse



{list δ x ∧ list ε y}
list_reverse(x,w)
{list -δ w}

List reverse



{list δ x ∧ list ε y}
list_reverse(x,w)
{list -δ w}

List reverse

$\{\mathrm{list}\ \delta\ x \land \mathrm{list}\ \varepsilon\ y$

$\land\ (\forall z.\ \mathrm{reach}(x,z) \land \mathrm{reach}(y,z) \Longrightarrow z=0)\}$

list_reverse(x,w)

$\{\mathrm{list}\ {-\delta}\ w\}$

List reverse

```
{list δ x ∧ list ε y
∧ (∀z. reach(x,z) ∧ reach(y,z) ⟹ z=0)}
w := 0;
{∃α,β. list α x ∧ list β w ∧ δ = -β·α
∧ (∀z. reach(x,z) ∧ reach(w,z) ⟹ z=0)}
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```

List reverse

```
{list δ x ∧ list ε y
∧ (∀z. reach(x,z) ∧ reach(y,z) ⟹ z=0)}
w := 0;
{∃α,β. list α x ∧ list β w ∧ δ = -β·α
∧ (∀z. reach(x,z) ∧ reach(w,z) ⟹ z=0)
∧ list ε y
∧ (∀z. (reach(x,z) ∨ reach(w,z))
  ∧ reach(y,z) ⟹ z=0)}
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```

List reverse

```
$\{\mathrm{list}\ \delta\ x \wedge \mathrm{list}\ \varepsilon\ y$
$\wedge (\forall z.\ \mathrm{reach}(x,z) \wedge \mathrm{reach}(y,z) \Rightarrow z=0)\}$
w := 0;
$\{\exists \alpha,\beta.\ \mathrm{list}\ \alpha\ x \wedge \mathrm{list}\ \beta\ w \wedge \delta = -\beta \cdot \alpha$
$\wedge (\forall z.\ \mathrm{reach}(x,z) \wedge \mathrm{reach}(w,z) \Rightarrow z=0)$
$\wedge \mathrm{list}\ \varepsilon\ y$
$\wedge (\forall z.\ (\mathrm{reach}(x,z) \vee \mathrm{reach}(w,z))$
  $\wedge \mathrm{reach}(y,z) \Rightarrow z=0)\}$
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
$\{\mathrm{list}\ -\delta\ w \wedge \mathrm{list}\ \varepsilon\ y$
$\wedge (\forall z.\ \mathrm{reach}(w,z) \wedge \mathrm{reach}(y,z) \Rightarrow z=0)\}$
```

List reverse

- Summary: the proof is fiddly and not modular, but it can be done.

$\{\mathrm{list}\ \delta\ x \wedge \mathrm{list}\ \varepsilon\ y$
$\wedge\ (\forall z.\ \mathrm{reach}(x,z) \wedge \mathrm{reach}(y,z) \Longrightarrow z=0)\}$
list_reverse(x,w)
$\{\mathrm{list}\ {-\delta}\ w \wedge \mathrm{list}\ \varepsilon\ y$
$\wedge\ (\forall z.\ \mathrm{reach}(w,z) \wedge \mathrm{reach}(y,z) \Longrightarrow z=0)\}\}$

Cigarettes and Alcohol

Peter O'Hearn
(1963–)

John C. Reynolds
(1935–2013)

Cigarettes and Alcohol

$$
\begin{aligned}
(P * Q)\ s &= \exists s_1, s_2.\ s = s_1 + s_2 \text{ and } (P\ s_1) \text{ and } (Q\ s_2) \\
(P \wedge Q)\ s &= (P\ s) \text{ and } (Q\ s) \\
(P \vee Q)\ s &= (P\ s) \text{ or } (Q\ s) \\
(\neg P)\ s &= \text{not } (P\ s) \\
\text{🚬}\ s &= s \geq \pounds 5 \\
\text{🥃}\ s &= s \geq \pounds 20
\end{aligned}
$$

$$
\begin{aligned}
(\text{🚬} \wedge \text{🥃})\ s &= (\text{🚬}\ s) \text{ and } (\text{🥃}\ s) \\
&= s \geq \pounds 5 \text{ and } s \geq \pounds 20 \\
&= s \geq \pounds 20
\end{aligned}
$$

Cigarettes and Alcohol

$(P * Q)\ s \;=\; \exists s_1, s_2.\ s = s_1 + s_2 \text{ and } (P\ s_1) \text{ and } (Q\ s_2)$

