Verifying heap-manipulating programs using Separation Logic

A lecture by John Wickerson,
as part of the
Software Reliability series

4 March 2014

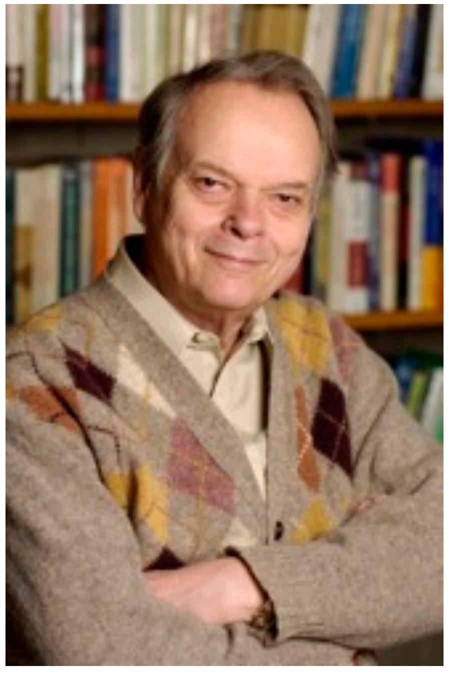
The Players

Hoare Logic
(1969)

Separation Logic
(2001)

Tony Hoare

Peter O'Hearn



John Reynolds

Lecture Plan

- A 20th century proof of `list_reverse`
- A proof of `list_reverse` in separation logic
- Separation logic's proof rules
- Soundness of the Frame rule

References

- Mike Gordon. *Hoare Logic.* Lecture Notes, 2014.
- John Reynolds. *Separation Logic: A Logic for Shared Mutable Data Structures.* LICS 2002.
- Peter O'Hearn, John Reynolds and Hongseok Yang. *Local Reasoning about Programs that Alter Data Structures.* CSL 2001.

Lecture Plan

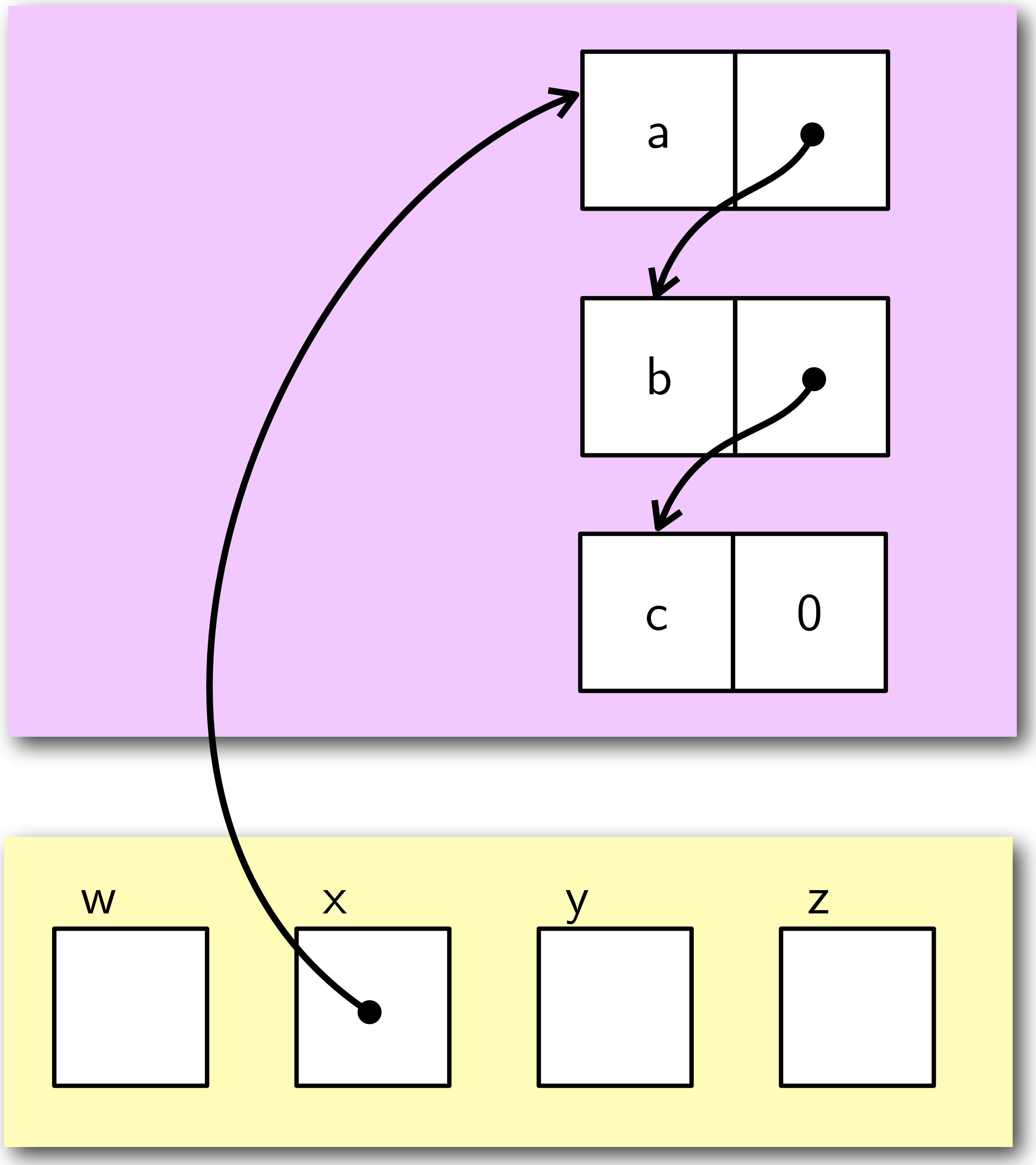
- A 20th century proof of `list_reverse`
- A proof of `list_reverse` in separation logic
- Separation logic's proof rules
- Soundness of the Frame rule

