Ribbon Proofs for Separation Logic

John Wickerson
Technische Universität Berlin
Mike Dodds
University of York
Matthew Parkinson
Microsoft Research Cambridge

Communications of the ACM October, 1969

An Axiomatic Basis for Computer Programming

C. A. R. Hoare

Line number	Formal proof	Justification
1	$\textbf{true} \supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0 \{r := x\} x = r + y \times 0$	D0
3	$x = r + y \times 0 \ \{q := 0\} \ x = r + y \times q$	D0
4	$\textbf{true} \ \{r := x\} \ x = r + y \times 0$	D1 (1, 2)
5	$\textbf{true} \ \{r := x; \ q := 0\} \ x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leqslant r \supset x = (r-y) + y \times (1+q)$	Lemma 2
7	$x = (r-y) + y \times (1+q) \{r := r-y\} x = r + y \times (1+q)$	D0
8	$x = r + y \times (1+q) \{q := 1+q\} x = r + y \times q$	D0
9	$x = (r-y) + y \times (1+q) \{r := r-y; \ q := 1+q\} \ x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leqslant r \ \{r := r-y; \ q := 1+q\} \ x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q \ \{\textbf{while} \ y \leqslant r \ \textbf{do} \ (r := r-y; \ q := 1+q)\} \ \neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	$\textbf{true} \ \{((r := x; \ q := 0); \ \textbf{while} \ y \leqslant r \ \textbf{do} \ (r := r-y; \ q := 1+q))\} \ \neg y \leqslant r \wedge x = r + y \times q$	D2 (5, 11)

```
begin
  comment This program operates on an array A[1:N], and a
    value of f(1 ≤ f ≤ N). Its effect is to rearrange the elements
    of A in such a way that:
      ∀p,q(1≤p≤f≤q≤N⊃A[p]≤A[f]≤A[q]);
  integer m, n; comment
      m ≤ f & ∀p,q(1≤p<m≤q≤N⊃A[p]≤A[q]),
      f ≤ n & ∀p,q(1≤p≤n<q≤N⊃A[p]≤A[q]);
    m := 1; n := N;
  while m < n do
  begin integer r, i, j, w;
    comment
        m ≤ i & ∀p(1≤p<i⊃A[p]≤r),
        j ≤ n & ∀q(j<q≤N⊃r≤A[q]);
      r := A[f]; i := m; j := n;
    while i ≤ j do
    begin while A[i] < r do i := i + 1;
      while r < A[j] do j := j − 1
      comment A[j] ≤ r ≤ A[i];
      if i ≤ j then
      begin w := A[i]; A[i] := A[j]; A[j] := w;
        comment A[i] ≤ r ≤ A[j];
        i := i + 1; j := j − 1;
      end
    end increase i and decrease j;
    if f ≤ j then n := j
  else if i ≤ f then m := i
    else go to L
  end reduce middle part;
L:
end Find
```

Communications of the ACM January, 1971

Proof of a Program: FIND

C. A. R. Hoare
Queen's University, Belfast, Ireland*

Acta Informatica 6, 319—340 (1976)

An Axiomatic Proof Technique for Parallel Programs I*

Susan Owicki and David Gries

$\{x=0\}$

S: **cobegin** $\{x=0\}$

$\{x=0 \vee x=2\}$

$S1$: **await true then** $x := x+1$

$\{Q1\colon x=1 \vee x=3\}$

//

$\{x=0\}$

$\{x=0 \vee x=1\}$

$S2$: **await true then** $x := x+2$

$\{Q2\colon x=2 \vee x=3\}$

coend

$\{(x=1 \vee x=3) \wedge (x=2 \vee x=3)\}$

$\{x=3\}$

Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)

Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*

$$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = \alpha^\dagger {\cdot} \beta \wedge \mathsf{i} \neq \mathbf{nil}\}$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathbf{list}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger {\cdot} \beta\}$$
$$\{\exists \mathsf{a}, \alpha, \beta, \mathsf{k}.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger {\cdot} \beta\}$$
$$\mathsf{k} := [\mathsf{i} + 1]\ ;$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger {\cdot} \beta\}$$
$$[\mathsf{i} + 1] := \mathsf{j}\ ;$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger {\cdot} \beta\}$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \mathsf{a}{\cdot}\beta\ (\mathsf{i}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = \alpha^\dagger {\cdot} \mathsf{a} {\cdot} \beta\}$$
$$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{i}, \mathbf{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger {\cdot} \beta\}$$
$$\mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}$$
$$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger {\cdot} \beta\}.$$

Ribbon proofs are...