$(P \wedge Q)\ s \;=\; (P\ s) \text{ and } (Q\ s)$

$(P \vee Q)\ s \;=\; (P\ s) \text{ or } (Q\ s)$

$(\neg P)\ s \;=\; \text{not } (P\ s)$

$s \;=\; s \geq £5$

$s \;=\; s \geq £20$

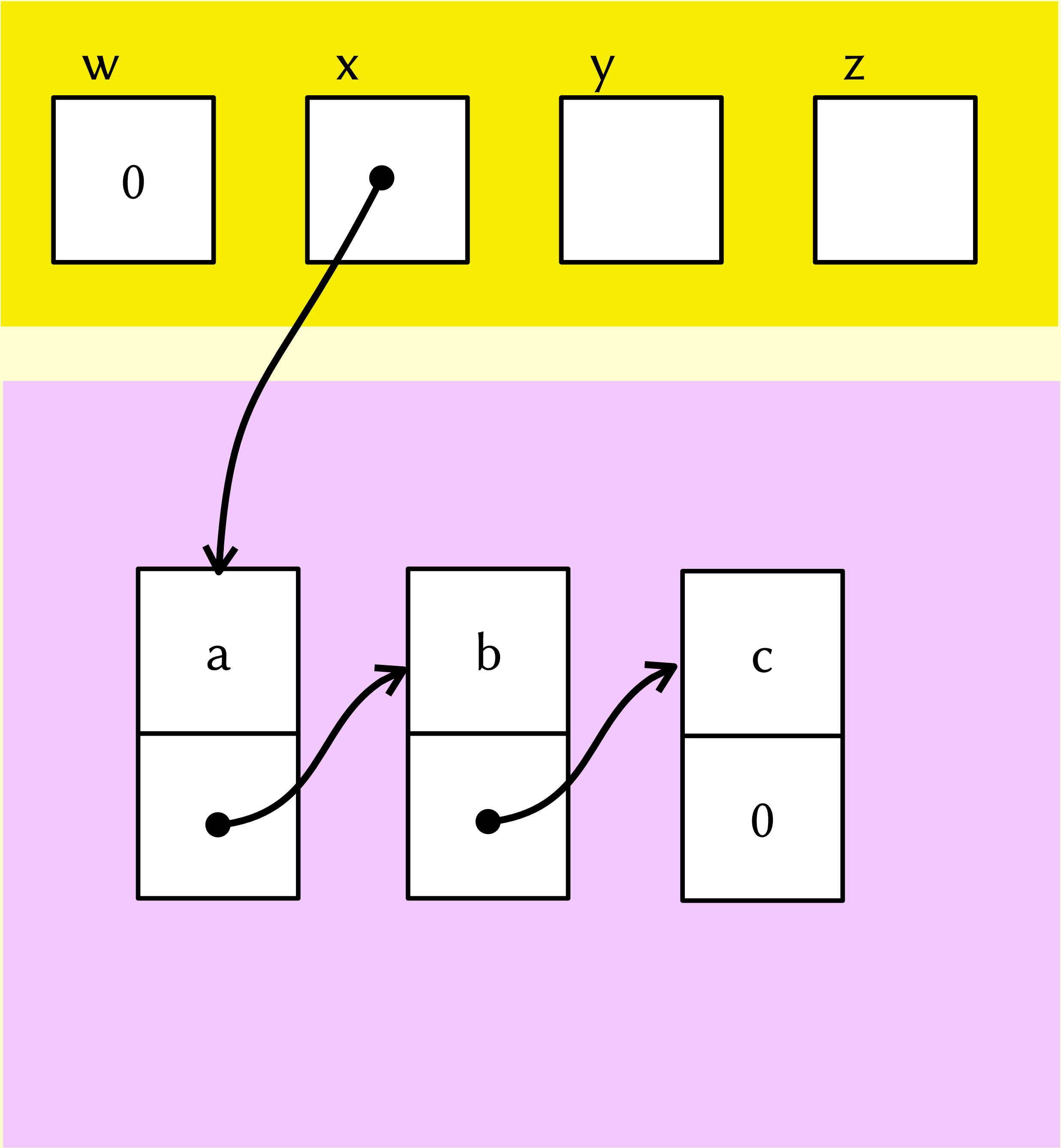
($*$) $s \;=\; \exists s_1, s_2.\ s = s_1 + s_2 \text{ and }$ (s_1) and (s_2)

$= \exists s_1, s_2.\ s = s_1 + s_2 \text{ and } s_1 \geq £5 \text{ and } s_2 \geq £20$