How `list_reverse` works

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

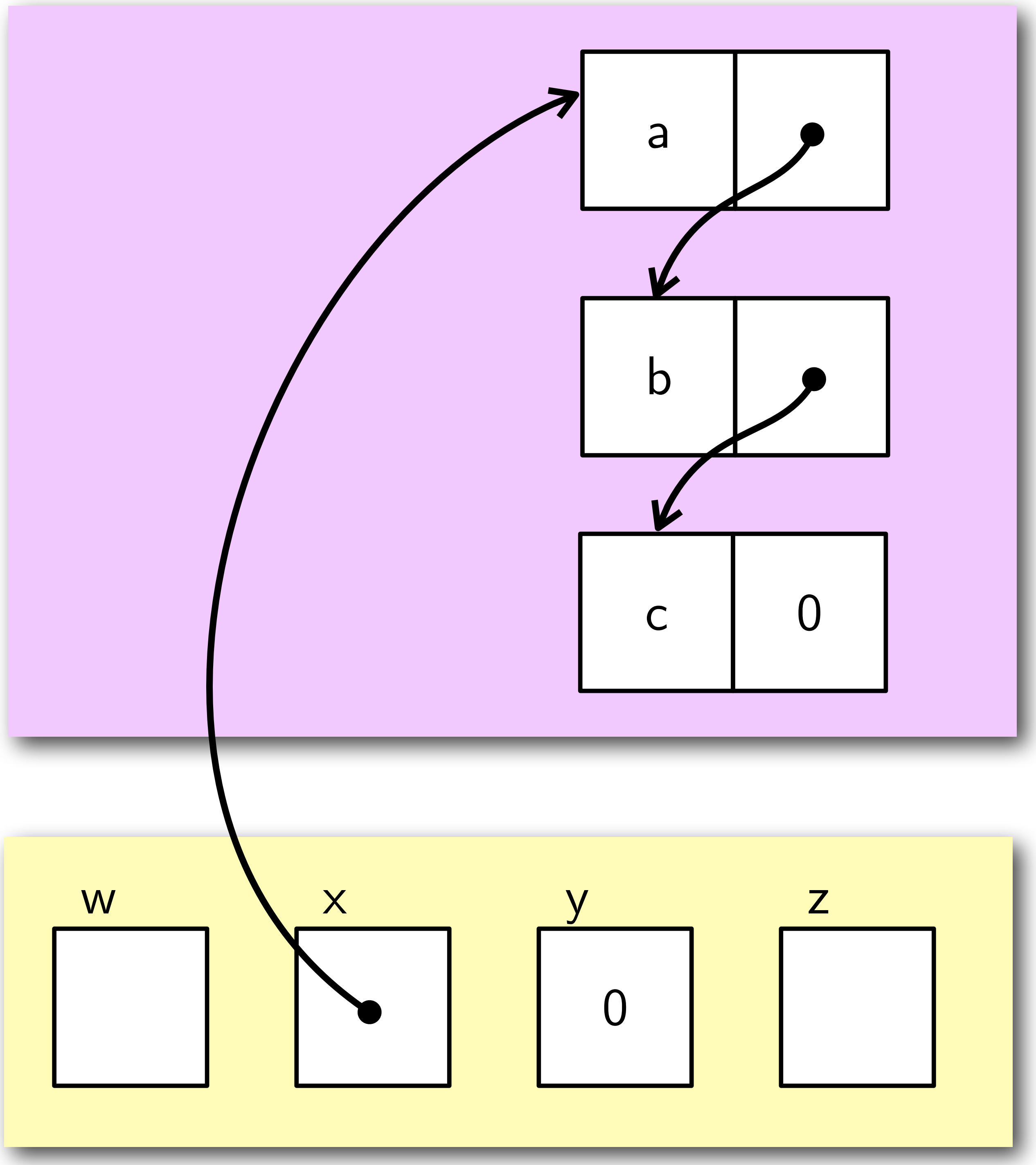
How `list_reverse` works

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



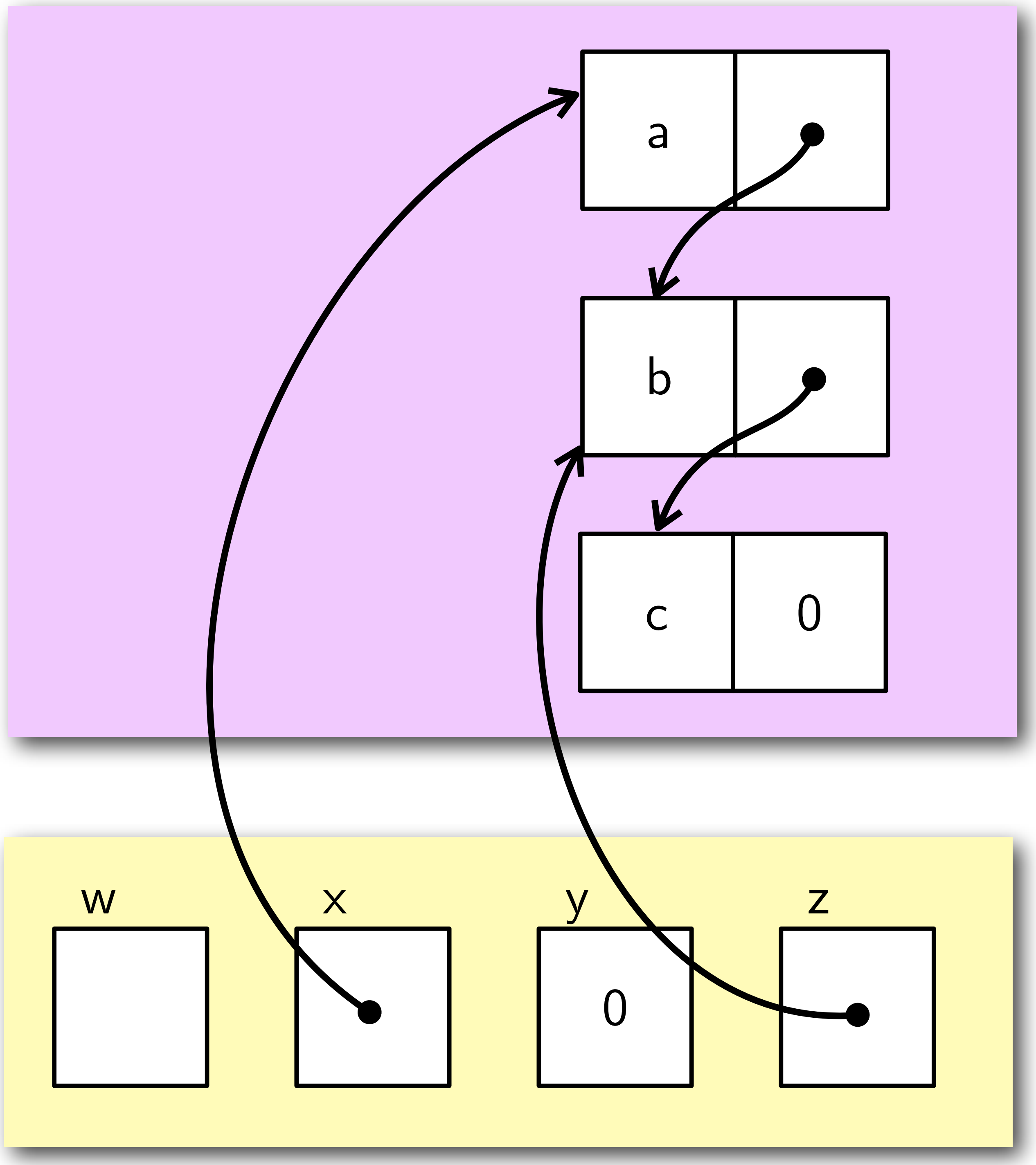
How `list_reverse` works

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



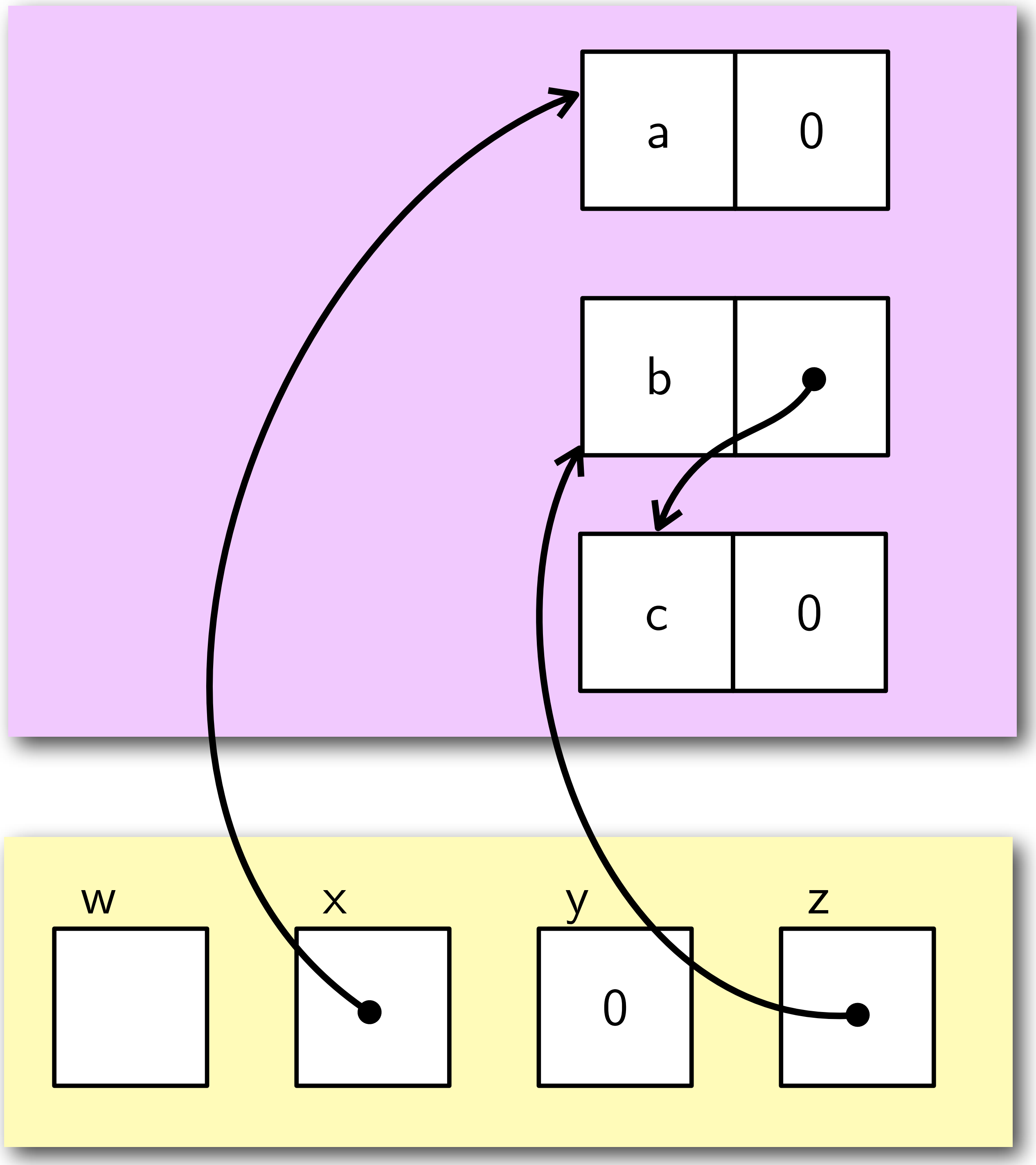
How `list_reverse` works

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



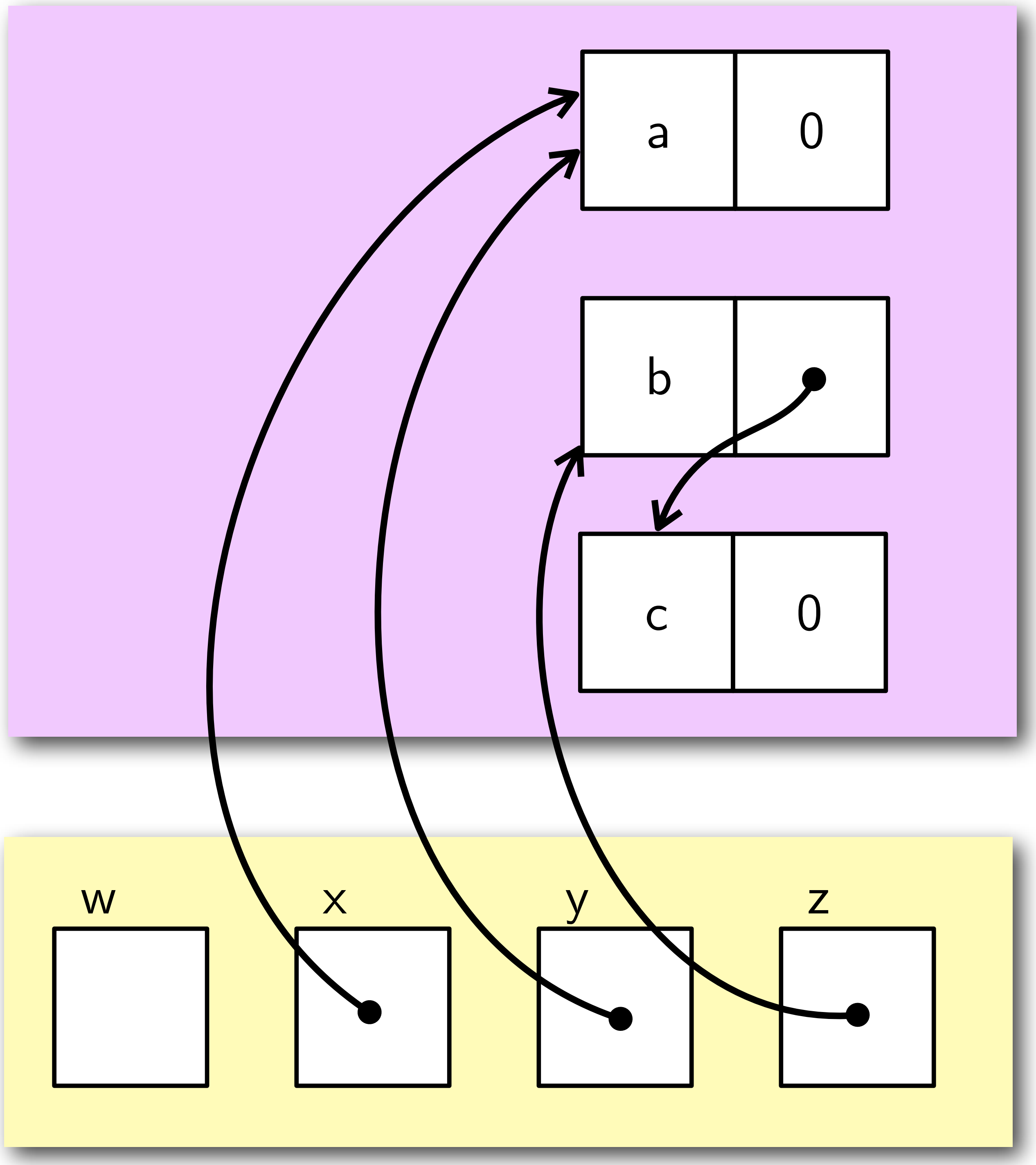
How `list_reverse` works

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



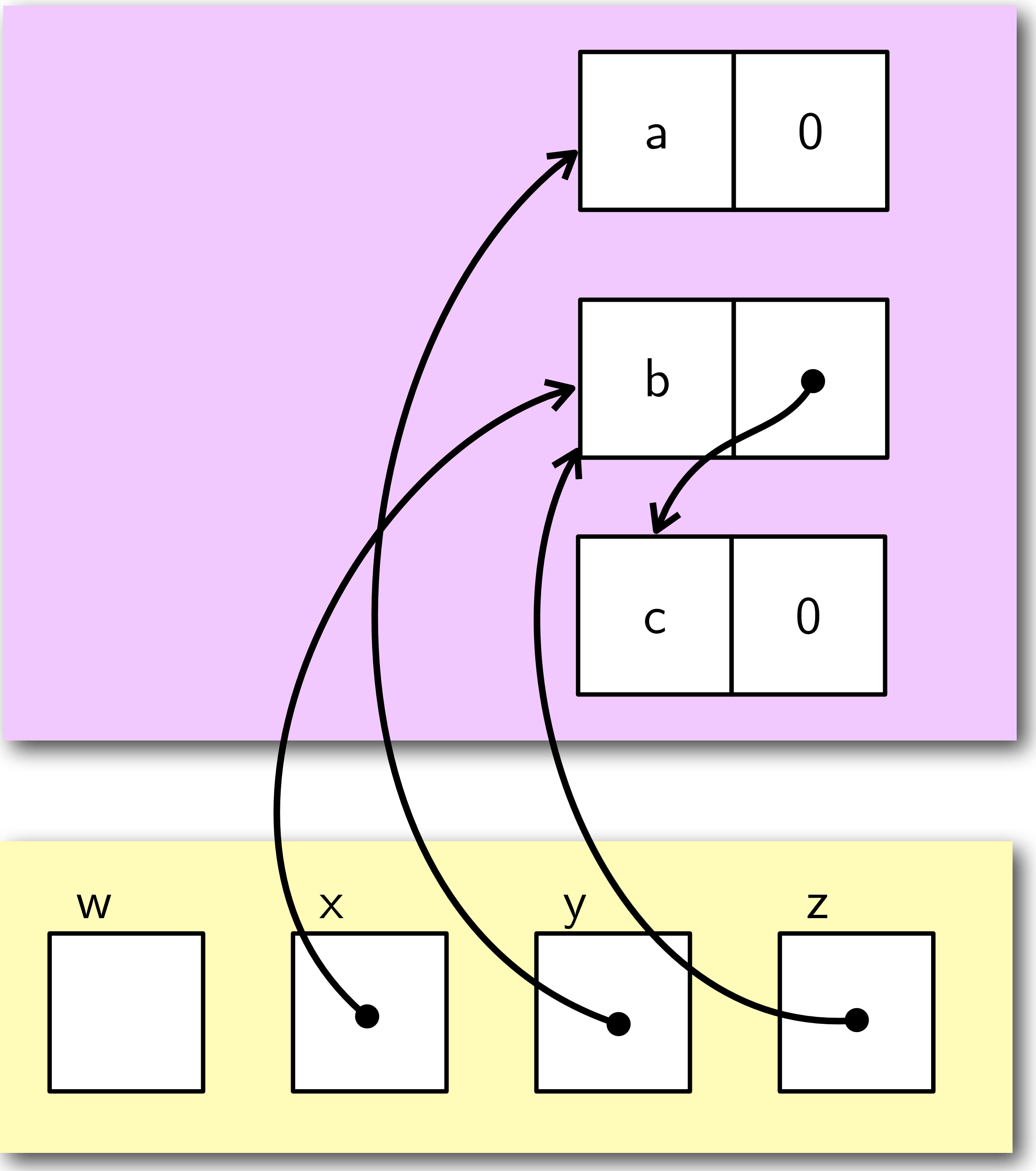
How `list_reverse` works

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



How `list_reverse` works

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

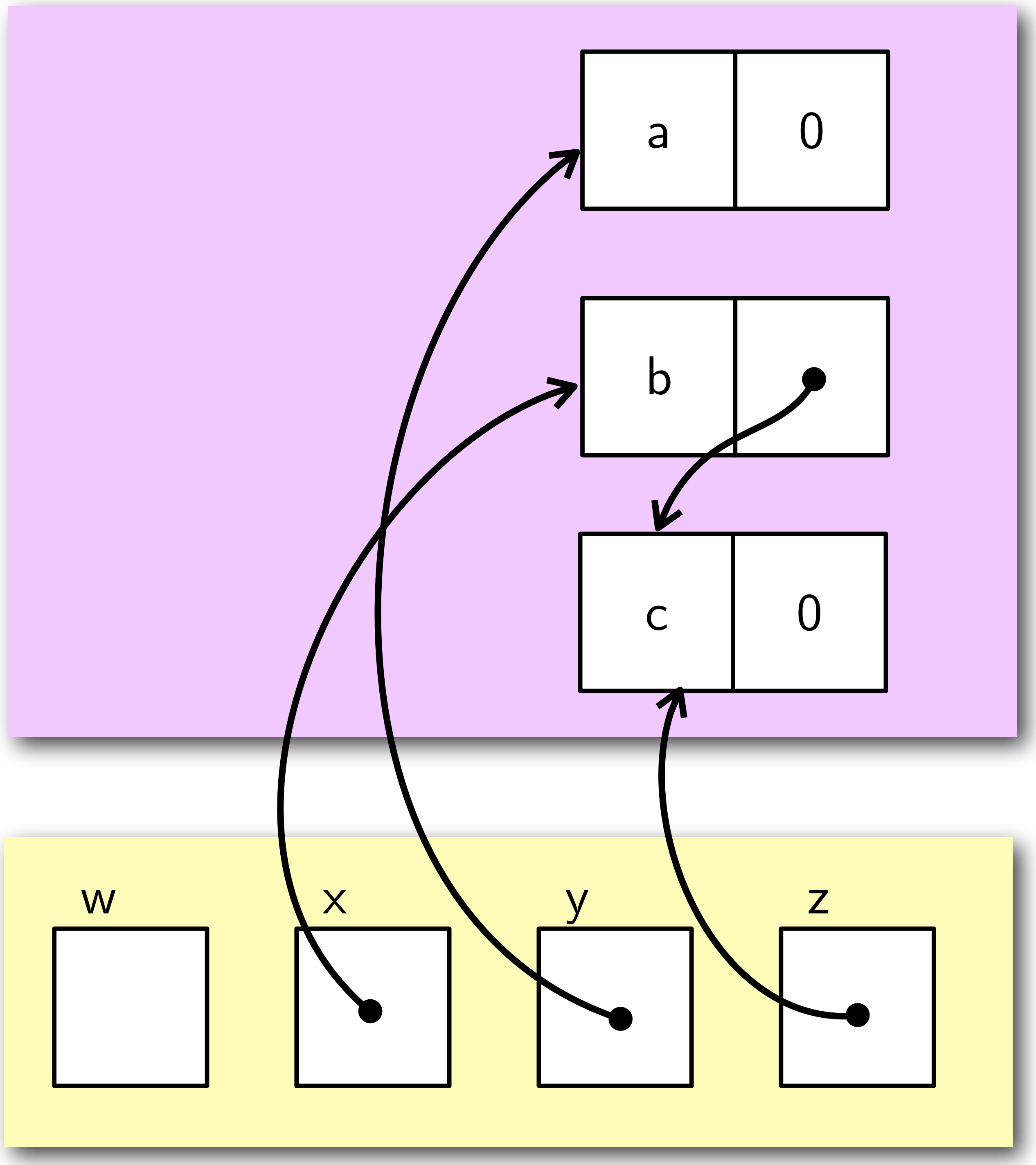


How `list_reverse` works

{list δ x}

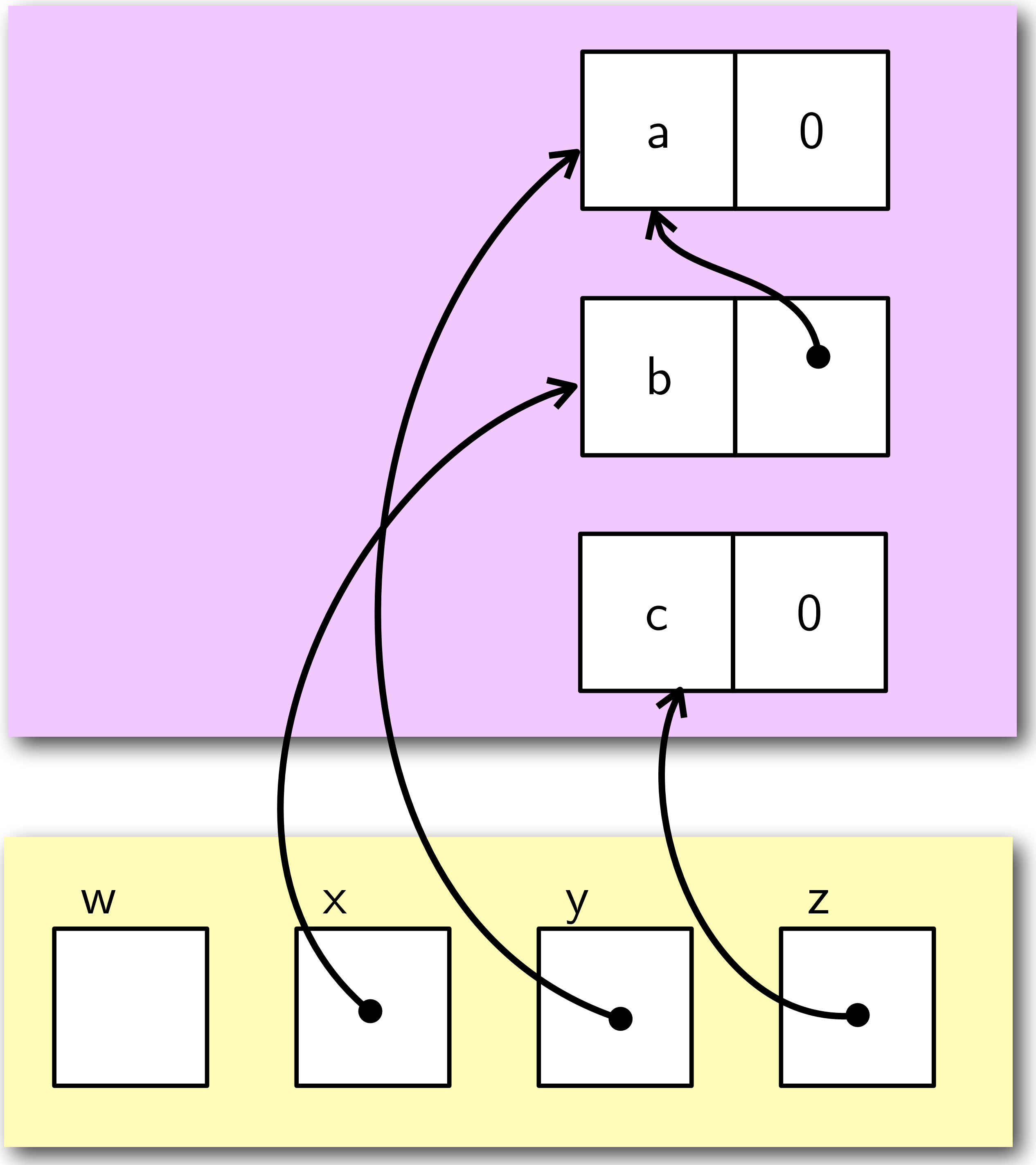
```
y := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}



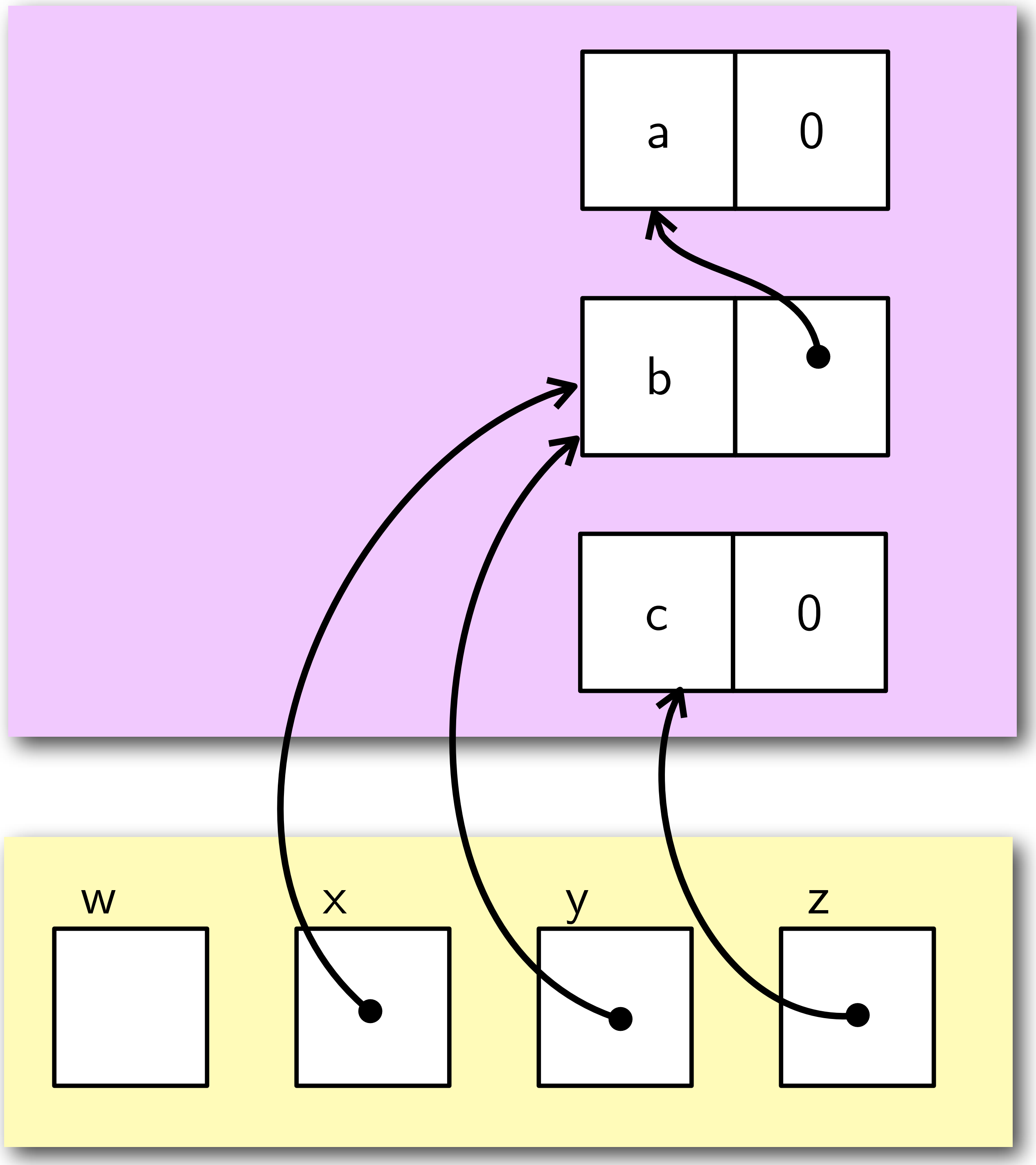
How `list_reverse` works

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



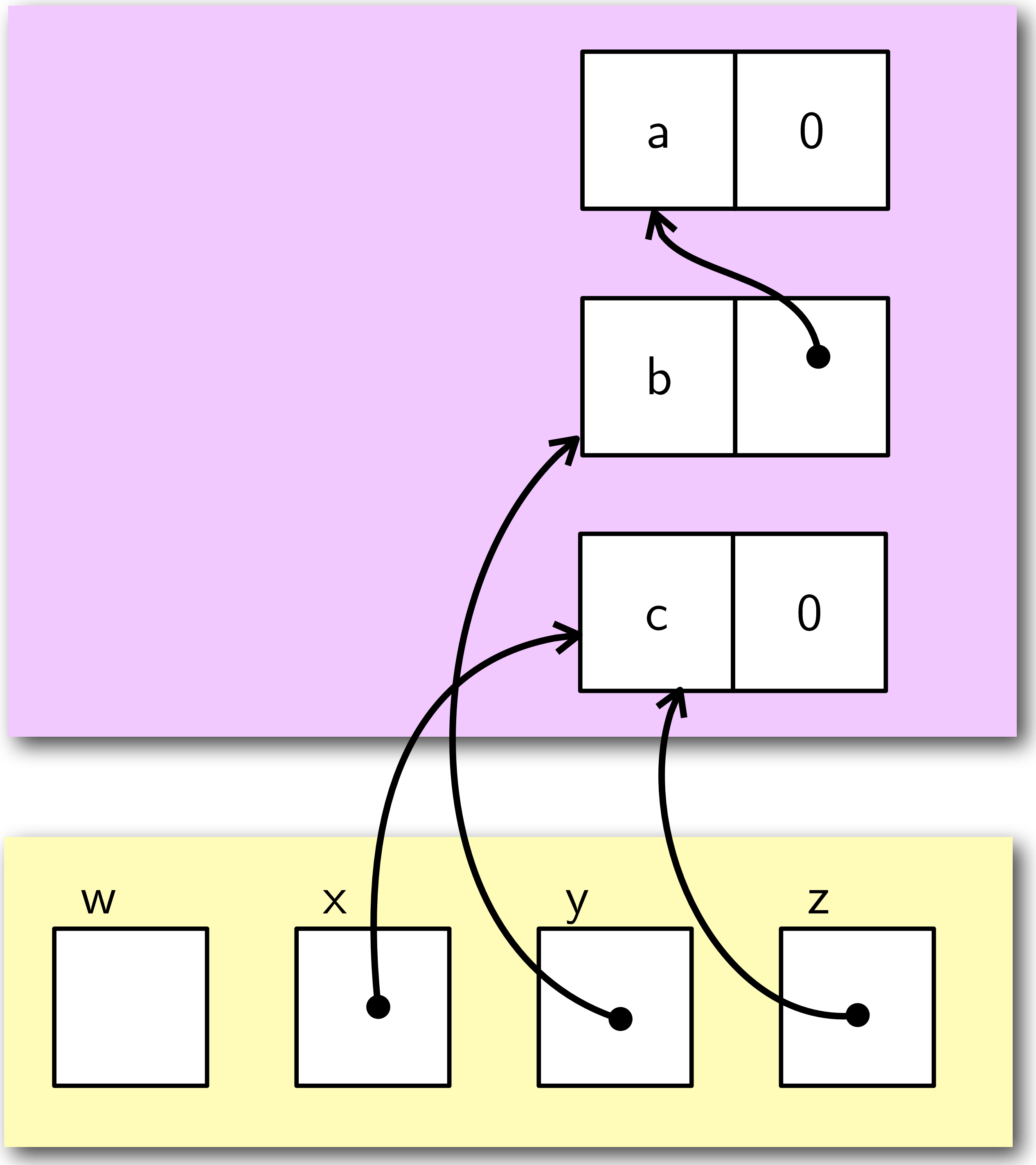
How `list_reverse` works

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

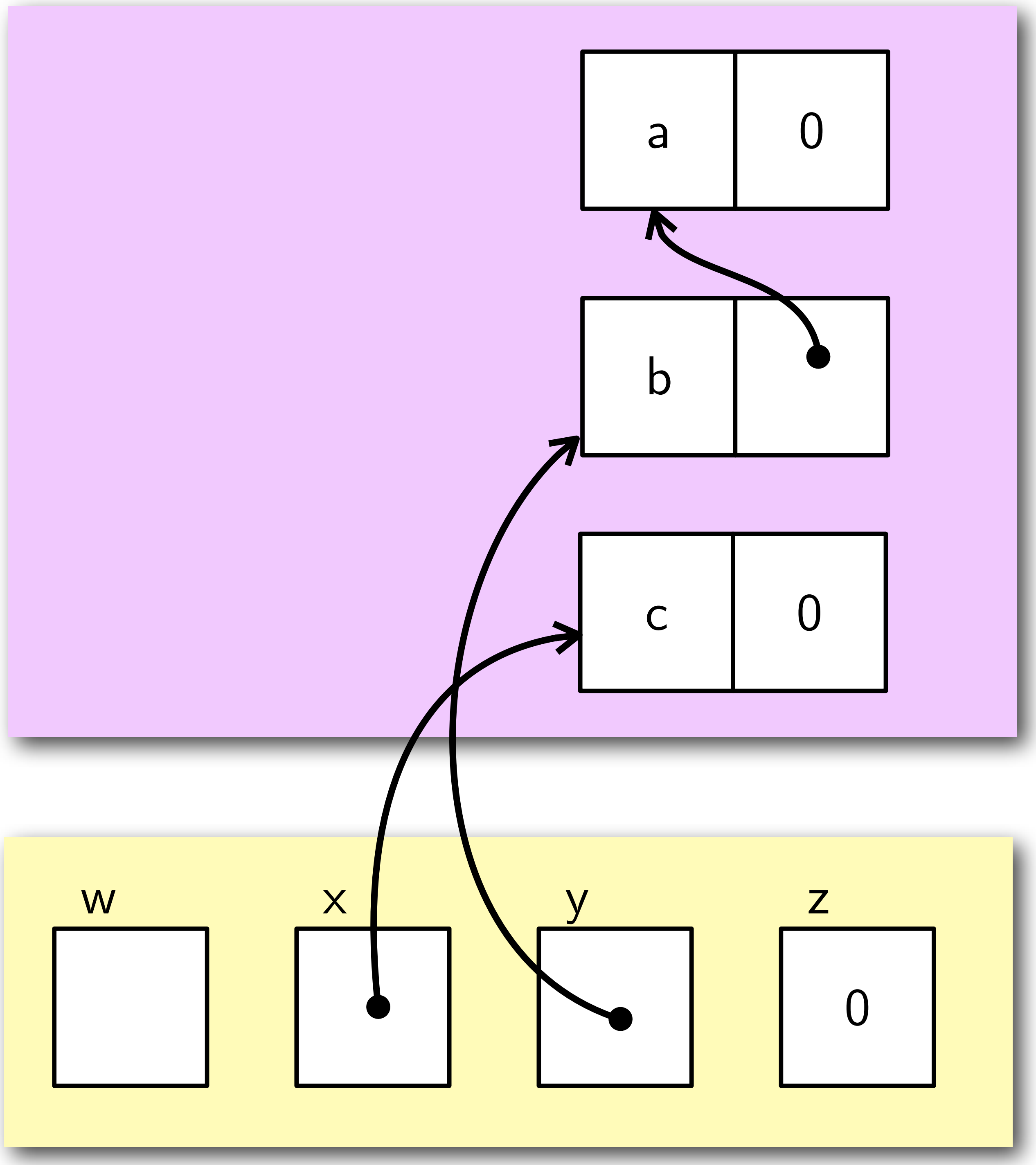


How `list_reverse` works

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



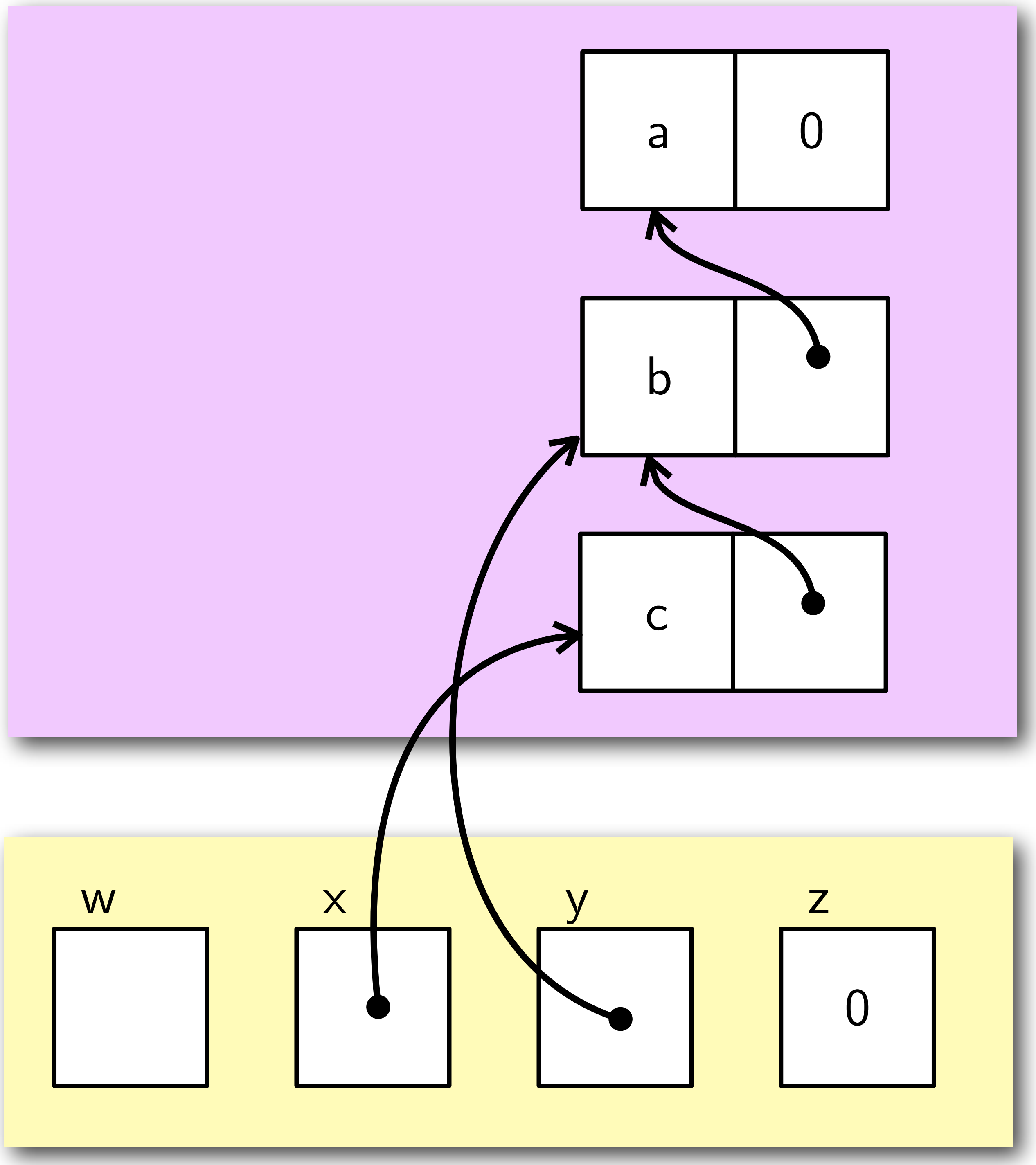
How `list_reverse` works



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

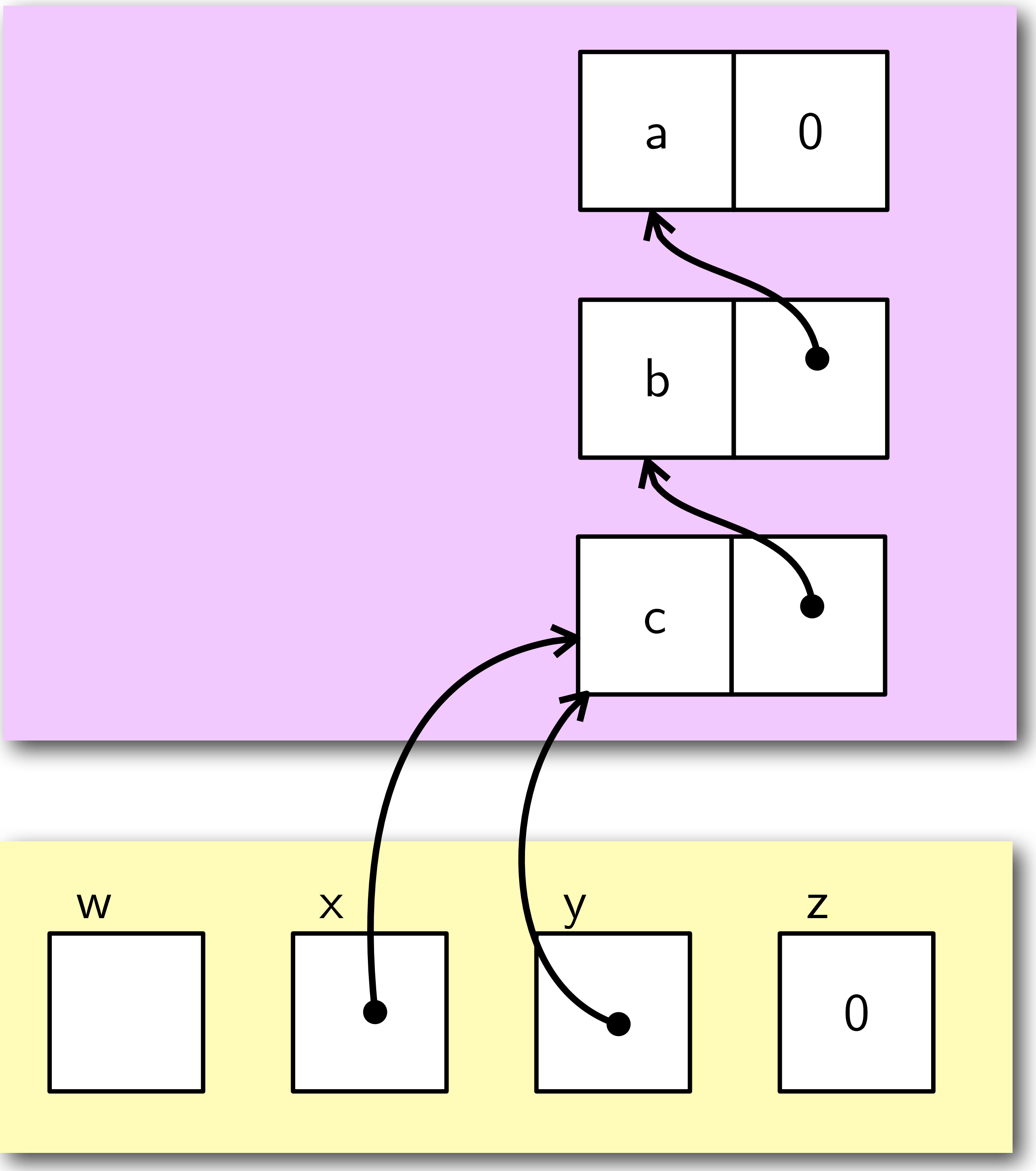
How `list_reverse` works

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



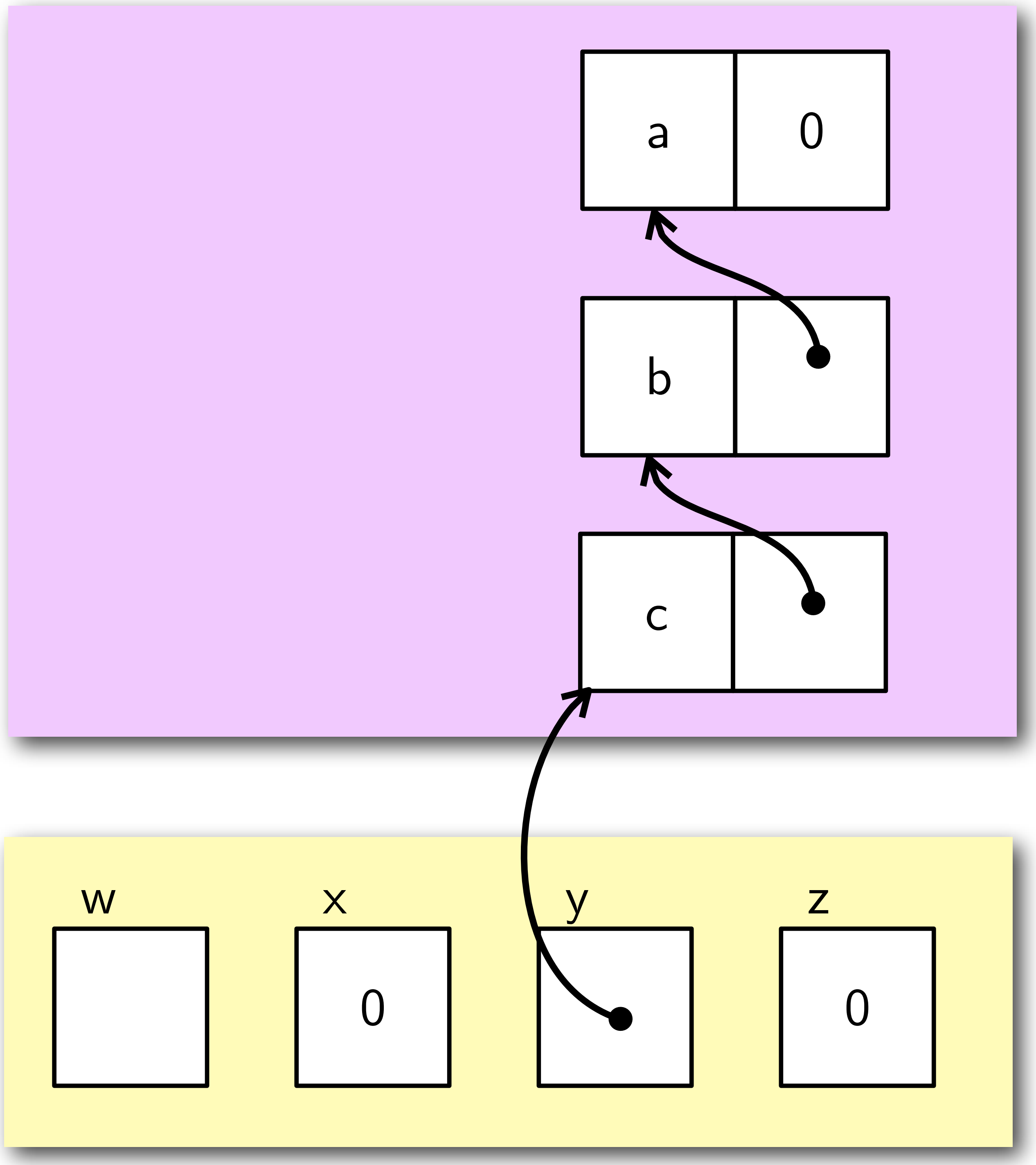
How `list_reverse` works

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



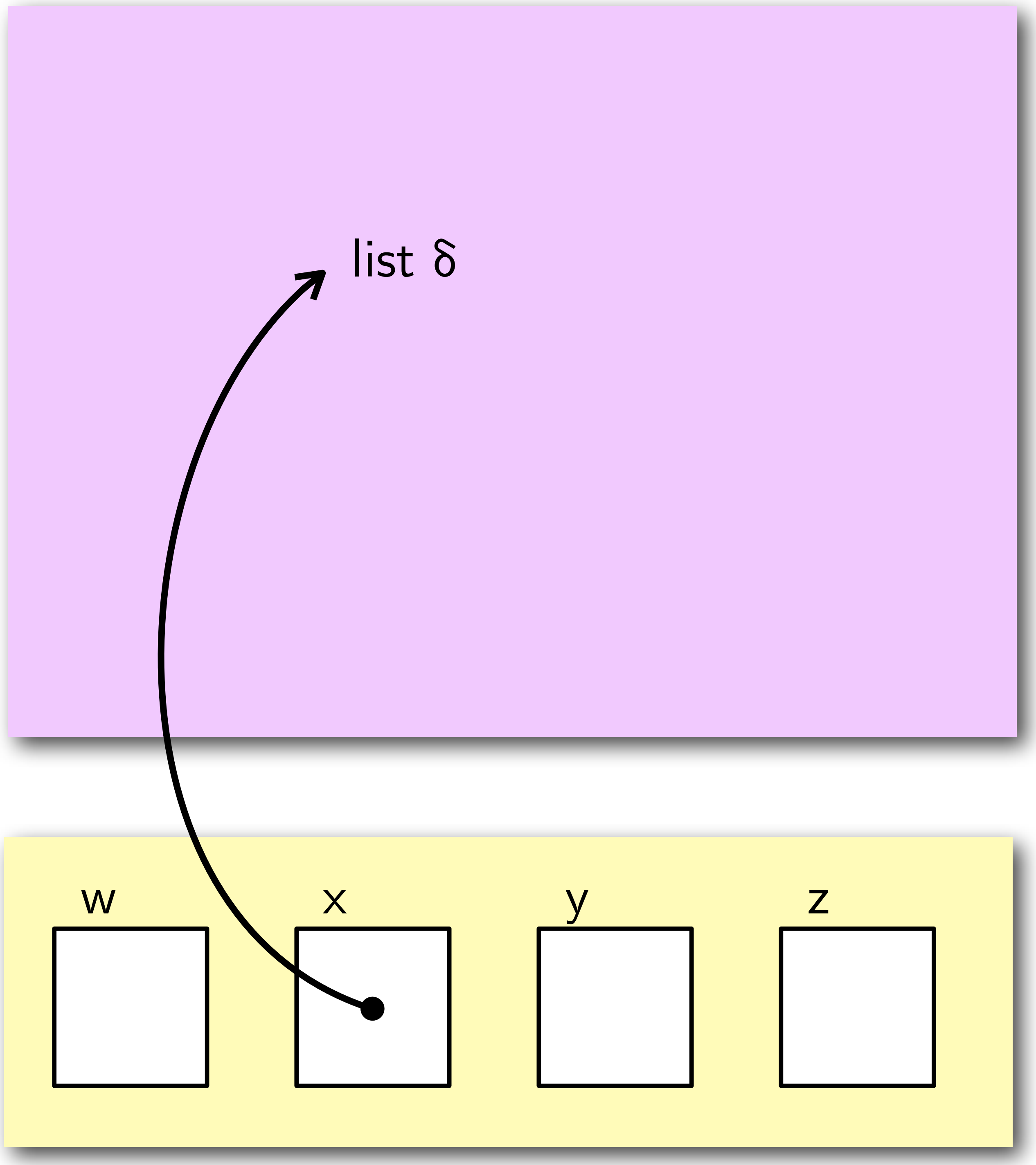
How `list_reverse` works

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



Proof of list reverse

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

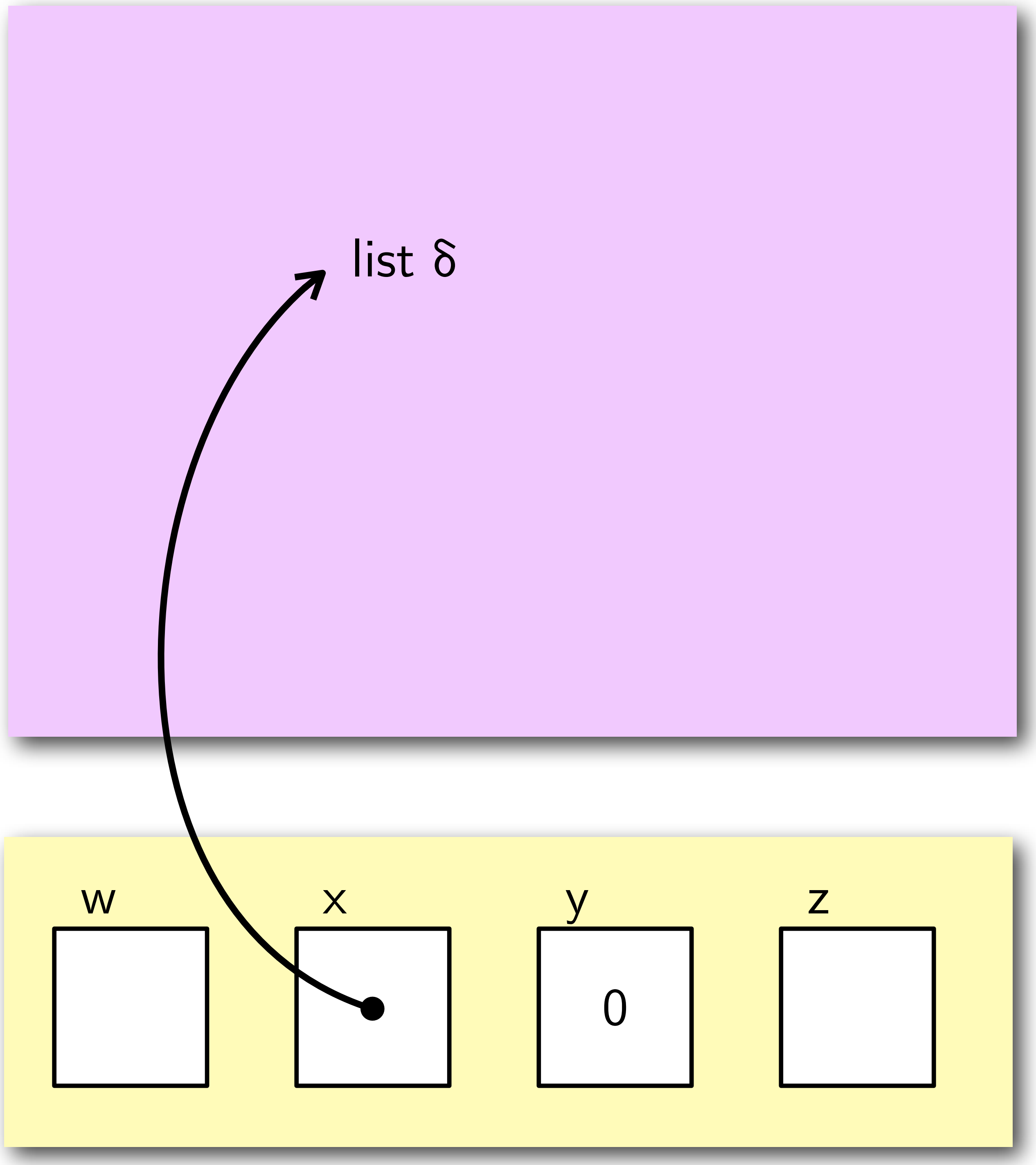


Proof of list reverse

{list δ x}

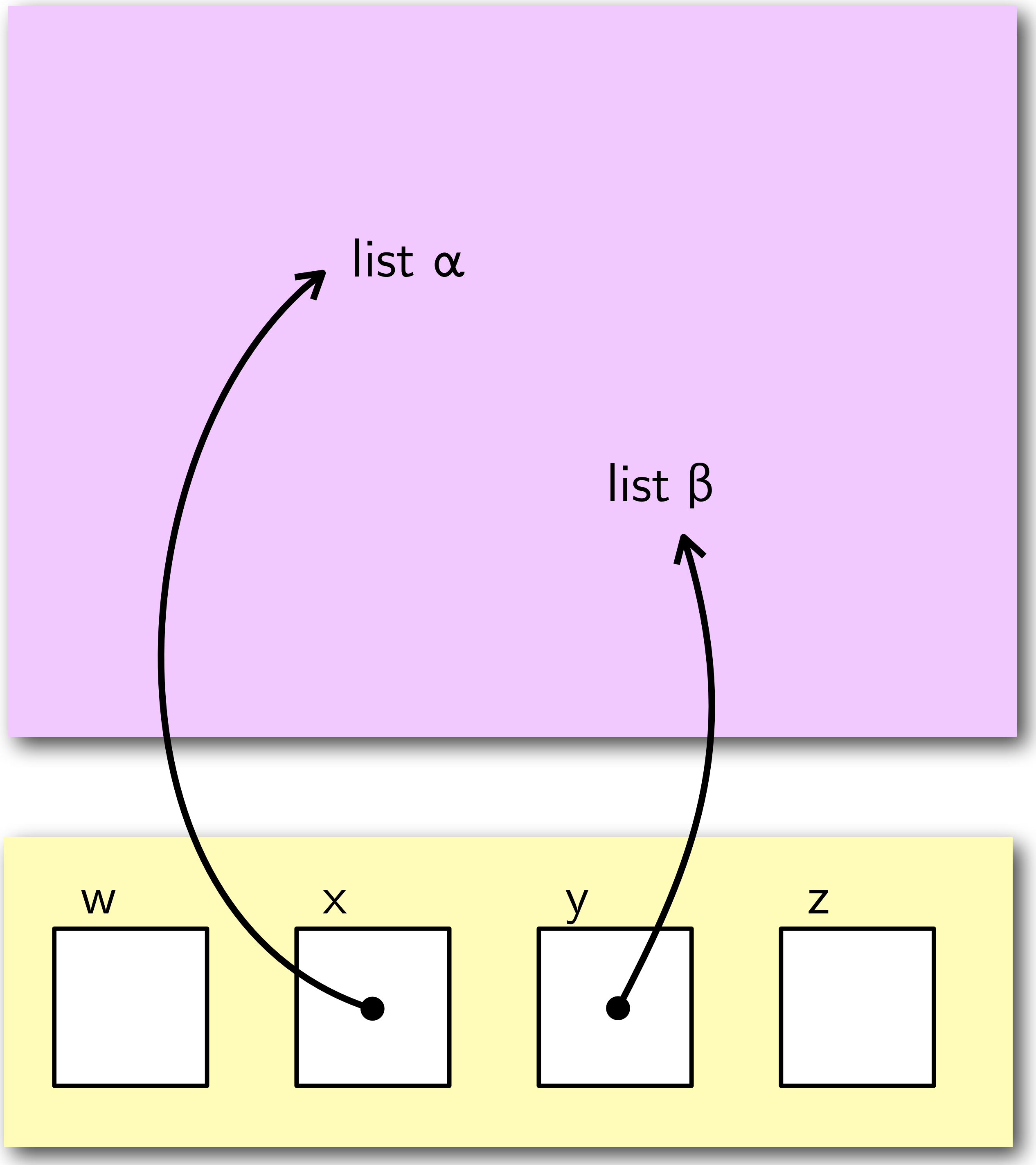
```
y := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}