- an alternative to **proof outlines**
- **readable**, **flexible**, and **attractive**
- applicable to **separation logic** (and descendants)
- less **repetitive** than proof outlines, so more **scalable**

Tiny example

```
[x]:=1;

[y]:=1;

[z]:=1;
```

$$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

```
[x]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

```
[y]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$$

```
[z]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$$

$$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

`[x]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

`[y]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$$

`[z]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$$

$$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

```
[x]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

```
[y]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$$

```
[z]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$$

$$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

`[x]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

$$\{\mathtt{y} \mapsto 0\}$$

`[y]:=1;`

$$\{\mathtt{y} \mapsto 1\}$$

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$$

`[z]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$$

frame
$\mathtt{x} \mapsto 1 * \mathtt{z} \mapsto 0$

small axiom
for heap update

$$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

```
[x]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

```
[y]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$$

```
[z]:=1;
```

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$$

```
mchunkptr b, p;
idx += ~smallbits & 1; /* Uses next bin if idx empty */
```

$$\left\{\begin{array}{l}\exists\{U_i \mid i \in [0,63)\}, n.\ \mathit{arena}(A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}}) \ * \ \texttt{least\_addr} = 5\mathsf{w} \\ * \ n\mathsf{w} = \lceil \texttt{bytes} \rceil_{\mathsf{w}} \ * \ 8\texttt{idx} \geq (n+1)\mathsf{w} \ * \ 2 \leq \texttt{idx} < 32 \ * \ \texttt{smallmap}_{[\texttt{idx}]} = 1 \\ * \ \bigstar_{i=0}^{32}. \mathit{smallbin}_i(U_i) \ * \ \bigstar_{i=0}^{32}. \mathit{treebin}_i(U_{i+32})\end{array}\right\}$$

```
b = smallbin_at(gm, idx);
```

$$\left\{\begin{array}{l}\exists\{U_i \mid i \in [0,63)\}, n.\ \mathit{arena}(A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}}) \ * \ \texttt{least\_addr} = 5\mathsf{w} \\ * \ n\mathsf{w} = \lceil \texttt{bytes} \rceil_{\mathsf{w}} \ * \ 8\texttt{idx} \geq (n+1)\mathsf{w} \ * \ 2 \leq \texttt{idx} < 32 \ * \ \texttt{smallmap}_{[\texttt{idx}]} = 1 \\ * \ \texttt{b} = \texttt{smallbins} + 8\texttt{idx} \ * \ \mathit{bin}(|\texttt{idx}|, \texttt{b}, U_{\texttt{idx}}) \ * \ U_{\texttt{idx}} \neq \{\} \\ * \ \bigstar_{i \in [0..32) - \texttt{idx}}. \mathit{smallbin}_i(U_i) \ * \ \bigstar_{i=0}^{32}. \mathit{treebin}_i(U_{i+32})\end{array}\right\}$$

```
// rename U_idx to U_idx++[p+2w->8idx-1w]
```