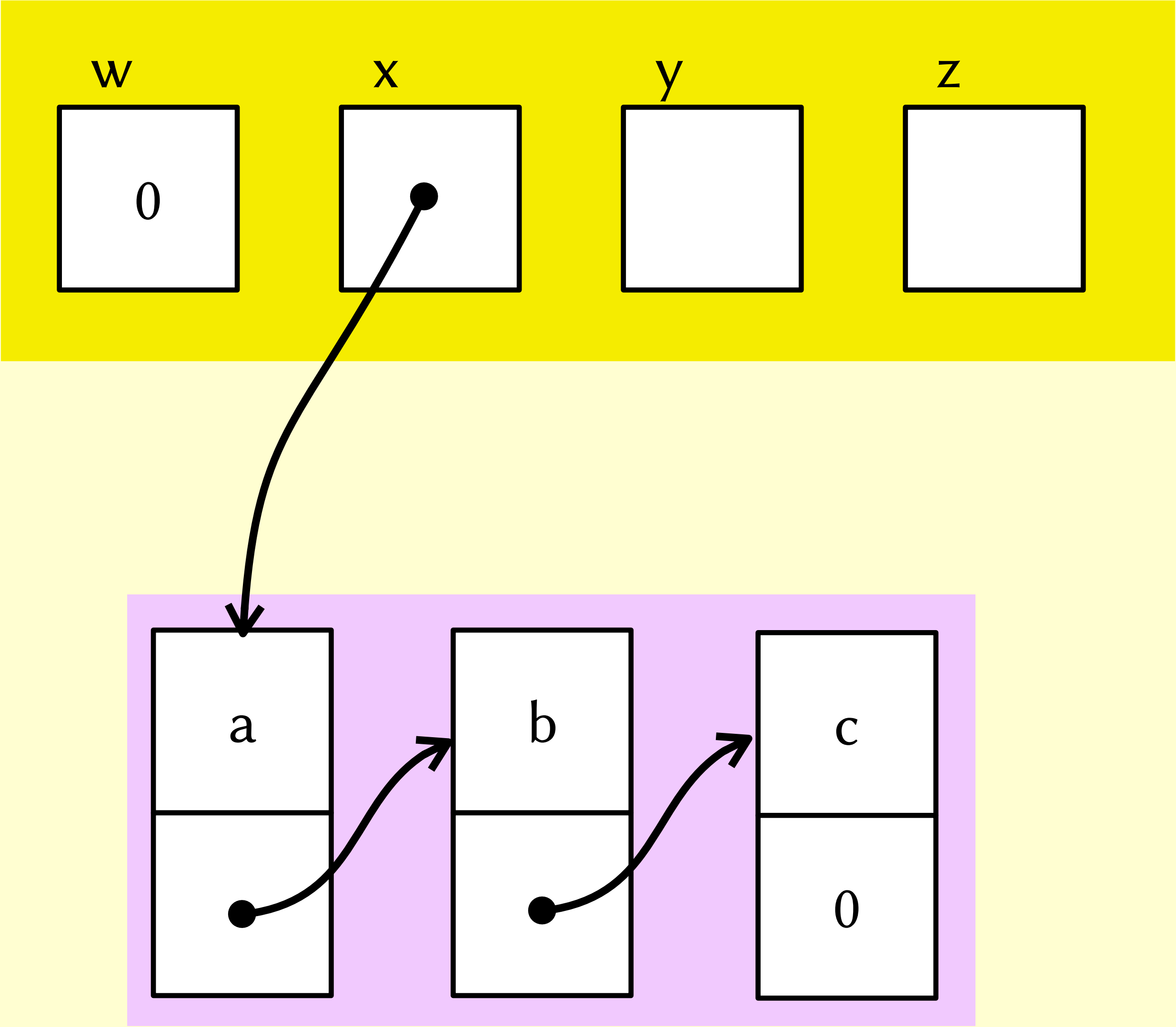
$= s \geq £25$

List reverse

```