Proof of list reverse

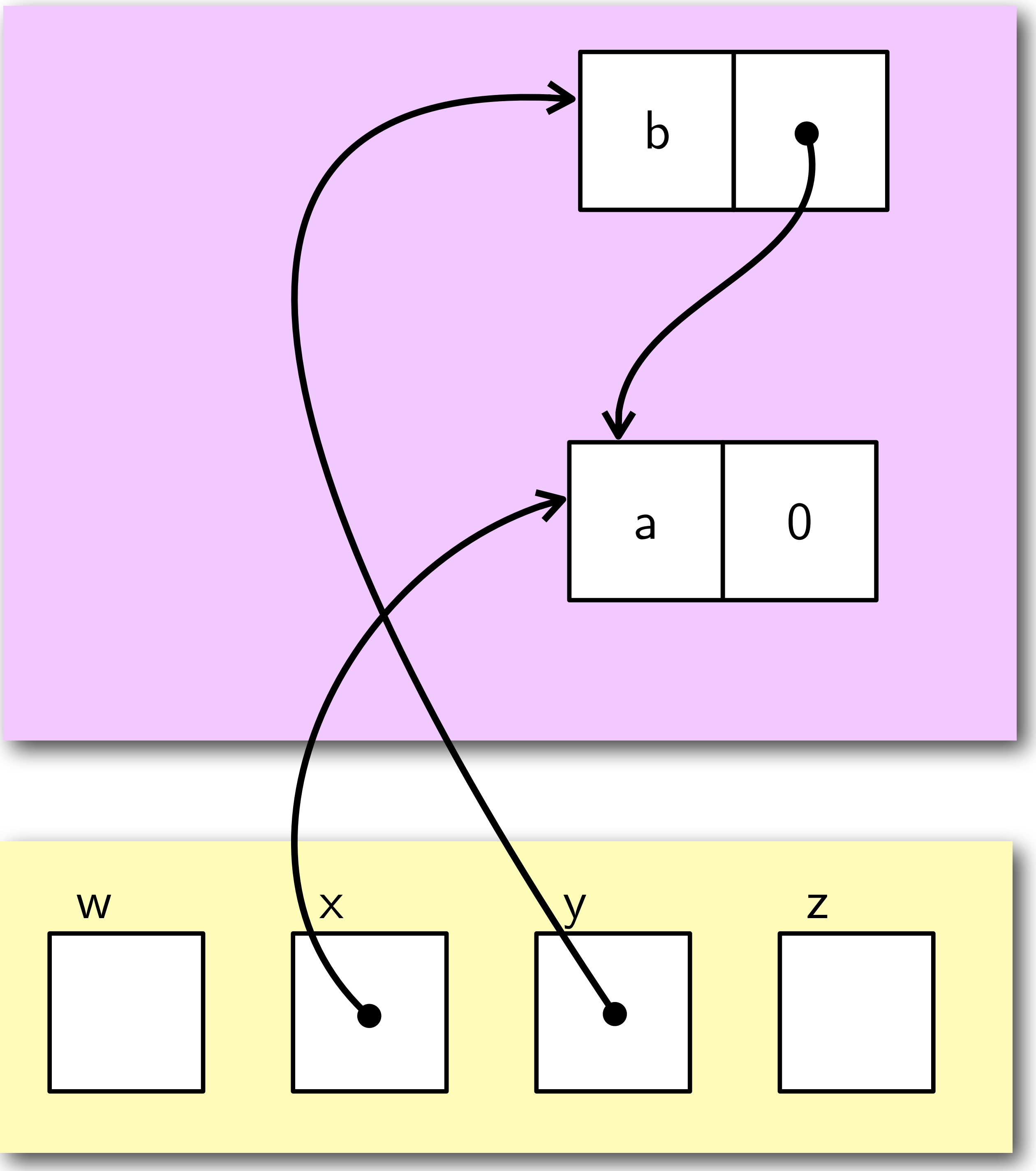
$\{$list δ x$\}$
y := 0;
$\{\exists\alpha,\beta.$ list α x $\wedge$ list β y $\wedge$ $\delta = -\beta\cdot\alpha\}$
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
$\{$list $-\delta$ y$\}$