$$\left\{\begin{array}{l}\exists\{U_i \mid i \in [0,63)\}, p, n.\ \mathit{arena}(A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}} \uplus \{p + 2\mathsf{w} \mapsto_{\mathsf{u}} 8\texttt{idx} - 1\mathsf{w}\}) \\ * \ \texttt{least\_addr} = 5\mathsf{w} \ * \ n\mathsf{w} = \lceil \texttt{bytes} \rceil_{\mathsf{w}} \ * \ 8\texttt{idx} \geq (n+1)\mathsf{w} \ * \ 2 \leq \texttt{idx} < 32 \\ * \ \texttt{smallmap}_{[\texttt{idx}]} = 1 \ * \ \texttt{b} = \texttt{smallbins} + 8\texttt{idx} \\ * \ \texttt{b} \overset{\mathsf{fd}}{\longmapsto} p \ * \ p \overset{\mathsf{bk}}{\longmapsto} \texttt{b} \ * \ (\mathit{bnode}\,|\texttt{idx}|)^*(p, \texttt{b}, U_{\texttt{idx}} \uplus \{p + 2\mathsf{w} \mapsto 8\texttt{idx} - 1\mathsf{w}\}) \\ * \ \bigstar_{i \in [0..32) - \texttt{idx}}. \mathit{smallbin}_i(U_i) \ * \ \bigstar_{i=0}^{32}. \mathit{treebin}_i(U_{i+32})\end{array}\right\}$$

```
p = b->fd;
```

$$\left\{\begin{array}{l}\exists\{U_i \mid i \in [0,63)\}, n, F.\ \mathit{arena}(A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}} \uplus \{\texttt{p} + 2\mathsf{w} \mapsto_{\mathsf{u}} 8\texttt{idx} - 1\mathsf{w}\}) \\ * \ \texttt{least\_addr} = 5\mathsf{w} \ * \ n\mathsf{w} = \lceil \texttt{bytes} \rceil_{\mathsf{w}} \ * \ 8\texttt{idx} \geq (n+1)\mathsf{w} \ * \ 2 \leq \texttt{idx} < 32 \\ * \ \texttt{smallmap}_{[\texttt{idx}]} = 1 \ * \ \texttt{b} = \texttt{smallbins} + 8\texttt{idx} \\ * \ \texttt{b} \overset{\mathsf{fd}}{\longmapsto} \texttt{p} \ * \ \texttt{p} \overset{\mathsf{bk}}{\longmapsto} \texttt{b} \ * \ \frac{1}{2}(\texttt{p} \overset{\mathsf{size}}{\longmapsto} 8\texttt{idx}) \ * \ \texttt{p} \overset{\mathsf{fd}}{\longmapsto} F \ * \ F \overset{\mathsf{bk}}{\longmapsto} \texttt{p} \ * \ (\mathit{bnode}\,|\texttt{idx}|)^*(F, \texttt{b}, U_{\texttt{idx}}) \\ * \ \bigstar_{i \in [0..32) - \texttt{idx}}. \mathit{smallbin}_i(U_i) \ * \ \bigstar_{i=0}^{32}. \mathit{treebin}_i(U_{i+32})\end{array}\right\}$$

```
//assert(chunksize(p) == small_index2size(idx));
unlink_first_small_chunk(gm, b, p, idx);
```