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```



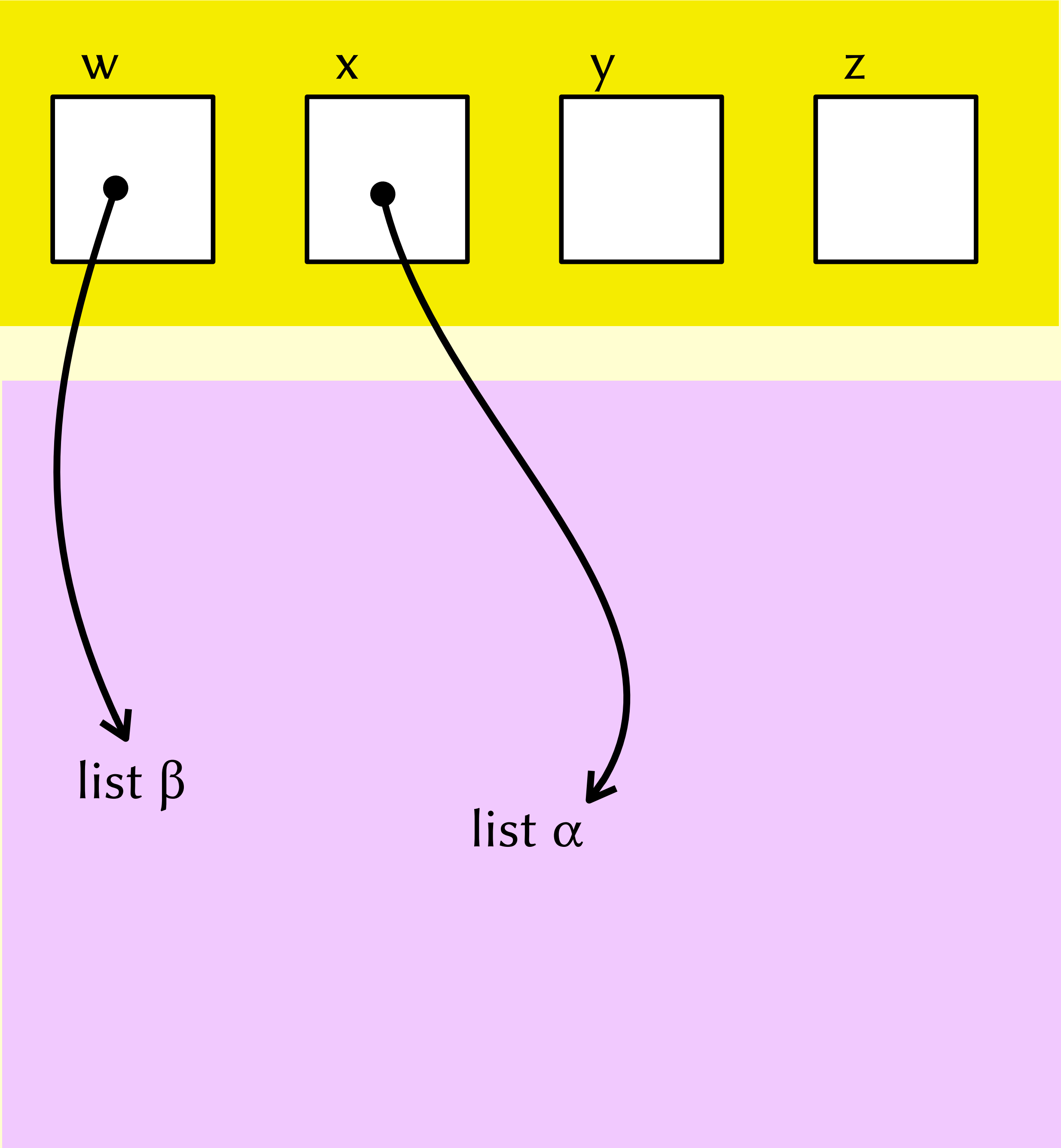
List reverse