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

Proof of list reverse

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



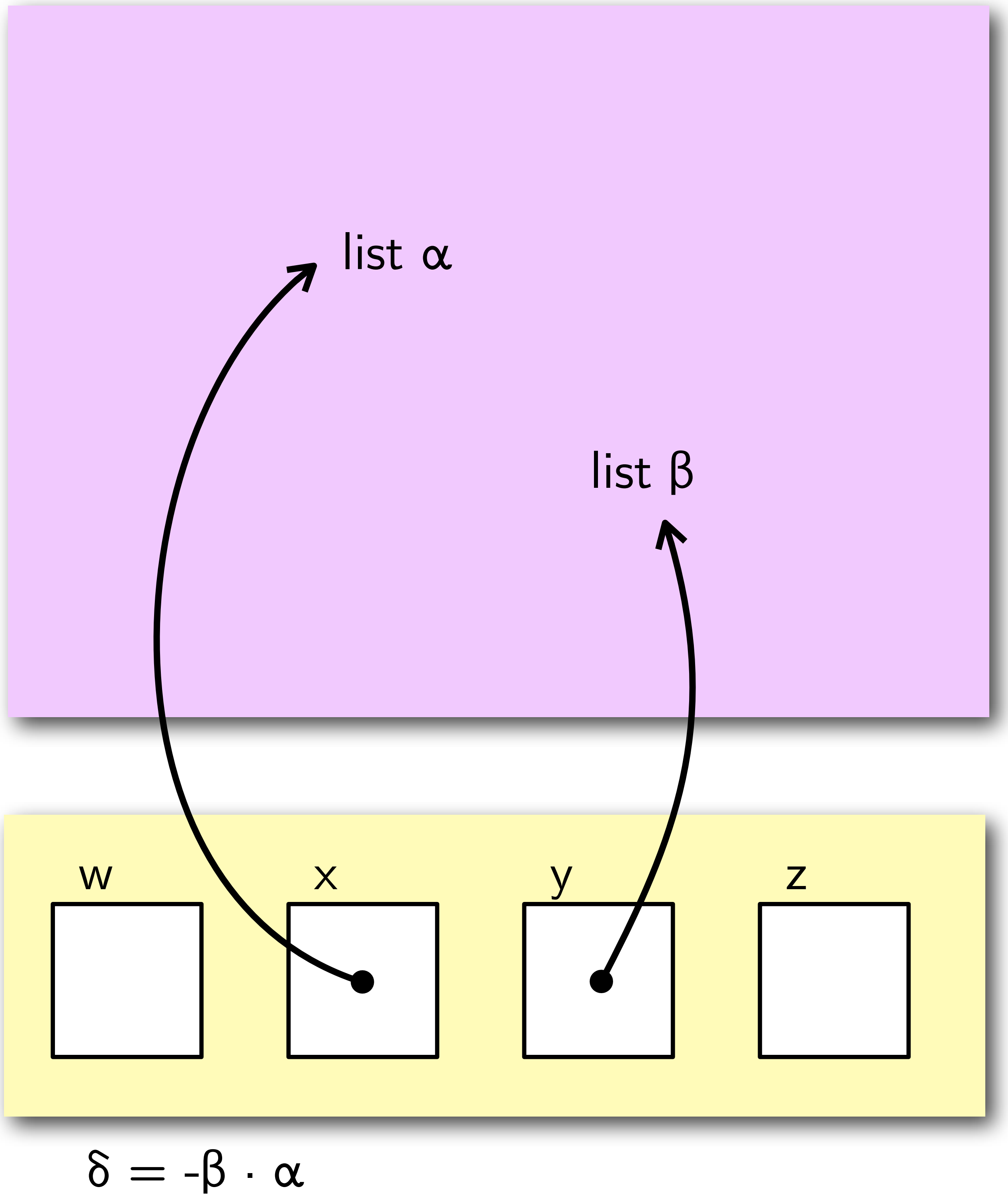
δ = -β · α

Proof of list reverse

{list δ x}
y := 0;
{∃α,β. list α x ∧ list β y ∧ δ = -β·α
∧ (∀z. reach(x,z) ∧ reach(y,z) ⇒ z=0)}

```
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}



Proof of list reverse

{list δ x}
list_reverse(x,y)
{list -δ y}

Proof of list reverse

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

Proof of list reverse

{list δ x $\wedge$ list ε w
$\wedge$ ($\forall$z. reach(x,z) $\wedge$ reach(w,z) $\Rightarrow$ z=0)}
list_reverse(x,y)
{list -δ y}

Proof of list reverse

{list δ x ∧ list ε w
∧ (∀z. reach(x,z) ∧ reach(w,z) ⇒ z=0)}

```
y := 0;
```

{∃α,β. list α x ∧ list β y ∧ δ = -β·α
∧ (∀z. reach(x,z) ∧ reach(y,z) ⇒ z=0)}

```
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}

Proof of list reverse

{list δ x ∧ list ε w
∧ (∀z. reach(x,z) ∧ reach(w,z) ⇒ z=0)}

```
y := 0;
```

{∃α,β. list α x ∧ list β y ∧ δ = -β·α
∧ (∀z. reach(x,z) ∧ reach(y,z) ⇒ z=0)
∧ list ε w
∧ (∀z. (reach(x,z) ∨ reach(y,z))
  ∧ reach(w,z) ⇒ z=0)}

```
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}

Proof of list reverse

{list δ x ∧ list ε w
∧ (∀z. reach(x,z) ∧ reach(w,z) ⇒ z=0)}

```
y := 0;
```

{∃α,β. list α x ∧ list β y ∧ δ = -β·α
∧ (∀z. reach(x,z) ∧ reach(y,z) ⇒ z=0)

∧ list ε w
∧ (∀z. (reach(x,z) ∨ reach(y,z))
∧ reach(w,z) ⇒ z=0)}

```
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y ∧ list ε w
∧ (∀z. reach(x,z) ∧ reach(w,z) ⇒ z=0)}

Proof of list reverse

{list δ x ∧ list ε w
∧ (∀z. reach(x,z) ∧ reach(w,z) ⇒ z=0)}

list_reverse(x,y)

{list -δ y ∧ list ε w
∧ (∀z. reach(x,z) ∧ reach(w,z) ⇒ z=0)}}

20th century proof

Summary:

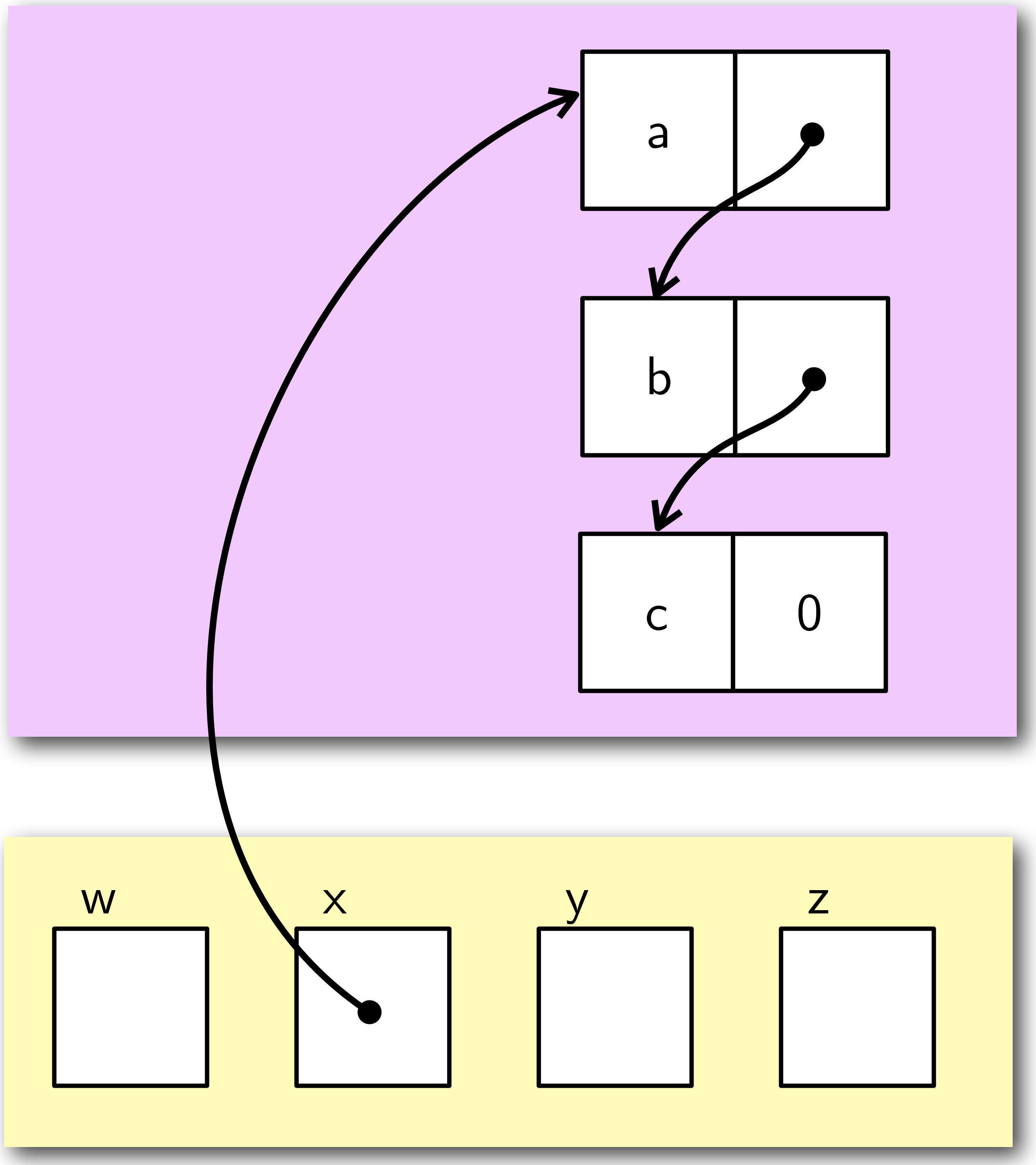
Without separation logic, proofs are **fiddly** and **not modular**, but they **can be done**.

Lecture Plan

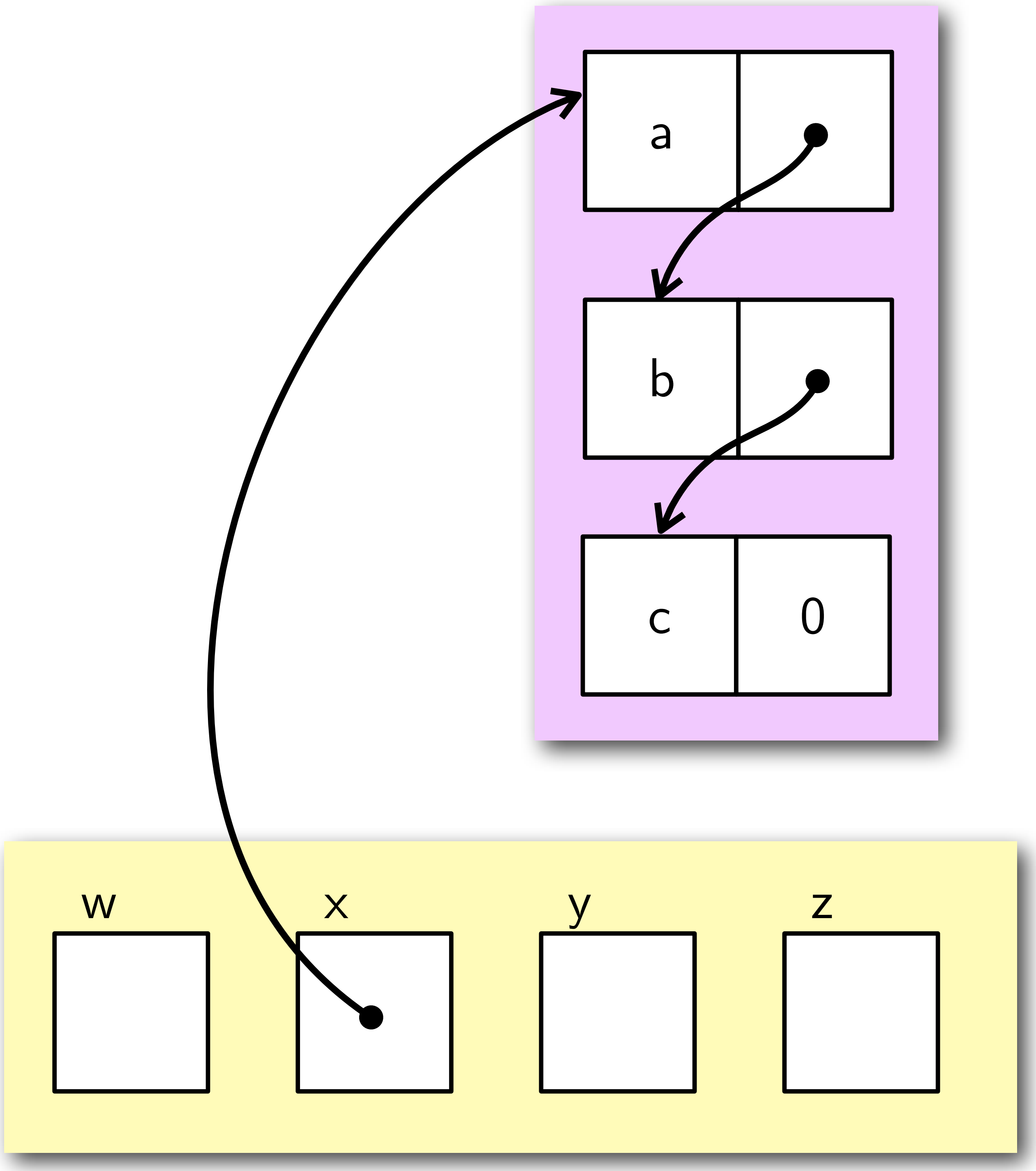
- A 20th century proof of `list_reverse`
- A proof of `list_reverse` in separation logic
- Separation logic's proof rules
- Soundness of the Frame rule

Proof of list reverse

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



Proof of list reverse

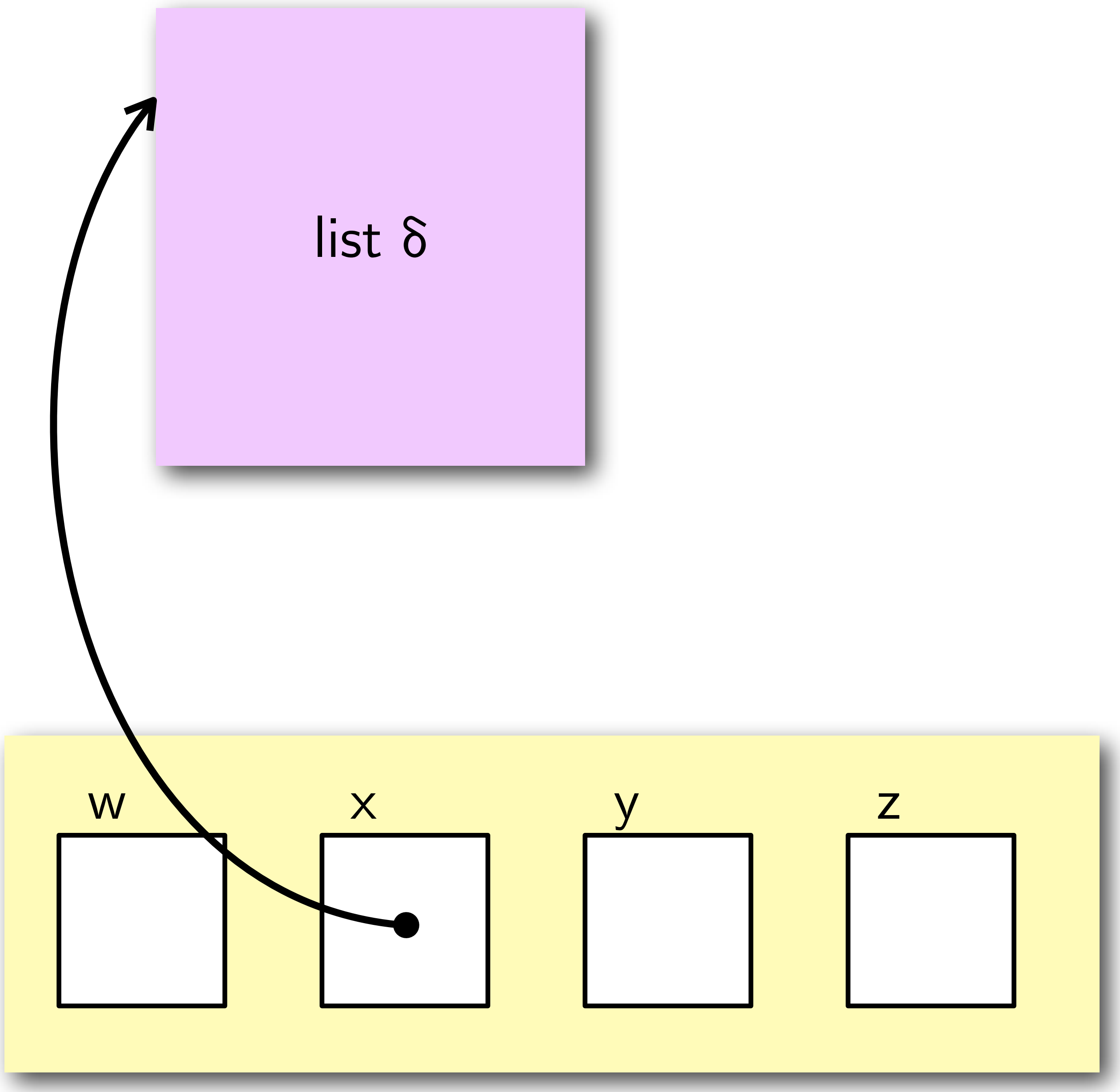


{list δ x}

```
y := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}

Proof of list reverse



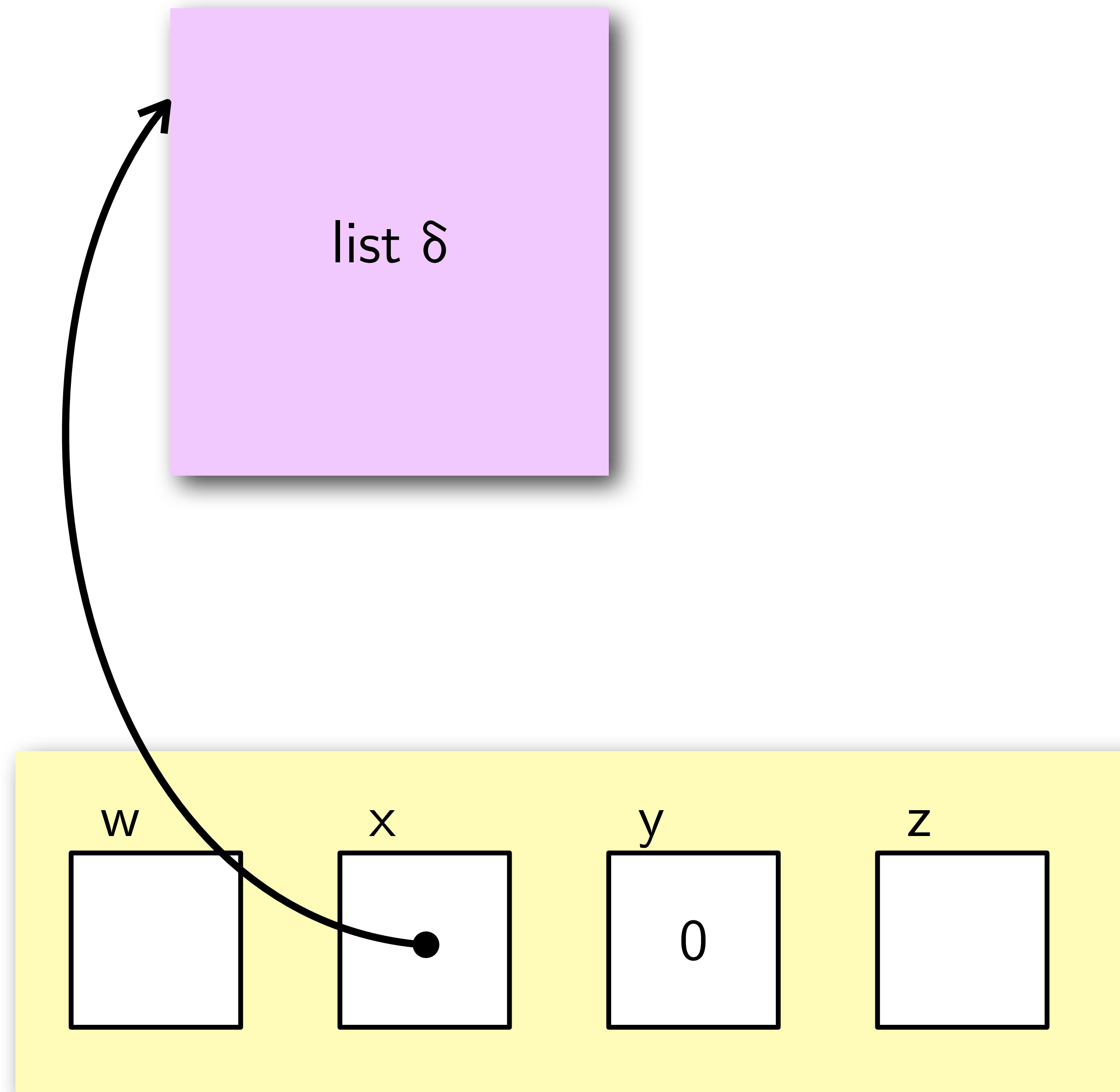
{list δ x}