$$\left\{\begin{array}{l}\exists\{U_i \mid i \in [0,63)\}, n.\ \mathit{arena}(A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}} \uplus \{\texttt{p} + 2\mathsf{w} \mapsto_{\mathsf{u}} 8\texttt{idx} - 1\mathsf{w}\}) \\ * \ \texttt{least\_addr} = 5\mathsf{w} \ * \ n\mathsf{w} = \lceil \texttt{bytes} \rceil_{\mathsf{w}} \ * \ 8\texttt{idx} \geq (n+1)\mathsf{w} \ * \ 2 \leq \texttt{idx} < 32 \\ * \ \frac{1}{2}(\texttt{p} \overset{\mathsf{size}}{\longmapsto} 8\texttt{idx}) \ * \ \texttt{p} \overset{\mathsf{fd}}{\longmapsto} \_ \ * \ \texttt{p} \overset{\mathsf{bk}}{\longmapsto} \_ \ * \ \bigstar_{i=0}^{32}. \mathit{smallbin}_i(U_i) \ * \ \bigstar_{i=0}^{32}. \mathit{treebin}_i(U_{i+32})\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists\{U_i \mid i \in [0,63)\}, B_1, B_2, n.\ \mathit{coallesced}(A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}} \uplus \{\texttt{p} + 2\mathsf{w} \mapsto_{\mathsf{u}} 8\texttt{idx} - 1\mathsf{w}\}) \\ * \ \texttt{start} \overset{\mathsf{prevfoot}}{\longmapsto} \_ \ * \ \texttt{start} \overset{\mathsf{pinuse}}{\longmapsto} 1 \ * \ \mathit{ublock}(\texttt{top}, \texttt{top} + \texttt{topsize}, \_) \\ * \ \mathit{block}^*(\texttt{start}, \texttt{p}, B_1) \ * \ \mathit{ublock}(\texttt{p}, \texttt{p} + 8\texttt{idx}, \{\texttt{p} + 2\mathsf{w} \mapsto_{\mathsf{u}} 8\texttt{idx} - 1\mathsf{w}\}) \\ * \ \mathit{block}^*(\texttt{p} + 8\texttt{idx}, \texttt{top}, B_2) \ * \ B_1 \uplus B_2 = A_{\mathsf{a}} \uplus (\biguplus_{i=0}^{64}. U_i)_{\mathsf{u}} \\ * \ \texttt{least\_addr} = 5\mathsf{w} \ * \ n\mathsf{w} = \lceil \texttt{bytes} \rceil_{\mathsf{w}} \ * \ 8\texttt{idx} \geq (n+1)\mathsf{w} \ * \ 2 \leq \texttt{idx} < 32 \\ * \ \frac{1}{2}(\texttt{p} \overset{\mathsf{size}}{\longmapsto} 8\texttt{idx}) \ * \ \texttt{p} \overset{\mathsf{fd}}{\longmapsto} \_ \ * \ \texttt{p} \overset{\mathsf{bk}}{\longmapsto} \_ \ * \ \bigstar_{i=0}^{32}. \mathit{smallbin}_i(U_i) \ * \ \bigstar_{i=0}^{32}. \mathit{treebin}_i(U_{i+32})\end{array}\right\}$$

$$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

`[x]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$$

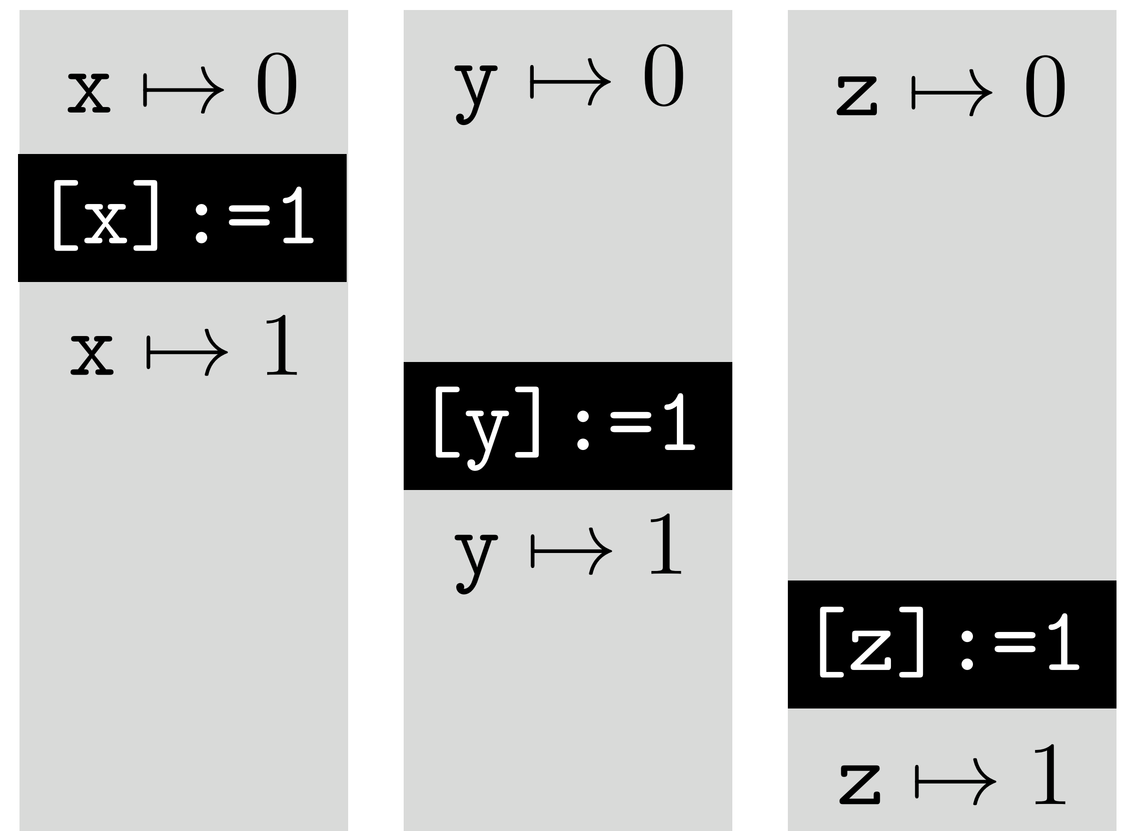
`[y]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$$