```
{list δ x}
w := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{list -δ w}
```

List reverse

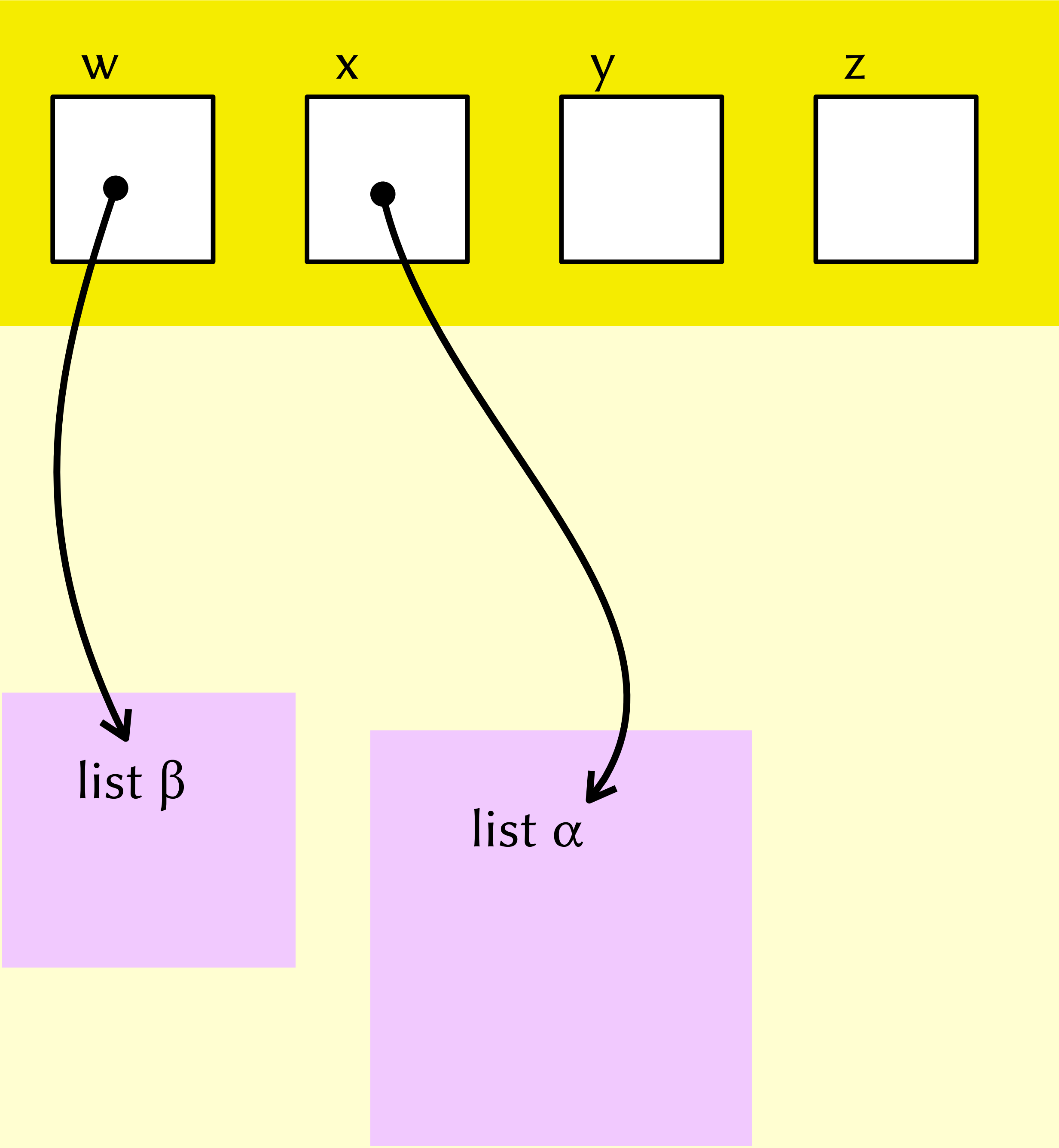
```
$\{\text{list } \delta\ x\}$
w := 0;
$\{\exists\alpha,\beta.\ \text{list } \alpha\ x \wedge \text{list } \beta\ w \wedge \delta = -\beta\cdot\alpha$
$\wedge\ (\forall z.\ \text{reach}(x,z) \wedge \text{reach}(w,z) \Longrightarrow z=0)\}$
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
$\{\text{list } -\delta\ w\}$
```



$\delta = -\beta \cdot \alpha$

List reverse

```
{$\mathrm{list}\ \delta\ x$}
w := 0;
{$\exists\alpha,\beta.\ \mathrm{list}\ \alpha\ x * \mathrm{list}\ \beta\ w * \delta = -\beta\cdot\alpha$}
while (x≠0) do {
    z := [x+1];
    [x+1] := w;
    w := x;
    x := z;
}
{$\mathrm{list}\ {-\delta}\ w$}
```



$\delta = -\beta \cdot \alpha$

List reverse

```
{list δ x}
list_reverse(x,w)
{list -δ w}
```

List reverse

{list δ x ∗ list ε y ∗ tree t}
list_reverse(x,w)
{list -δ w}

List reverse

{list δ x * list ε y * tree t}
list_reverse(x,w)
{list -δ w * list ε y * tree t}

$$\frac{\{\, p \,\}\ C\ \{\, q \,\} \quad (\dagger)}{\{\, p * r \,\}\ C\ \{\, q * r \,\}}$$

[†]provided r doesn’t mention any variable modified by C

Conclusion

- Hoare Logic kicked off the field of software verification around 1969.

- Hoare Logic always struggled to reason about heap-allocated data structures.

- Separation Logic provided a solution around 2001.

- This has powered a lot of software verification tools since then, such as Facebook Infer.