```
y := 0;
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

{list -δ y}

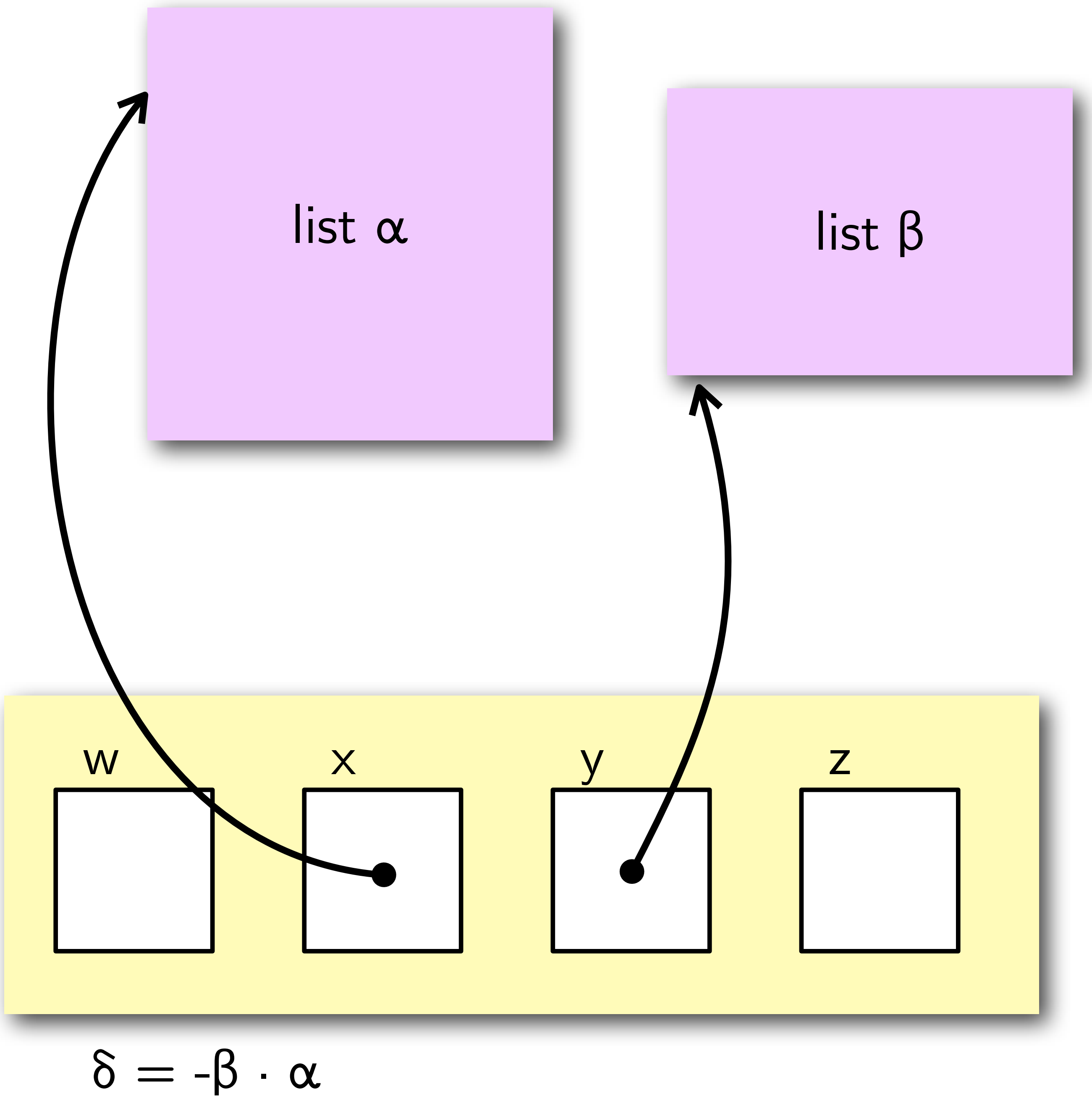
Proof of list reverse

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



Proof of list reverse

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



Proof of list reverse

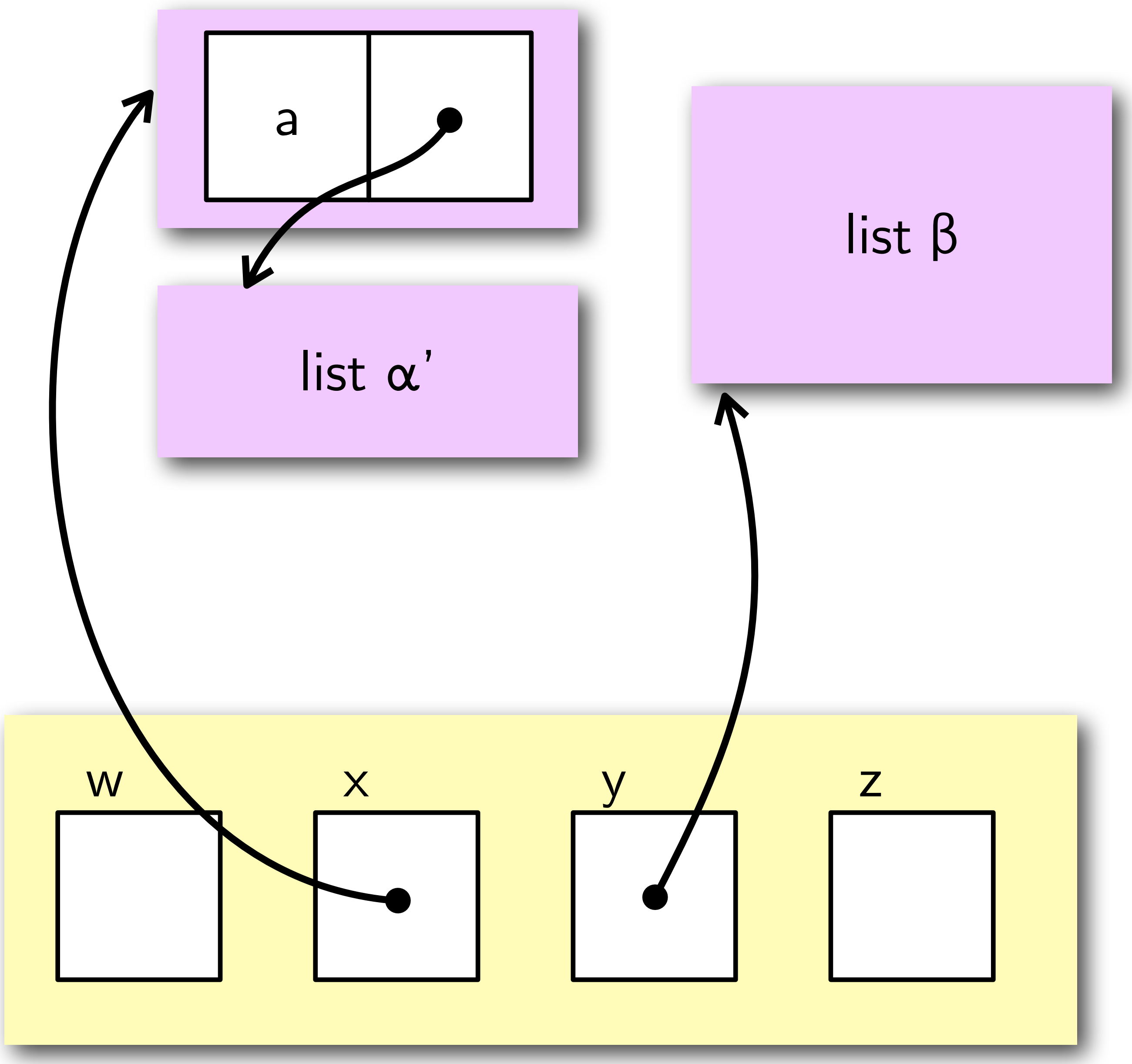
$\{\text{list } \delta \text{ x}\}$

y := 0;

$\{\exists\alpha,\beta.\ \text{list } \alpha \text{ x} * \text{list } \beta \text{ y} * \delta \doteq -\beta\cdot\alpha\}$

```
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

$\{\text{list } -\delta \text{ y}\}$



$\delta = -\beta \cdot (a \cdot \alpha')$

Proof of list reverse

{list δ x}

y := 0;

{∃α,β. list α x ∗ list β y ∗ δ ≐ -β·α}

while (x≠0) do {

    z := [x+1];

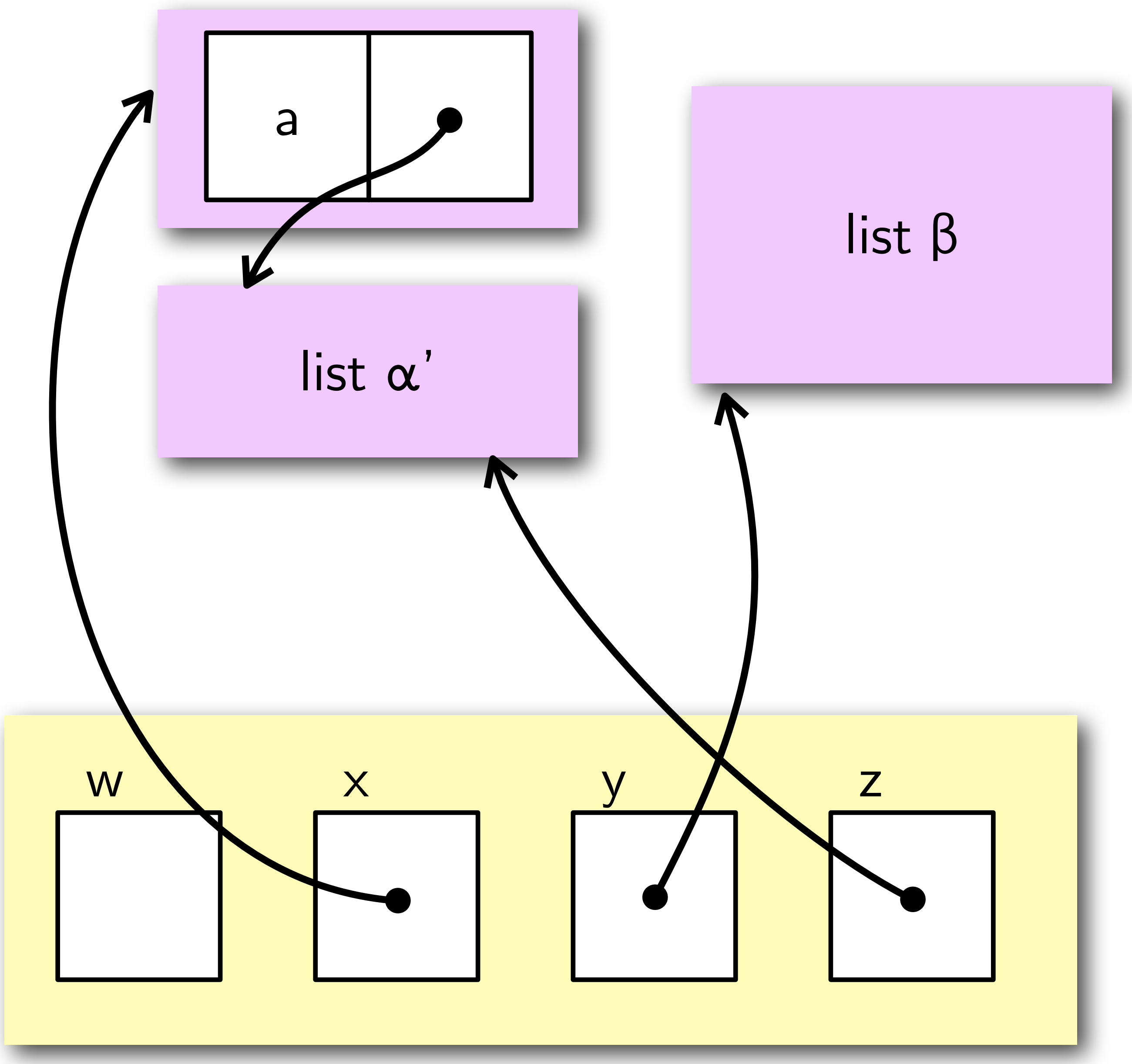
    [x+1] := y;

    y := x;

    x := z;

}

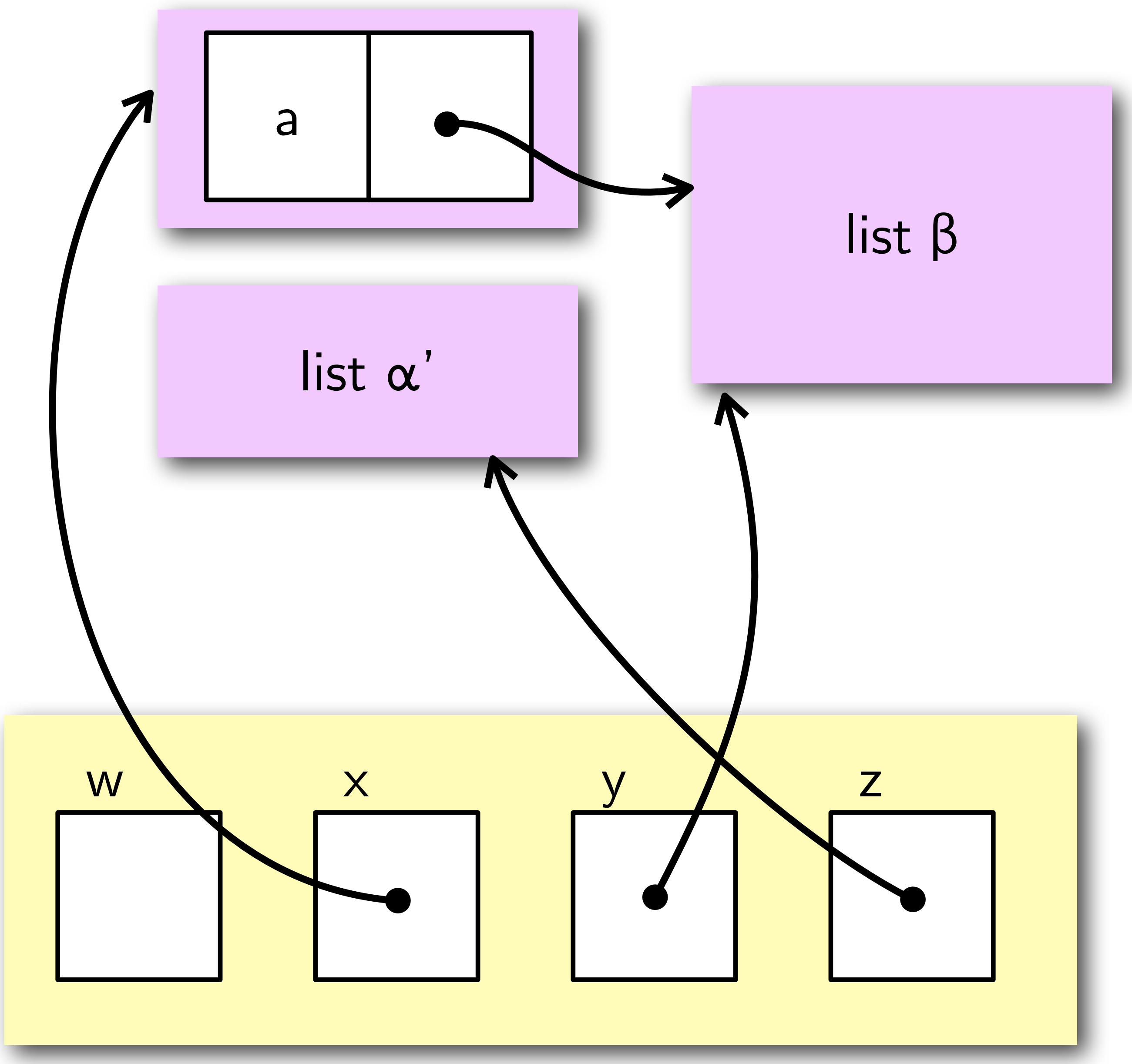
{list -δ y}



δ = -β · (a · α')

Proof of list reverse

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



δ = -β · (a · α’)

Proof of list reverse

$\{\text{list } \delta\ x\}$

y := 0;

$\{\exists\alpha,\beta.\ \text{list } \alpha\ x * \text{list } \beta\ y * \delta \doteq -\beta\cdot\alpha\}$

while (x≠0) do {

    z := [x+1];

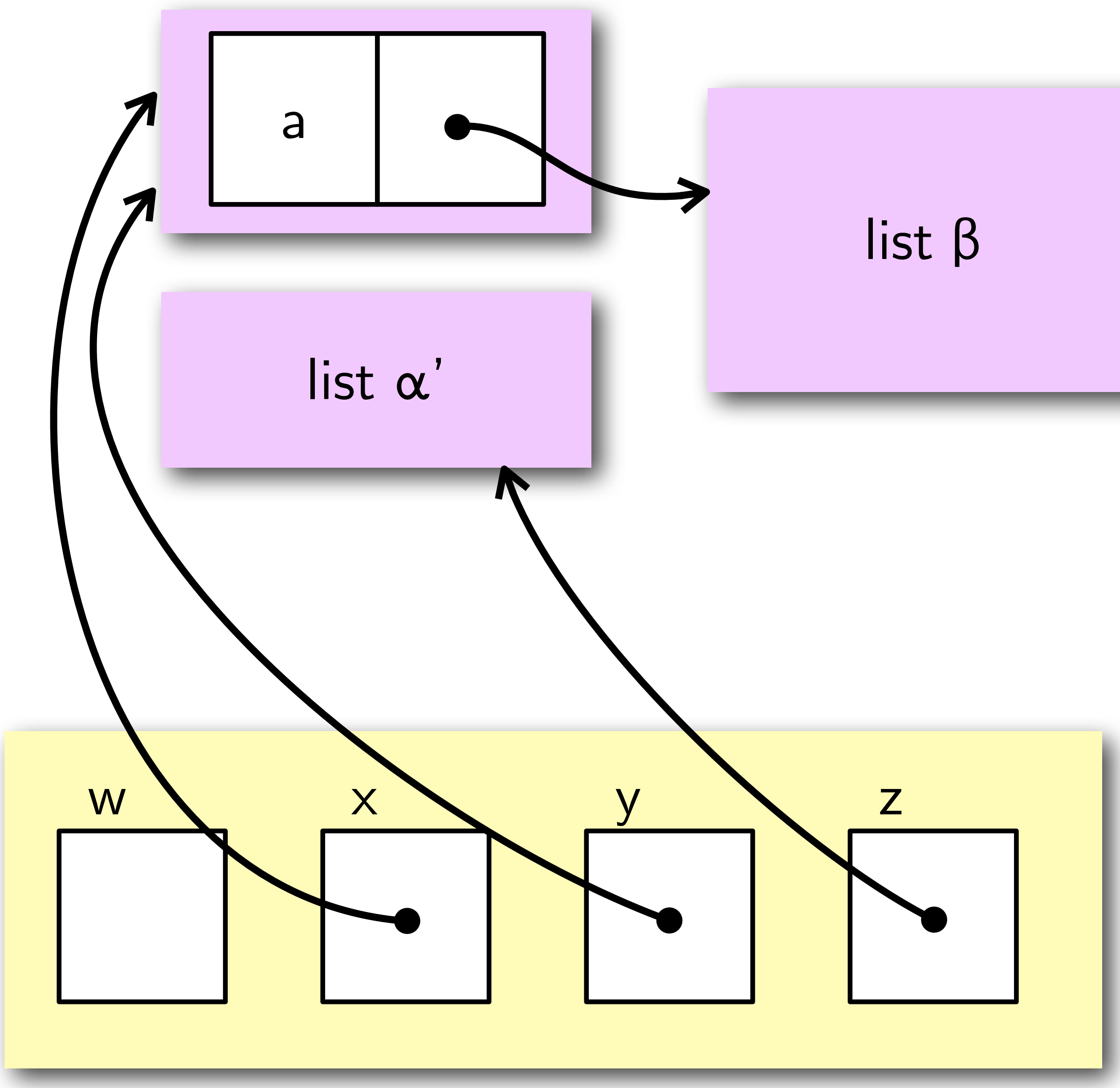
    [x+1] := y;

    y := x;

    x := z;

}

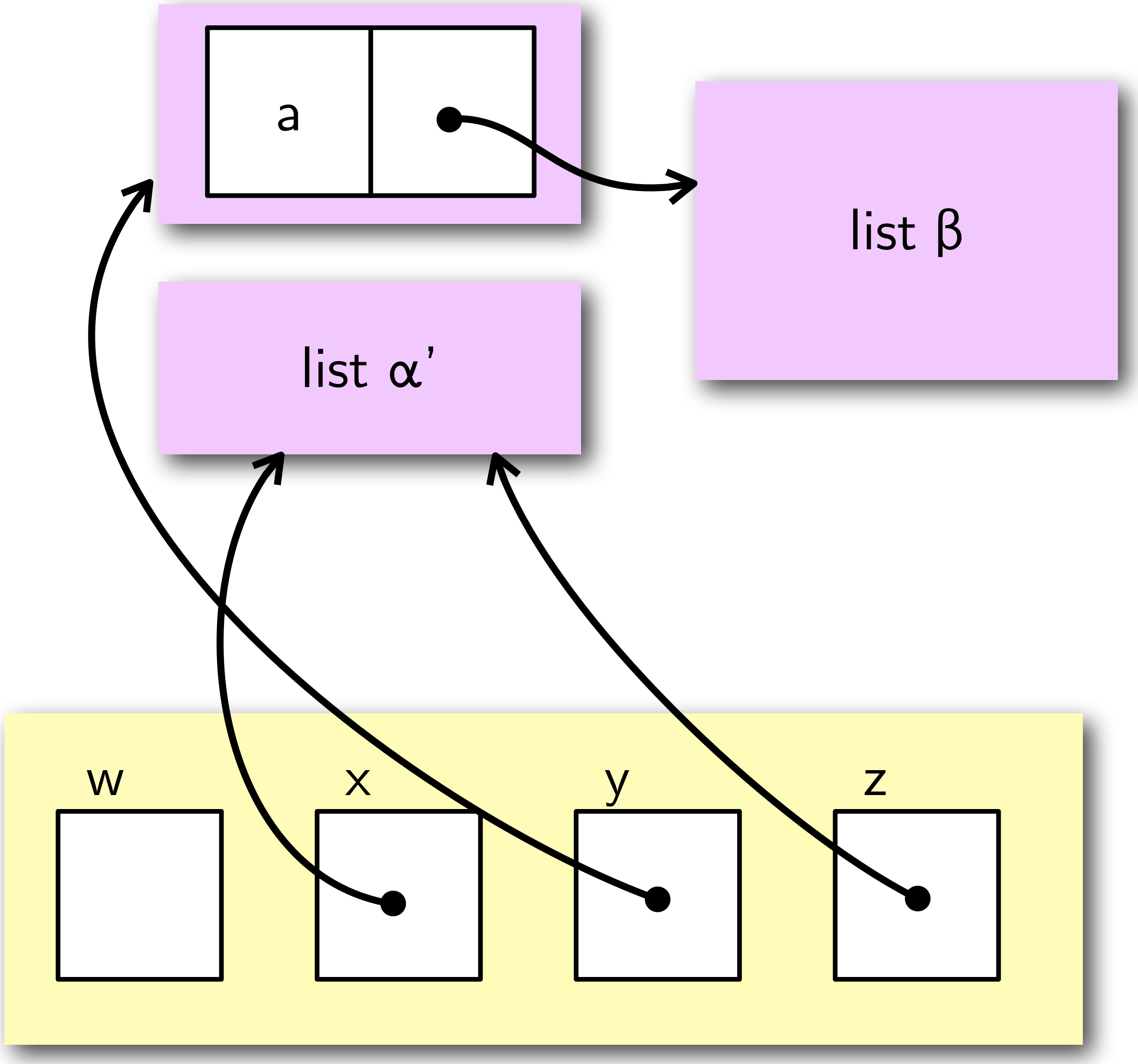
$\{\text{list } -\delta\ y\}$



$\delta = -\beta \cdot (a \cdot \alpha')$

Proof of list reverse

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



$\delta = -\beta \cdot (a \cdot \alpha')$

Proof of list reverse

{list δ x}

y := 0;

{∃α,β. list α x ∗ list β y ∗ δ ≐ -β·α}

while (x≠0) do {

    z := [x+1];

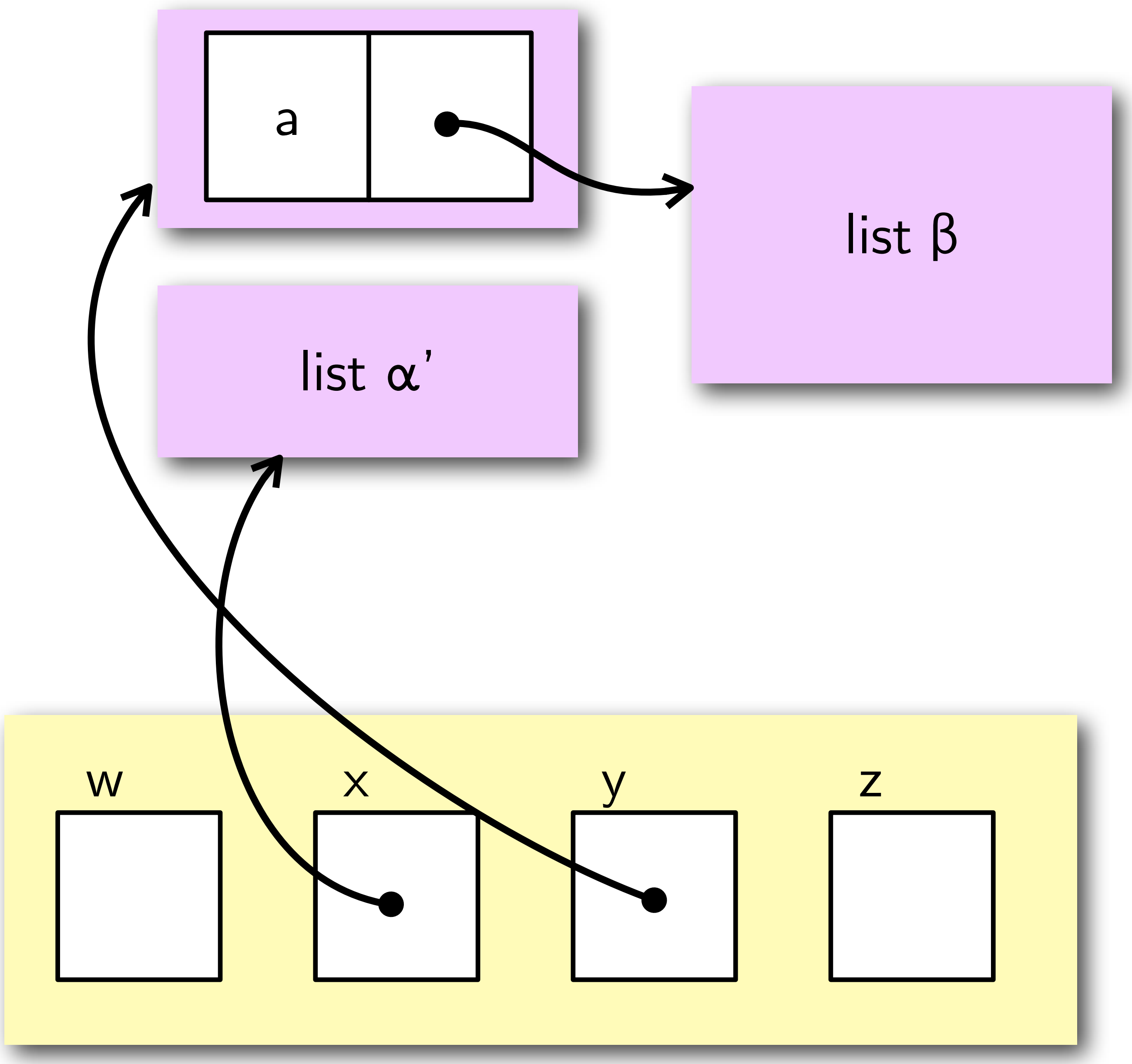
    [x+1] := y;

    y := x;

    x := z;

}

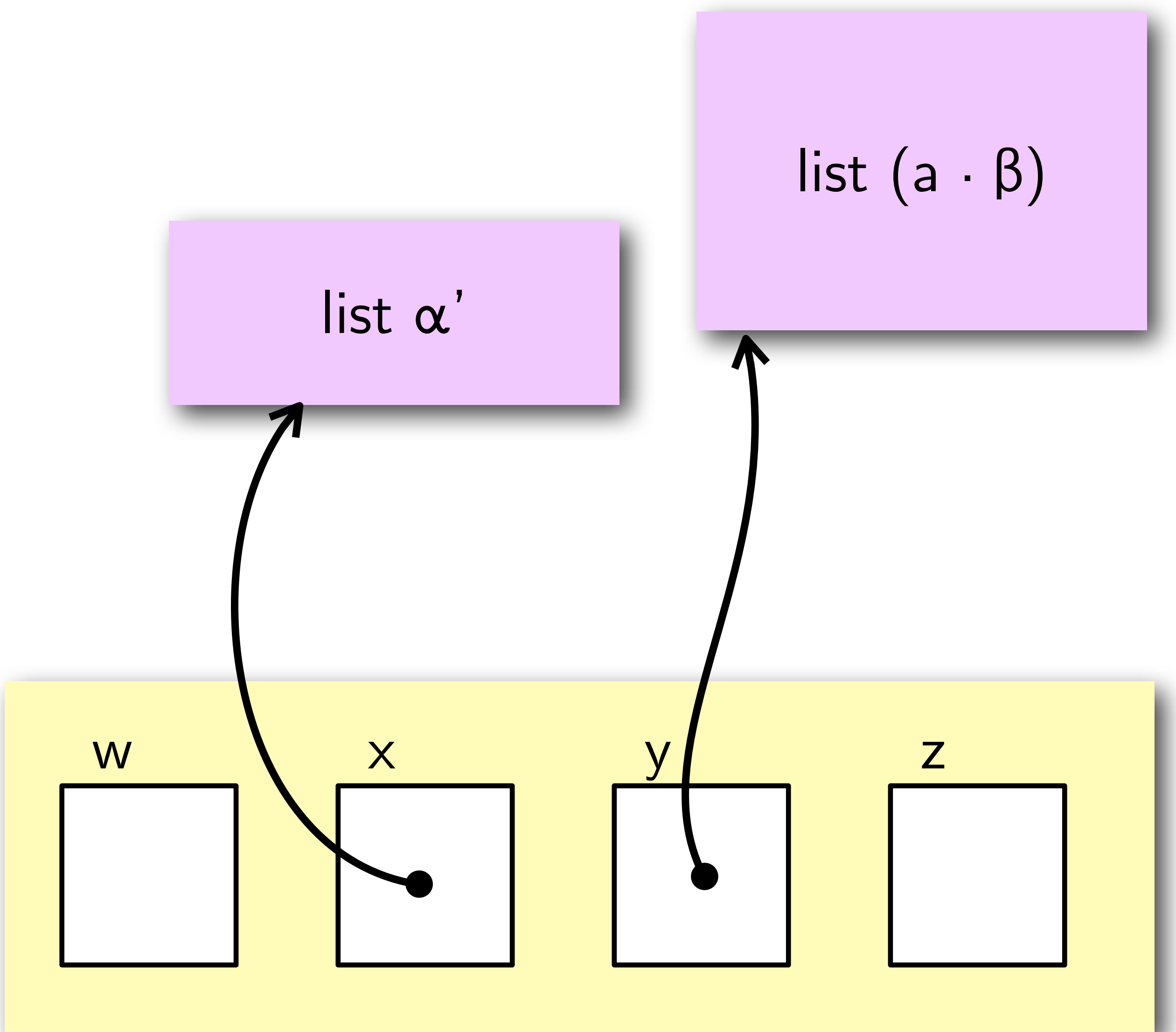
{list -δ y}



$\delta = -\beta \cdot (a \cdot \alpha')$

Proof of list reverse

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



$\delta = -\beta \cdot (a \cdot \alpha')$

Proof of list reverse

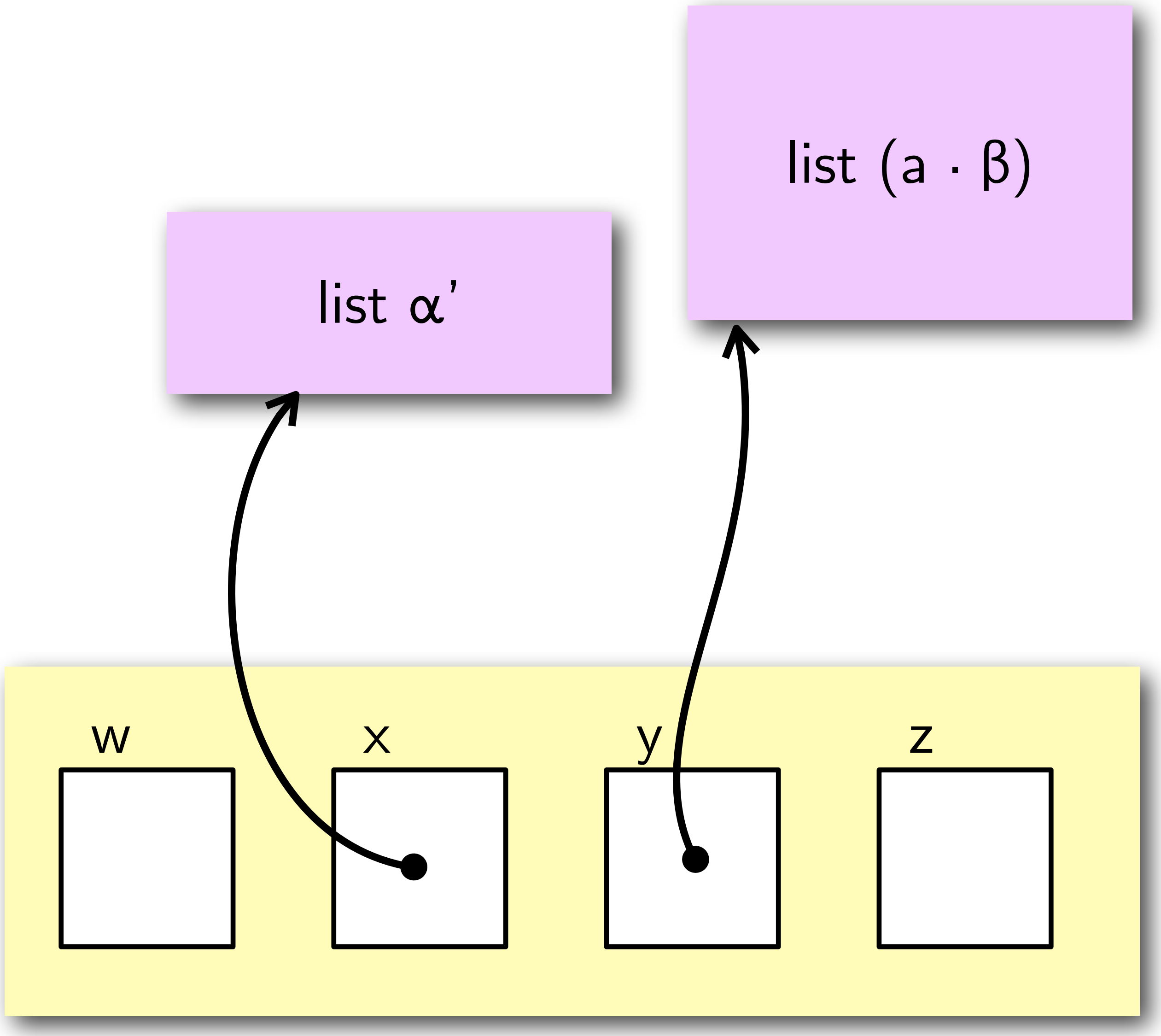
$\{\text{list } \delta \text{ x}\}$

y := 0;

$\{\exists\alpha,\beta.\ \text{list } \alpha \text{ x} * \text{list } \beta \text{ y} * \delta \doteq \text{-}\beta\cdot\alpha\}$

```
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

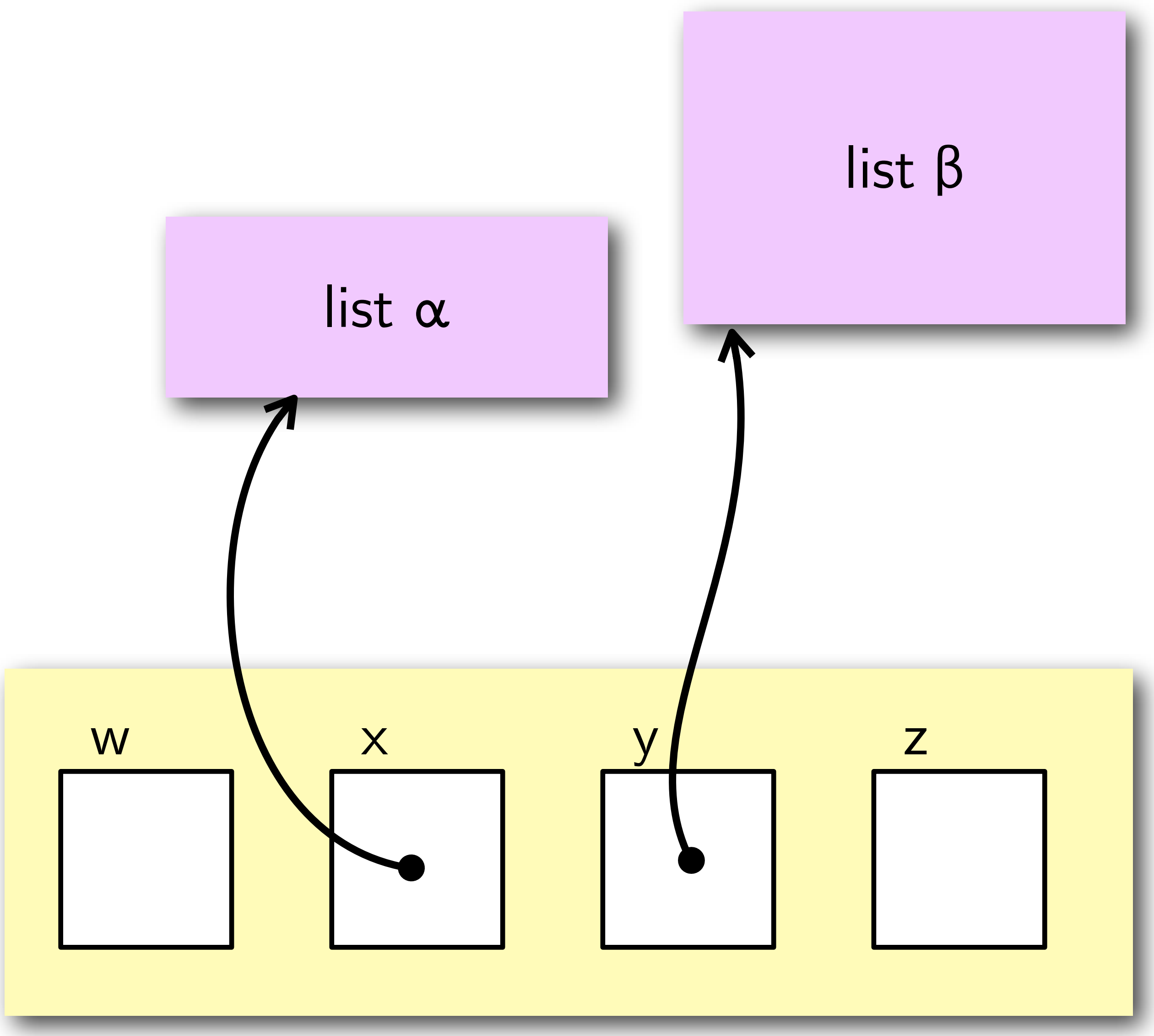
$\{\text{list -}\delta \text{ y}\}$



$\delta = \text{-}(a \cdot \beta) \cdot \alpha'$

Proof of list reverse

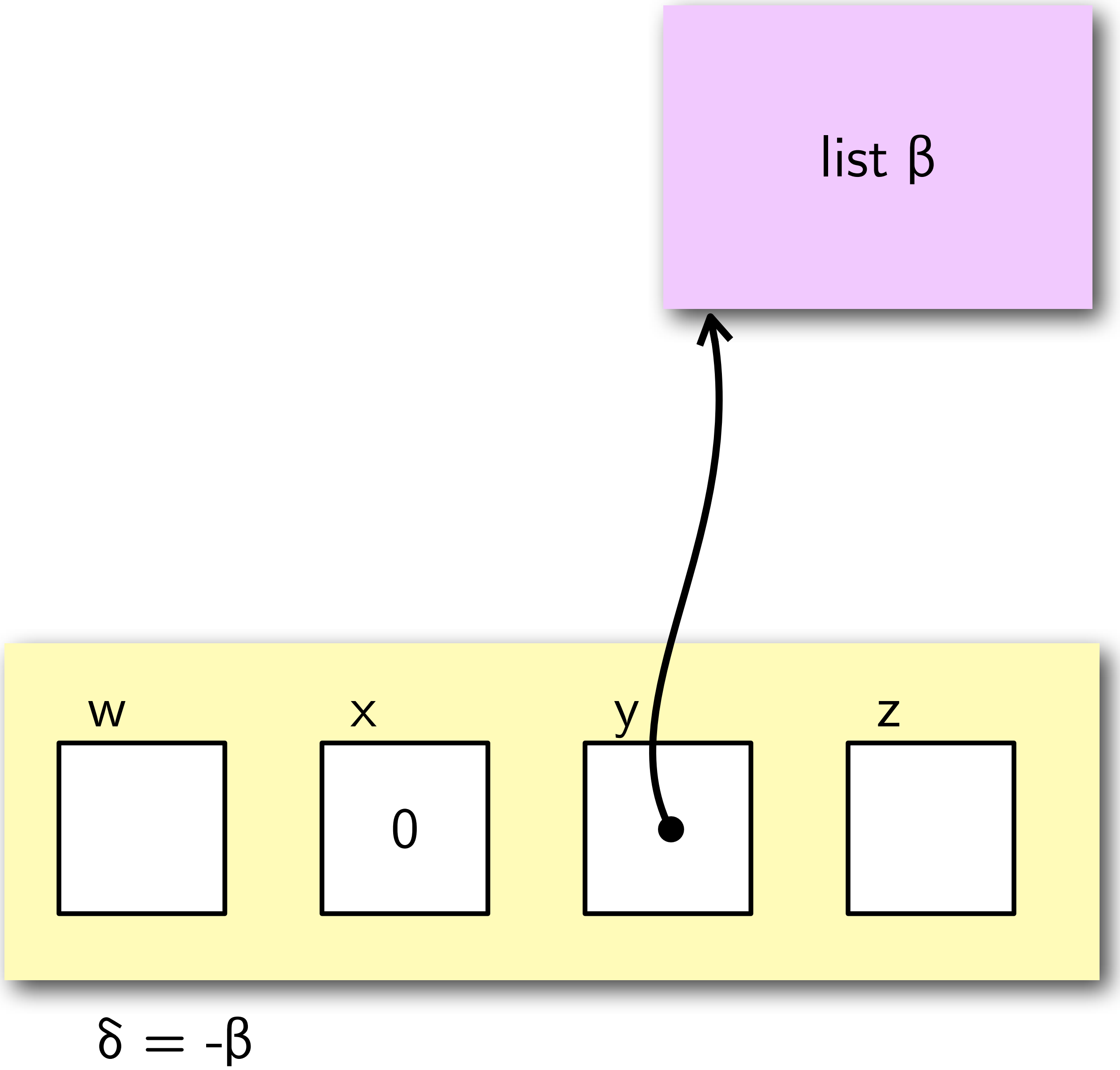
{list δ x}
y := 0;
{∃α,β. list α x ∗ list β y ∗ δ ≐ -β·α}
while (x≠0) do {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
{list -δ y}



δ = -β · α

Proof of list reverse

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



Proof of list reverse

$\{\text{list } \delta \ x\}$
y := 0;
while $\{\exists\alpha,\beta.\ \text{list } \alpha \ x * \text{list } \beta \ y * \delta \doteq \text{-}\beta{\cdot}\alpha\}$ (x≠0) do {
  $\{\exists a,\alpha,\beta,Z.\ x \mapsto a,Z * \text{list } \alpha \ Z * \text{list } \beta \ y * \delta \doteq \text{-}\beta{\cdot}a{\cdot}\alpha\}$
  z := [x+1];
  $\{\exists a,\alpha,\beta.\ x \mapsto a,z * \text{list } \alpha \ z * \text{list } \beta \ y * \delta \doteq \text{-}\beta{\cdot}a{\cdot}\alpha\}$
  [x+1] := y;
  $\{\exists a,\alpha,\beta.\ x \mapsto a,y * \text{list } \alpha \ z * \text{list } \beta \ y * \delta \doteq \text{-}\beta{\cdot}a{\cdot}\alpha\}$
  $\{\exists\alpha,\beta.\ \text{list } \alpha \ z * \text{list } \beta \ x * \delta \doteq \text{-}\beta{\cdot}\alpha\}$
  y := x; x := z;
  $\{\exists\alpha,\beta.\ \text{list } \alpha \ x * \text{list } \beta \ y * \delta \doteq \text{-}\beta{\cdot}\alpha\}$
}
$\{\text{list } \text{-}\delta \ y\}$

Proof of list reverse

{list δ x}
list_reverse(x,y)
{list -δ y}

Proof of list reverse

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

Proof of list reverse

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

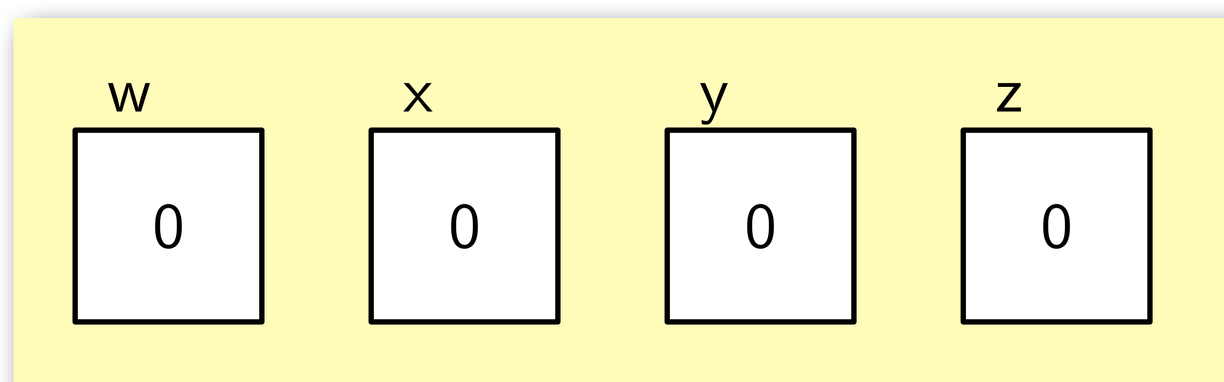
$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ (\dagger)$$

[†]provided R doesn't mention any variable modified by C

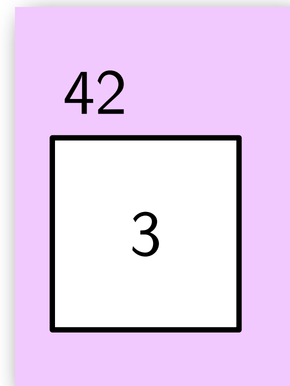
Lecture Plan

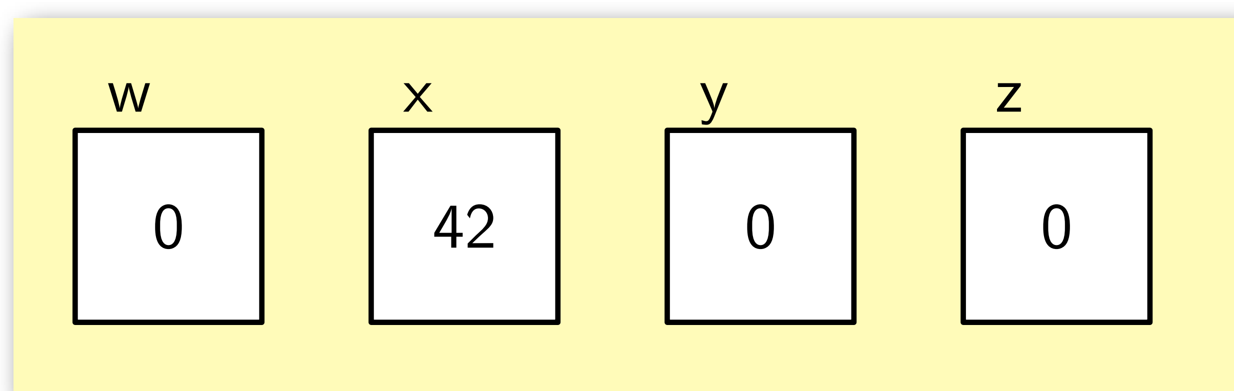
- A 20th century proof of `list_reverse`
- A proof of `list_reverse` in separation logic
- Separation logic's proof rules
- Soundness of the Frame rule

```
⇒ x := cons(3);
  y := cons(5,1);
  x := [x];
  [y+1] := 6;
  dispose(y);
```

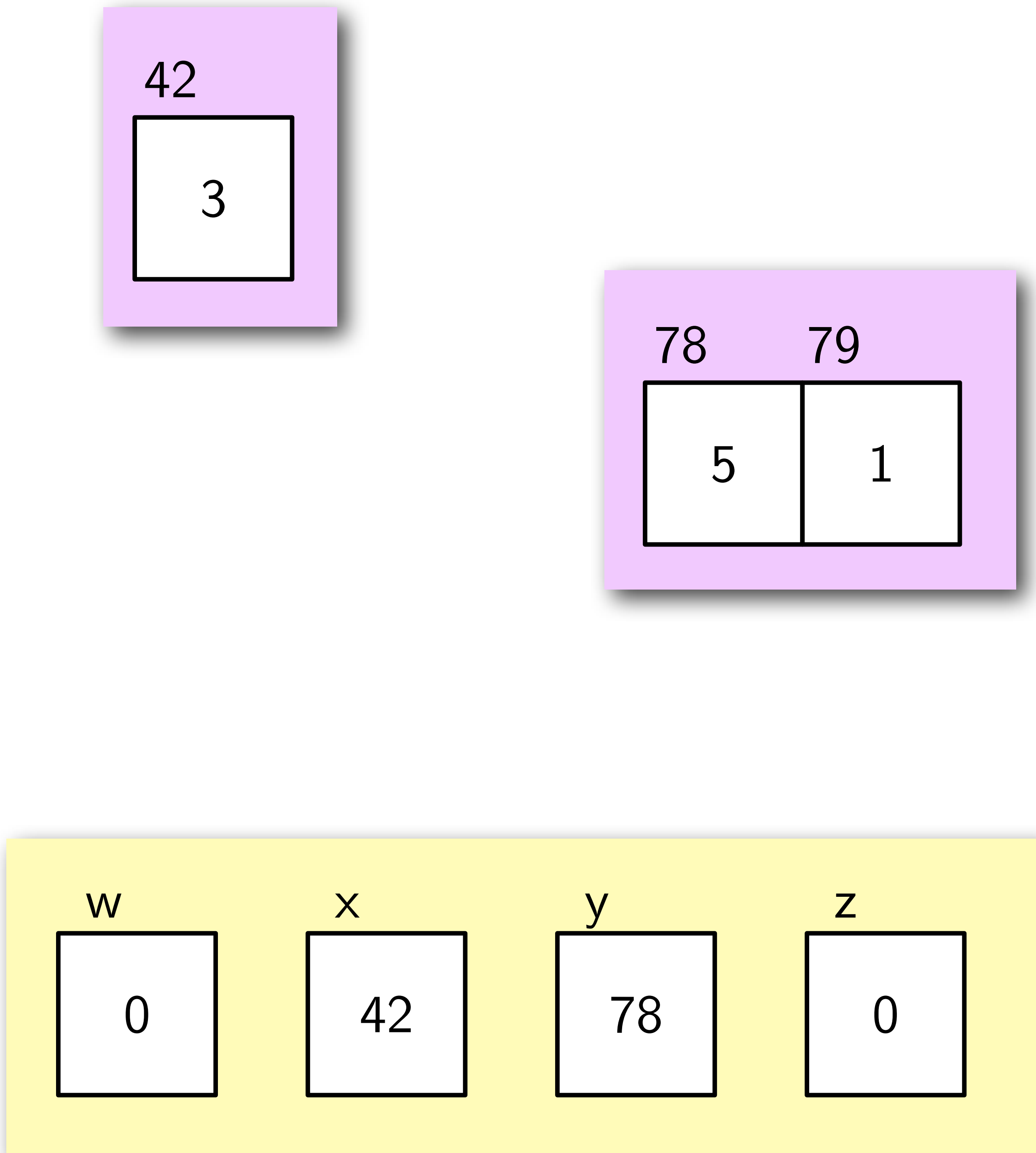


```
x := cons(3);
y := cons(5,1);
x := [x];
[y+1] := 6;
dispose(y);
```

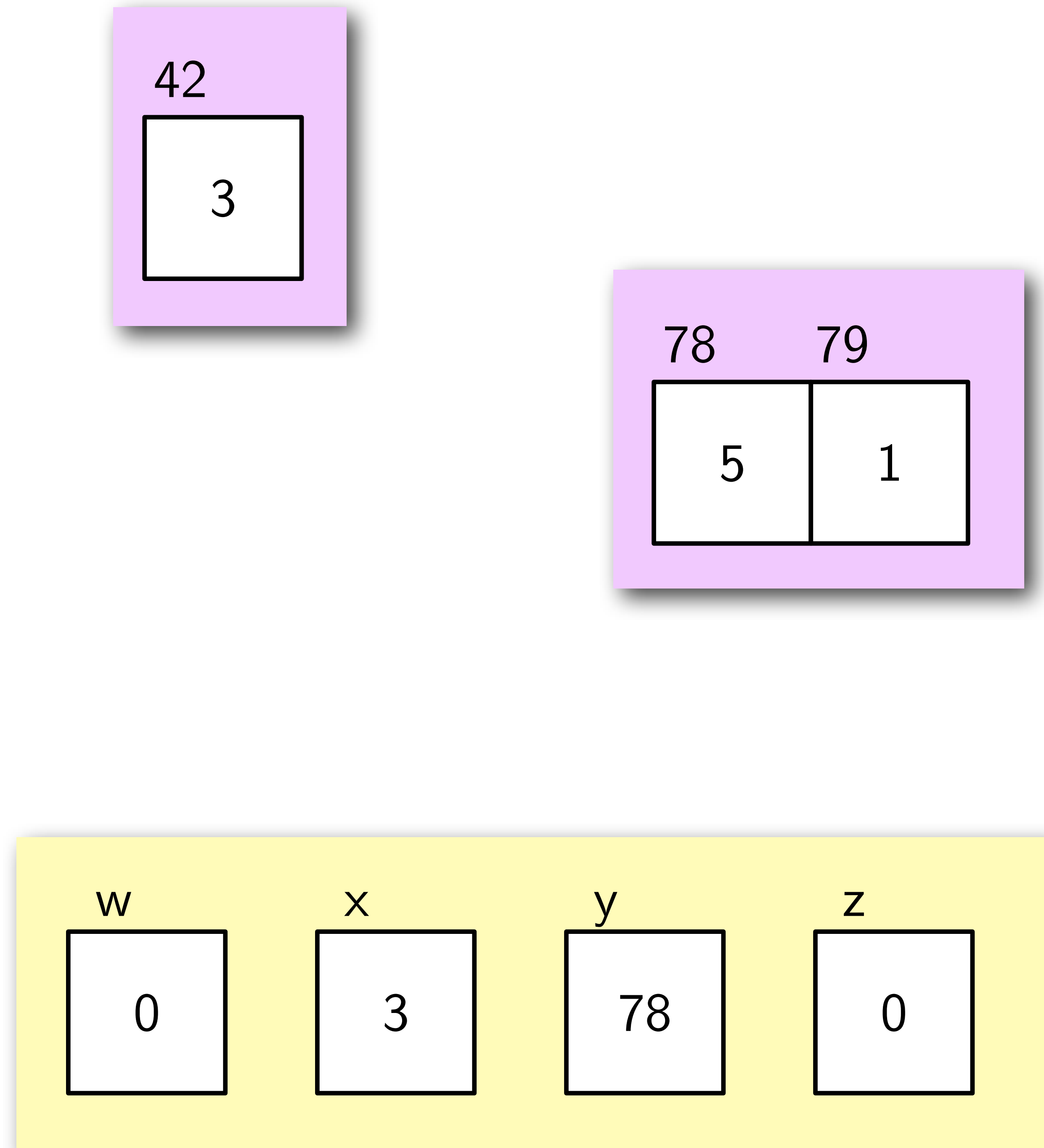




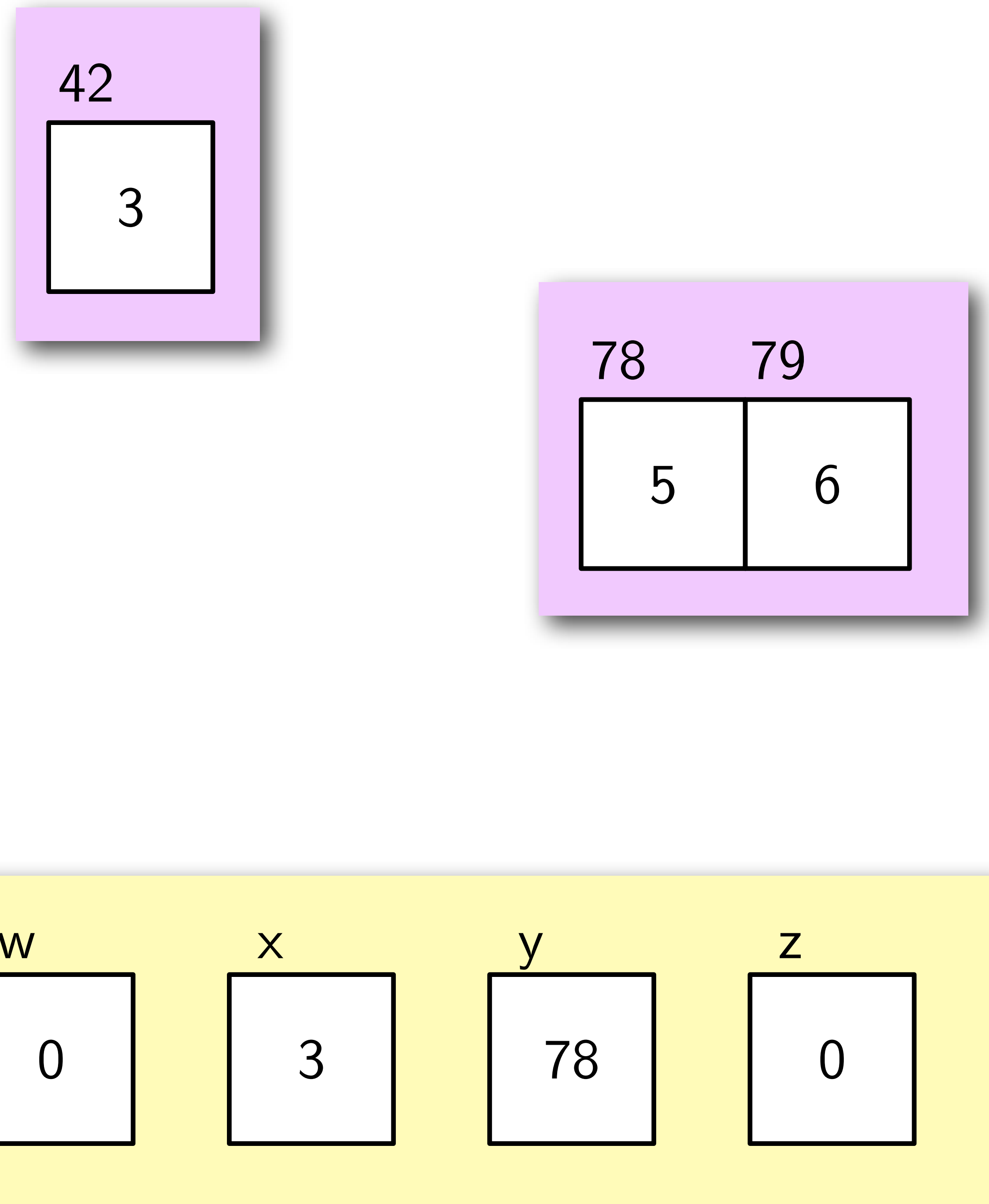
```
x := cons(3);
y := cons(5,1);
x := [x];
[y+1] := 6;
dispose(y);
```



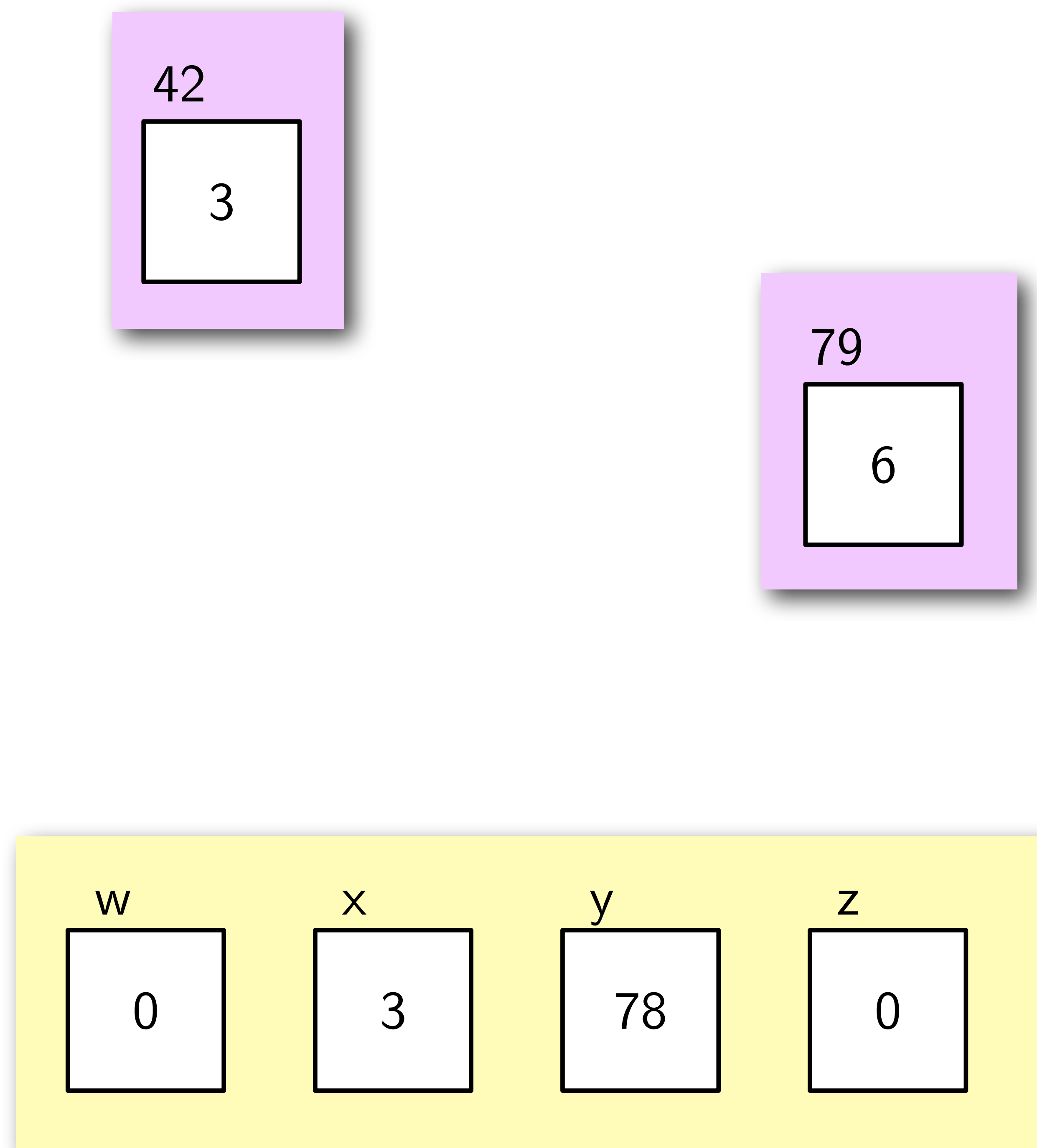
```
x := cons(3);
y := cons(5,1);
x := [x];
[y+1] := 6;
dispose(y);
```



```
x := cons(3);
y := cons(5,1);
x := [x];
[y+1] := 6;
⇒ dispose(y);
```



```
x := cons(3);
y := cons(5,1);
x := [x];
[y+1] := 6;
dispose(y);
```



$\{\mathsf{emp}\}$

$x := \mathsf{cons}(3);$

$\{x \mapsto 3\}$

$y := \mathsf{cons}(5,1);$

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$x := [x];$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$[y+1] := 6;$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

$\mathsf{dispose}(y);$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$\{emp\}$

$x := cons(3);$

$\{x \mapsto 3\}$

$y := cons(5,1);$

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$x := [x];$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$[y+1] := 6;$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

$dispose(y);$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{emp\}\ x := cons(e_1,\ldots,e_n)\ \{x \mapsto e_1,\ldots,e_n\}}$$

$\{\mathsf{emp}\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{\mathsf{emp}\}\ x := \mathsf{cons}(e_1,\ldots,e_n)\ \{x \mapsto e_1,\ldots,e_n\}}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}(\dagger)$$

†provided R doesn't mention any variable modified by C

$\{\mathsf{emp}\}$

$x := \mathsf{cons}(3);$

$\{x \mapsto 3 * \mathsf{emp}\}$

$y := \mathsf{cons}(5,1);$

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$x := [x];$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1$ …

$[y+1] := 6;$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1$ …

$\mathsf{dispose}(y);$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{\mathsf{emp}\}\ x := \mathsf{cons}(e_1,\ldots,e_n)\ \{x \mapsto e_1,\ldots,e_n\}}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ (\dagger)$$

†provided R doesn't mention any variable modified by C

$\{\mathsf{emp}\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$\{emp\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{e \mapsto Y\}\ x := [e]\ \{x \doteq Y * e \mapsto Y\}}$$

$\{\mathsf{emp}\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{e \doteq X * X \mapsto Y\}\ x := [e]\ \{x \doteq Y * X \mapsto Y\}}$$

$\{\mathsf{emp}\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{\exists X.\ x \doteq X * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3$ … $+1 \mapsto 6\}$

dispose(…

$\{\exists X.\ x \doteq$ …

$$\frac{}{\{e \doteq X * X \mapsto Y\}\ \ x := [e]\ \ \{x \doteq Y * X \mapsto Y\}}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{\exists X.P\}\ C\ \{\exists X.Q\}}$$

$\{\mathsf{emp}\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y{+}1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y{+}1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y{+}1 \mapsto 6\}$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y{+}1 \mapsto 6\}$

$\{emp\}$

$x := cons(3);$

$\{x \mapsto 3\}$

$y := cons(5,1);$

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$x := [x];$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

$[y+1] := 6;$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

$dispose(y);$

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{e_1 \mapsto \_\}\ [e_1] := e_2\ \{e_1 \mapsto e_2\}}$$

$\{emp\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

$$\frac{}{\{e \mapsto \_\}\ \text{dispose}(e)\ \{emp\}}$$

$\{emp\}$
x := cons(3);
$\{x \mapsto 3\}$
y := cons(5,1);
$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$
x := [x];
$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$
[y+1] := 6;
$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$
dispose(y);
$\{\exists X.\ x \doteq 3 * X \mapsto 3 * emp * y+1 \mapsto 6\}$

$$\frac{}{\{e \mapsto \_\}\ \text{dispose}(e)\ \{emp\}}$$

$\{\mathsf{emp}\}$

x := cons(3);

$\{x \mapsto 3\}$

y := cons(5,1);

$\{x \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

x := [x];

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 1\}$

[y+1] := 6;

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y \mapsto 5 * y+1 \mapsto 6\}$

dispose(y);

$\{\exists X.\ x \doteq 3 * X \mapsto 3 * y+1 \mapsto 6\}$

Lecture Plan

- A old-style proof of `list_reverse`
- A proof of `list_reverse` in separation logic
- Separation logic's proof rules
- Soundness of the Frame rule

Soundness

if $\vdash \{P\}\ C\ \{Q\}$

then $\models \{P\}\ C\ \{Q\}$

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P * R\}\ C\ \{Q * R\}}(\dagger)$$

†provided R doesn't mention any variable modified by C

assume $\models \{P\}\ C\ \{Q\}$

show $\models \{P{*}R\}\ C\ \{Q{*}R\}$

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

show $\models \{P * R\}\ C\ \{Q * R\}$

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

if $(P*R)\sigma$ then:
(C,σ) doesn't fault, and $(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

$(P*R)\sigma$

(C,σ) doesn't fault, and $(C,\sigma) \Downarrow \sigma' \Rightarrow (Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P*R)\sigma$

(C,σ) doesn't fault

$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models$ {P*R} C {Q*R}

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P * R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

(C,σ) doesn't fault

$(C,\sigma) \Downarrow \sigma' \Rightarrow (Q * R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P{*}R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

(C,σ_1) doesn't fault

(C,σ) doesn't fault

$(C,\sigma){\Downarrow}\sigma' \Rightarrow (Q{*}R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

$(P * R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

(C,σ_1) doesn't fault

Safety Monotonicity

(C,σ) doesn't fault

$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q * R)\sigma'$

show $\models \{P * R\}\ C\ \{Q * R\}$

Soundness of Frame rule

assum

$(P*$

Safety Monotonicity

(C,σ) doesn't fault

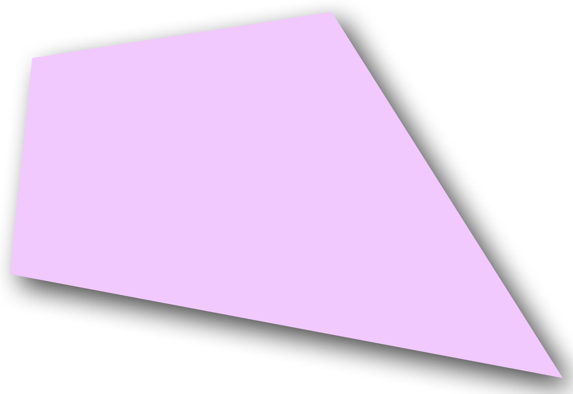
$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assum

$(P*$



C doesn't fault

Safety Monotonicity

(C,σ) doesn't fault

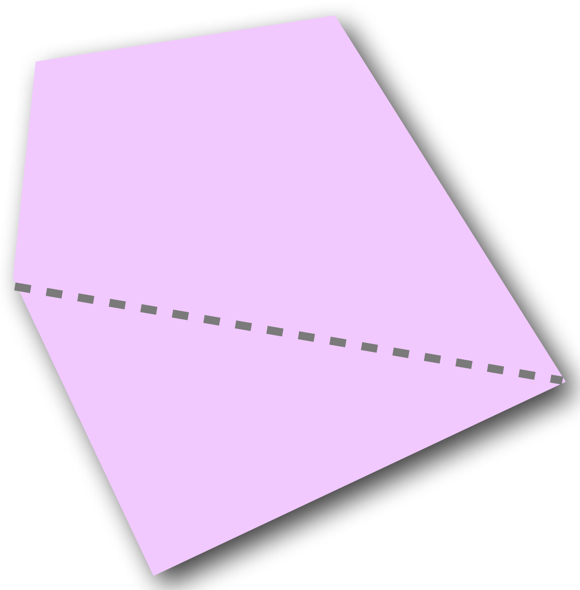
$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assum

(P∗



Safety Monotonicity

(C,σ) doesn't fault

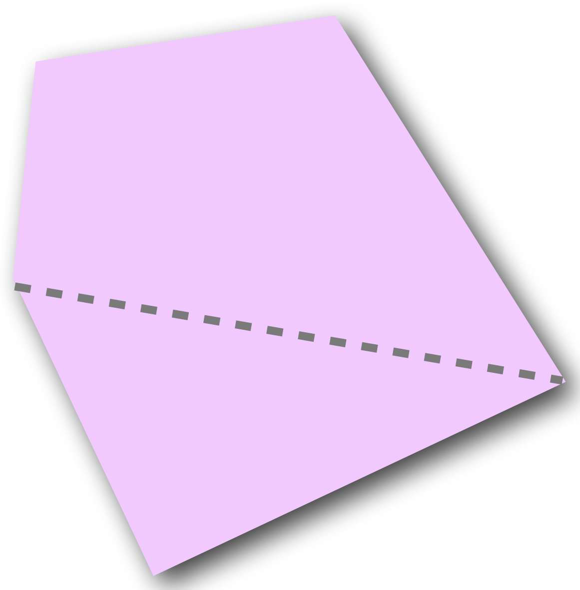
$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assum

(P∗



C still doesn't fault

Safety Monotonicity

(C,σ) doesn't fault

$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

$(P * R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

(C,σ_1) doesn't fault

Safety Monotonicity

(C,σ) doesn't fault

$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q * R)\sigma'$

show $\models \{P * R\}\ C\ \{Q * R\}$

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P*R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

(C,σ_1) doesn't fault

$(C,\sigma)\Downarrow\sigma' \Rightarrow (Q*R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P * R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

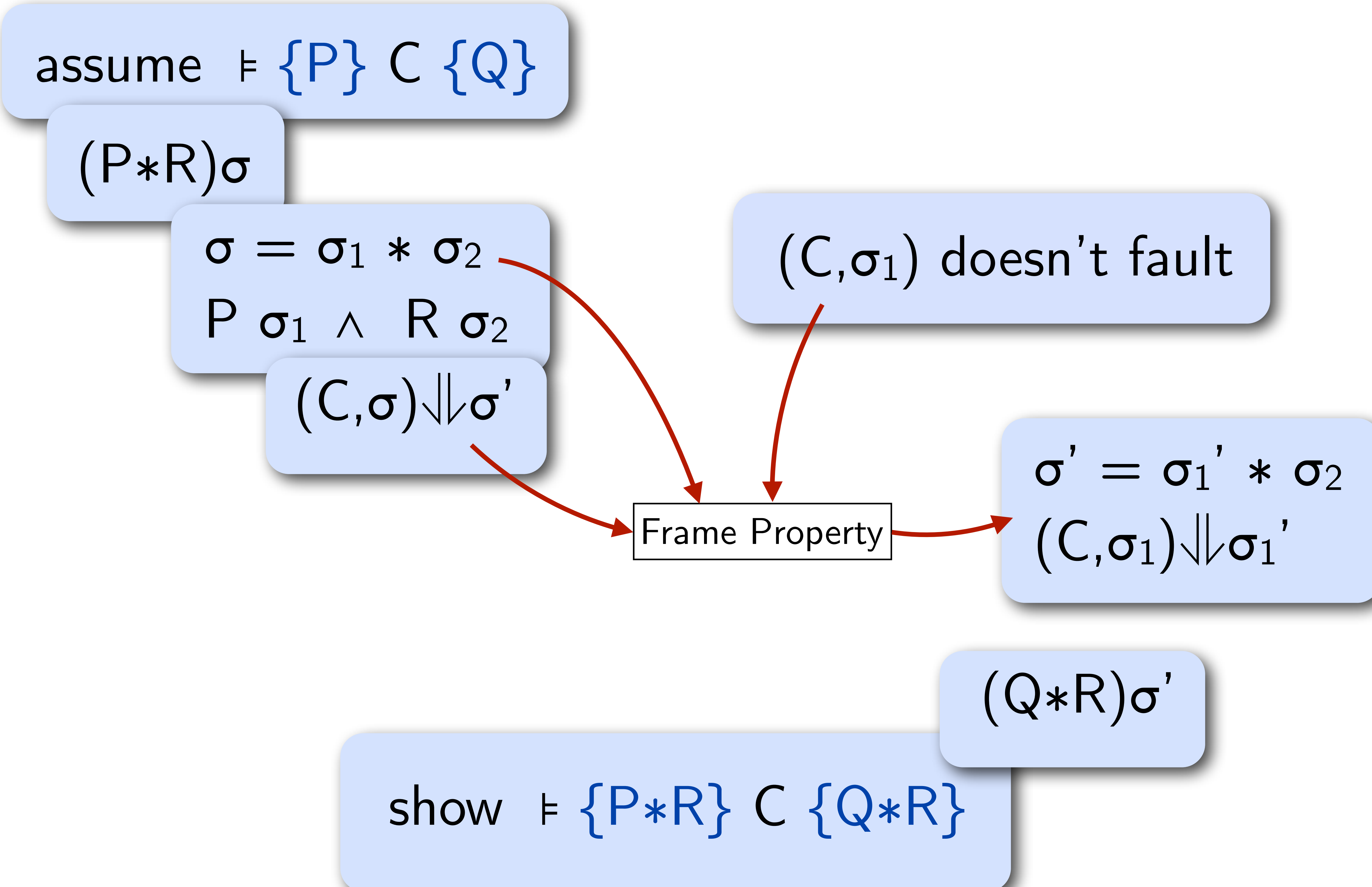
$(C,\sigma) \Downarrow \sigma'$

(C,σ_1) doesn't fault

$(Q * R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

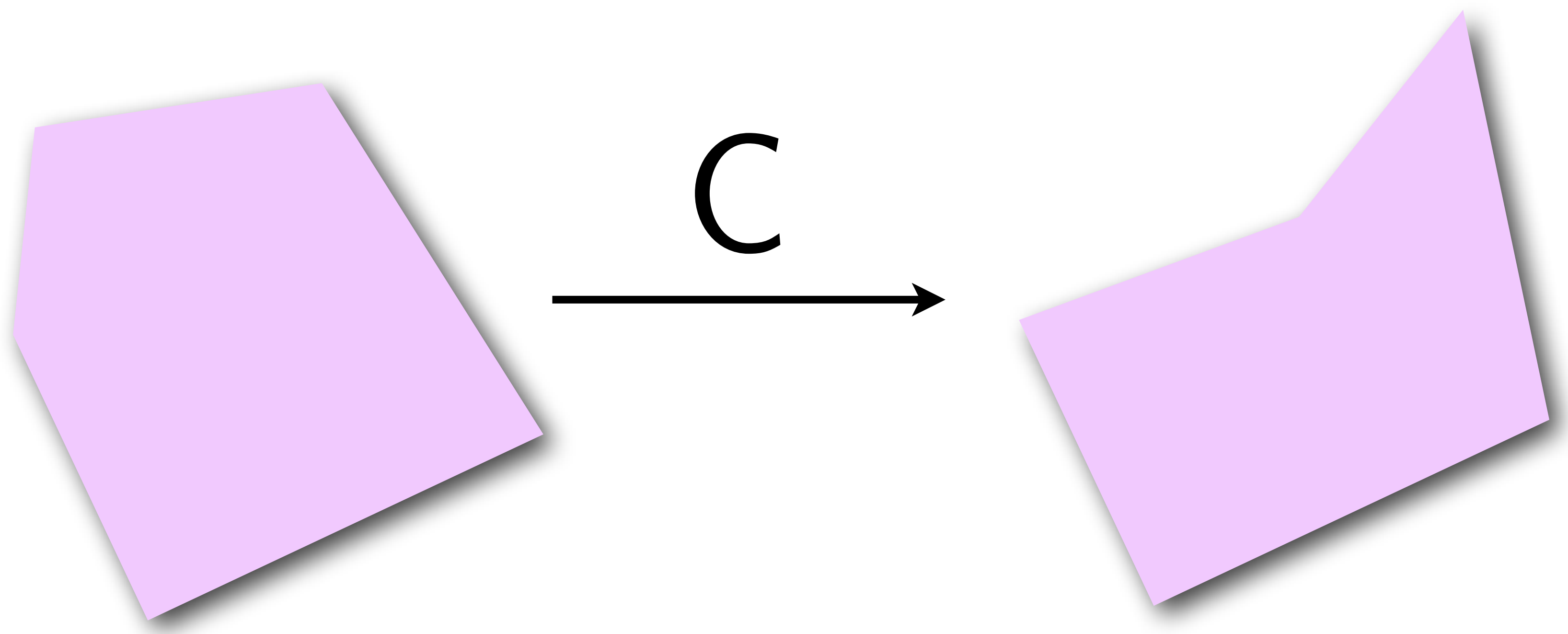
Soundness of Frame rule



Soundness of Frame rule

assum

(P∗



Frame Property

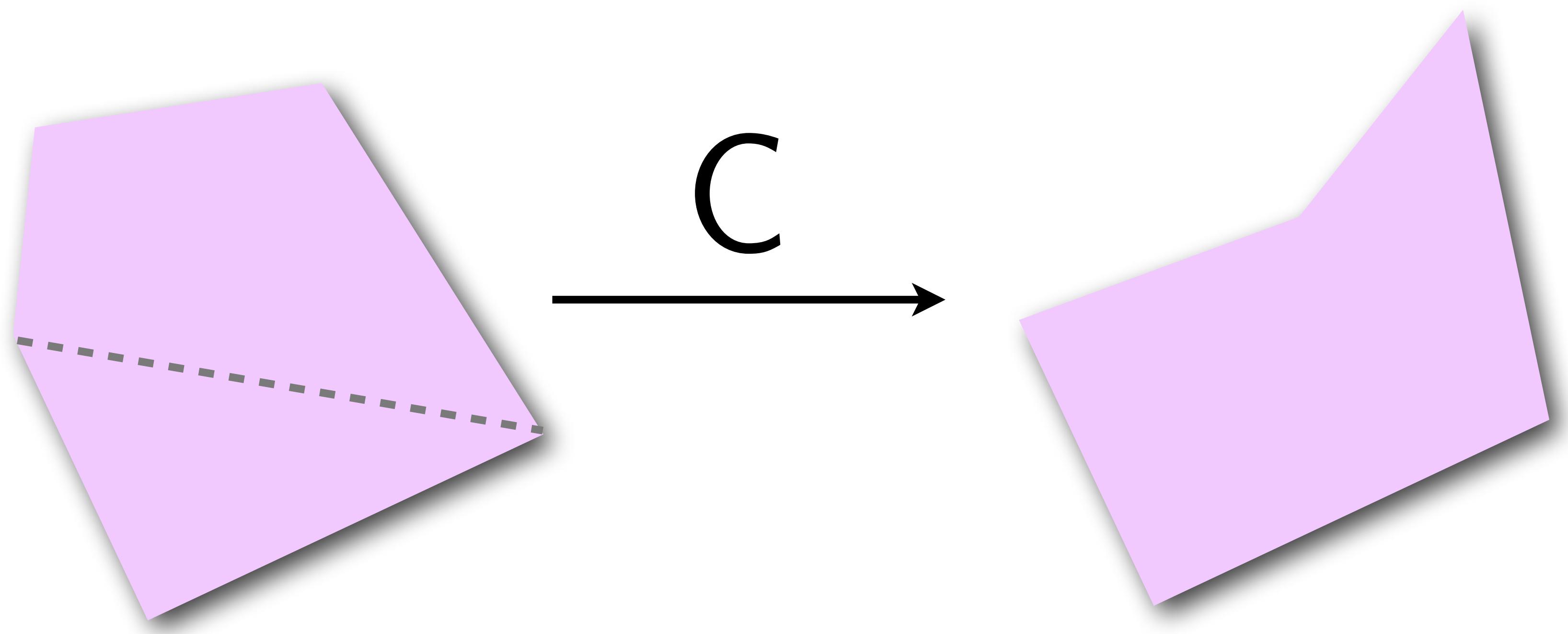
(Q∗R)σ’

show ⊨ {P∗R} C {Q∗R}

Soundness of Frame rule

assum

(P∗



Frame Property

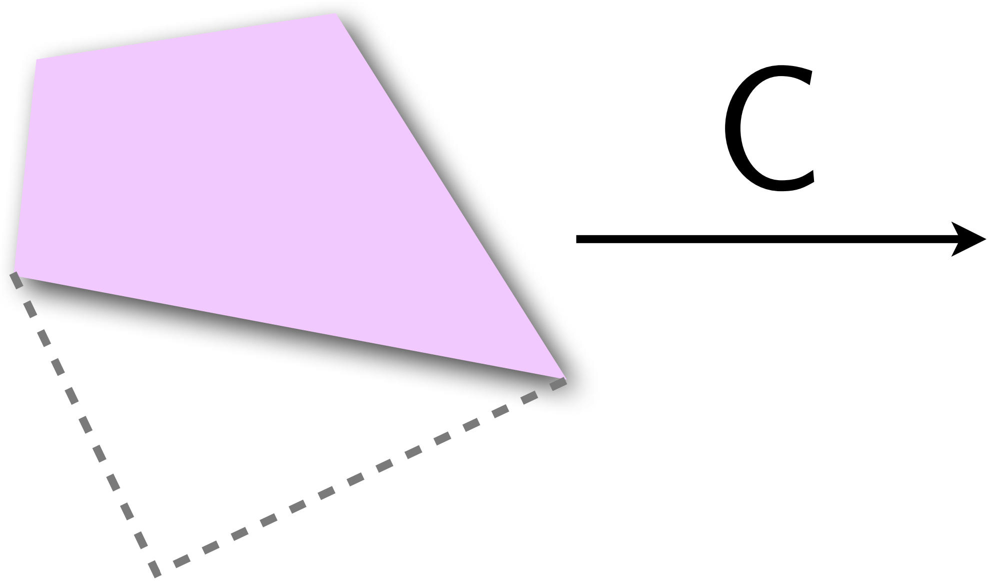
$(Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assum

(P∗



Frame Property

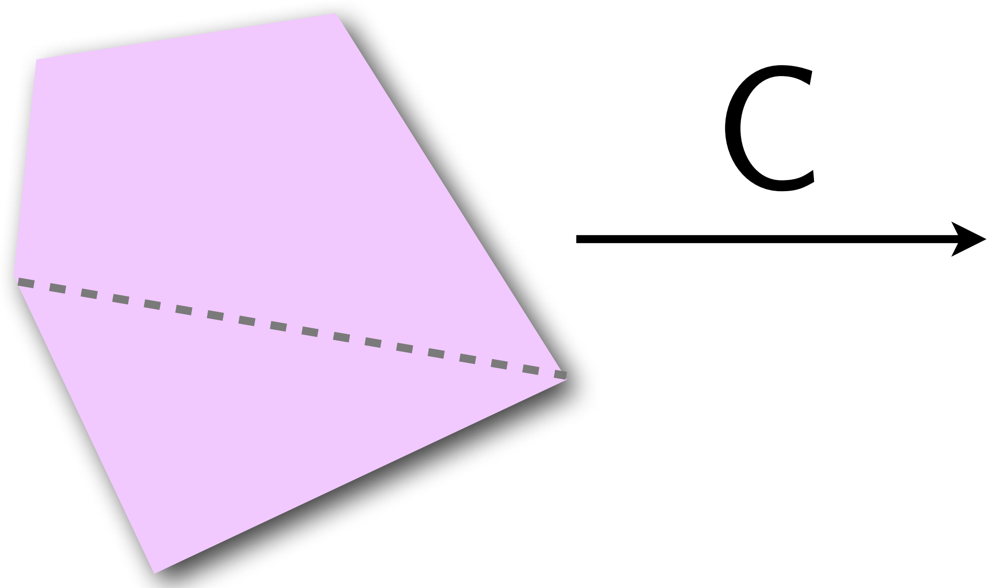
(Q∗R)σ'

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assum

(P∗



Frame Property

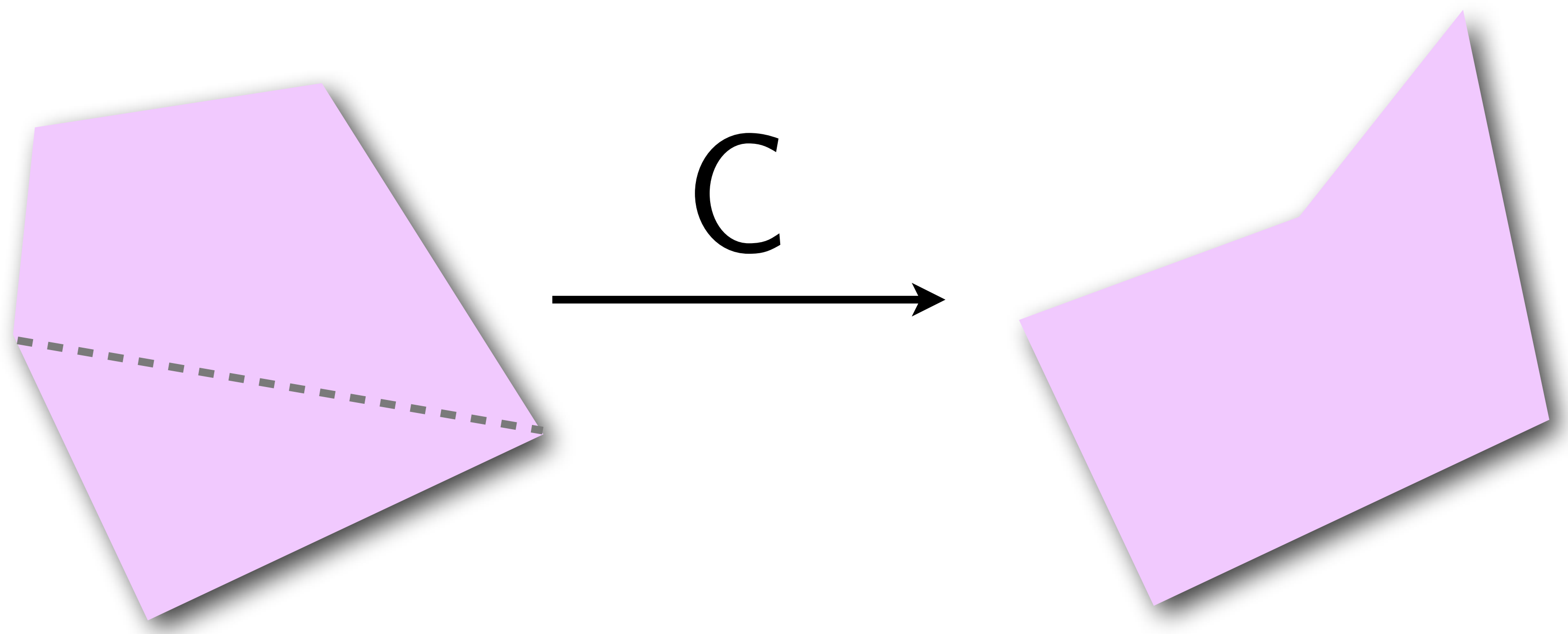
$(Q*R)\sigma'$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame rule

assum

(P∗



Frame Property

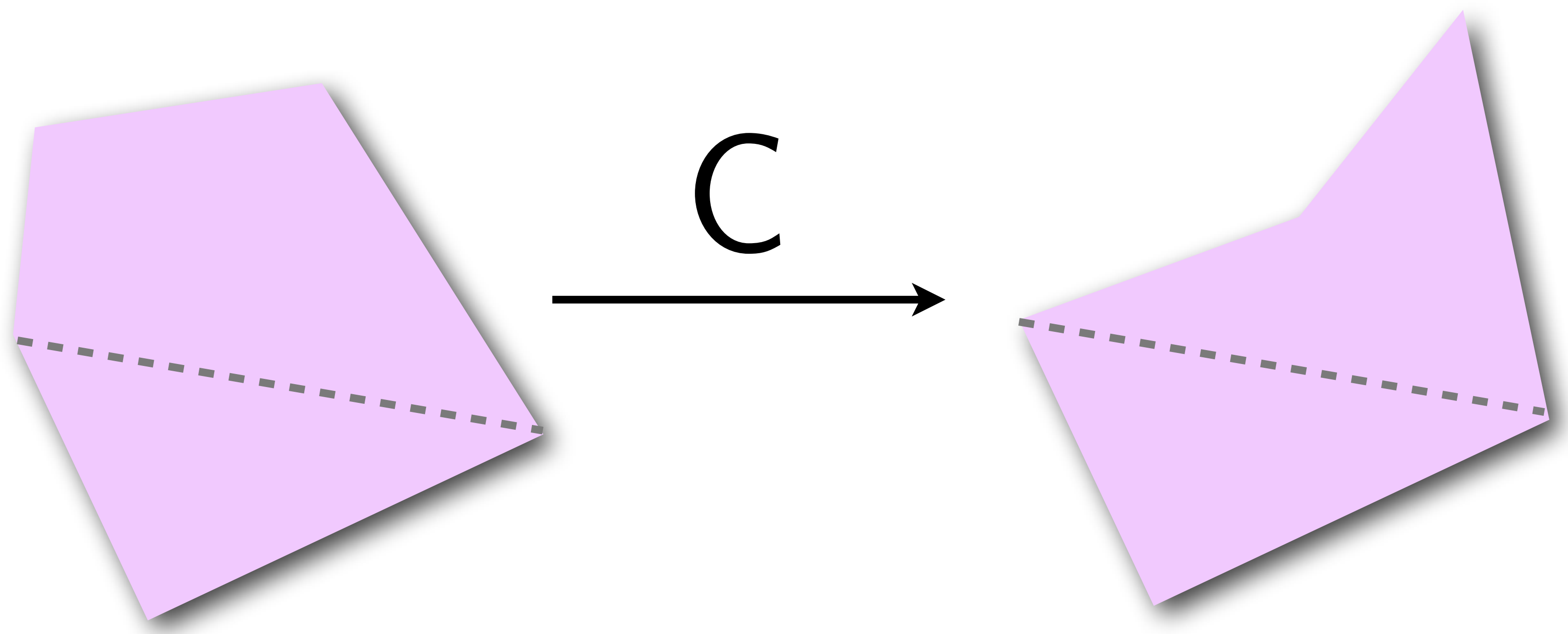
(Q∗R)σ'

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assum

(P∗



Frame Property

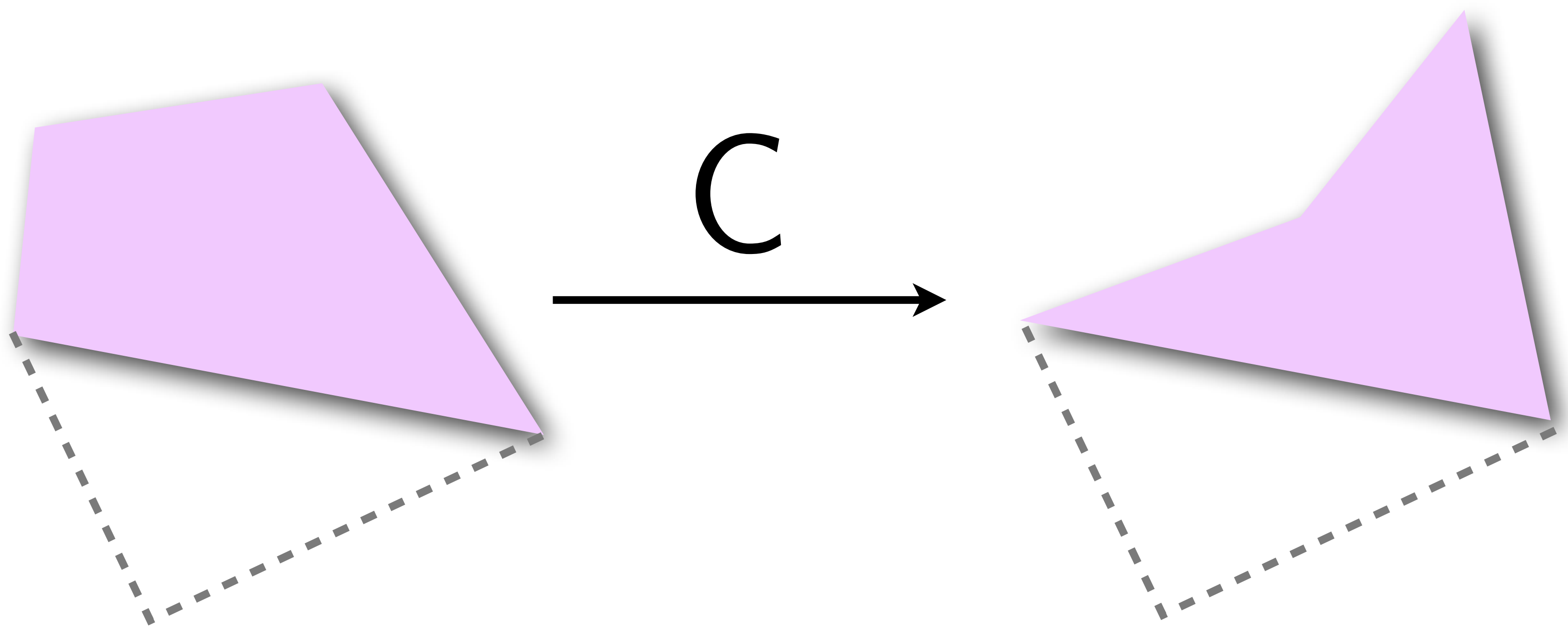
(Q∗R)σ'

show ⊨ {P∗R} C {Q∗R}

Soundness of Frame rule

assum

(P∗



Frame Property

$(Q*R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P*R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

$(C,\sigma) \Downarrow \sigma'$

(C,σ_1) doesn't fault

→ Frame Property →

$\sigma' = \sigma_1' * \sigma_2$

$(C,\sigma_1) \Downarrow \sigma_1'$

$(Q*R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assume $\models$ {P} C {Q}

$(P*R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \wedge R\ \sigma_2$

$\sigma' = \sigma_1' * \sigma_2$

$(C,\sigma_1) \Downarrow \sigma_1'$

$(Q*R)\sigma'$

show $\models$ {P∗R} C {Q∗R}

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

$(P * R)\sigma$

$\sigma = \sigma_1 * \sigma_2$

$P\ \sigma_1 \ \wedge \ R\ \sigma_2$

$\sigma' = \sigma_1' * \sigma_2$

$(C, \sigma_1) \Downarrow \sigma_1'$

$Q\ \sigma_1'$

$(Q * R)\sigma'$

show $\models \{P * R\}\ C\ \{Q * R\}$

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

show $\models \{P * R\}\ C\ \{Q * R\}$

Soundness of Frame rule

assume $\models \{P\}\ C\ \{Q\}$

show $\models \{P * R\}\ C\ \{Q * R\}$

Summary

- A 20th century proof of `list_reverse`
- A proof of `list_reverse` in separation logic
- Separation logic's proof rules
- Soundness of the Frame rule

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

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

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

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

 $s \ = \ s \geq £5$

 $s \ = \ s \geq £20$

( $\wedge$ ) $s \ = \ ($  $s) \text{ and } ($  $s)$

$= \ s \geq £5 \text{ and } s \geq £20$

$= \ s \geq £20$

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

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

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

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

 $s \;=\; s \geq £5$

 $s \;=\; s \geq £20$

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

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

$\;=\; s \geq £25$