`[z]:=1;`

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$$

A proof outline



A ribbon proof

$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$

`[x]:=1;`

$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$

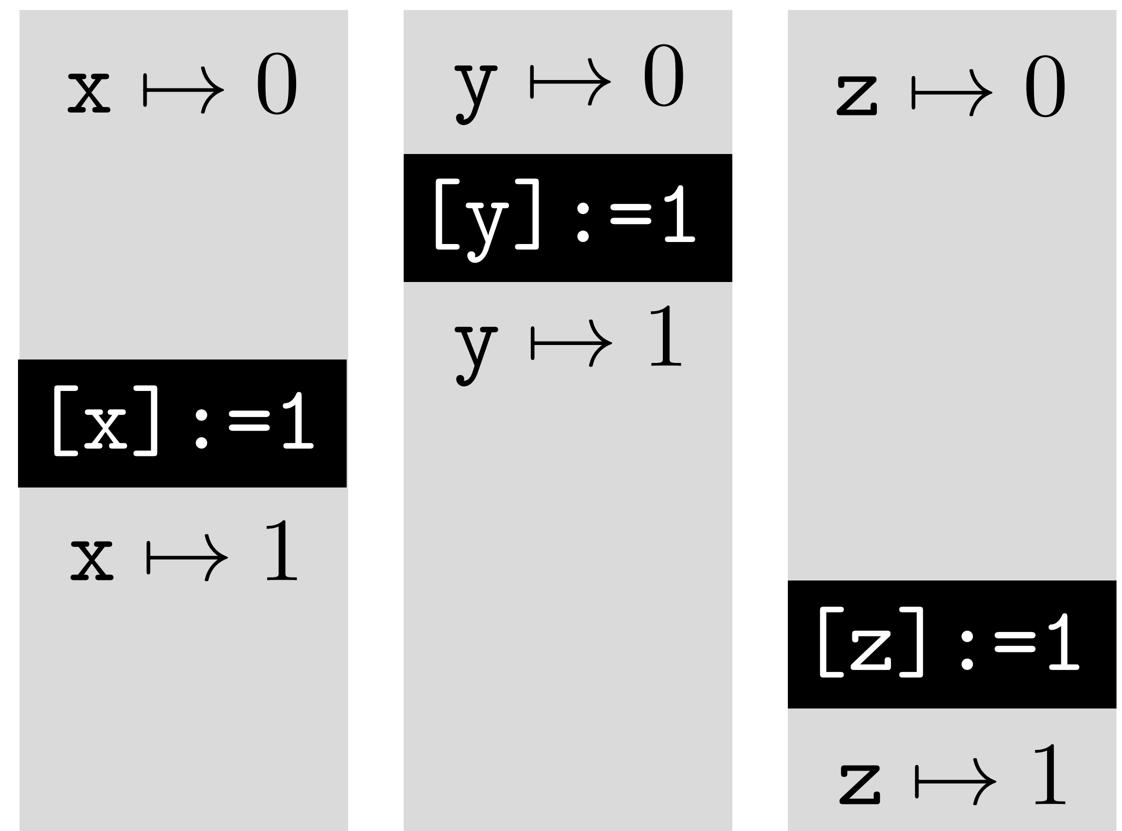
`[y]:=1;`

$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$

`[z]:=1;`

$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$

A proof outline



A ribbon proof

$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$

[x]:=1;

$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$

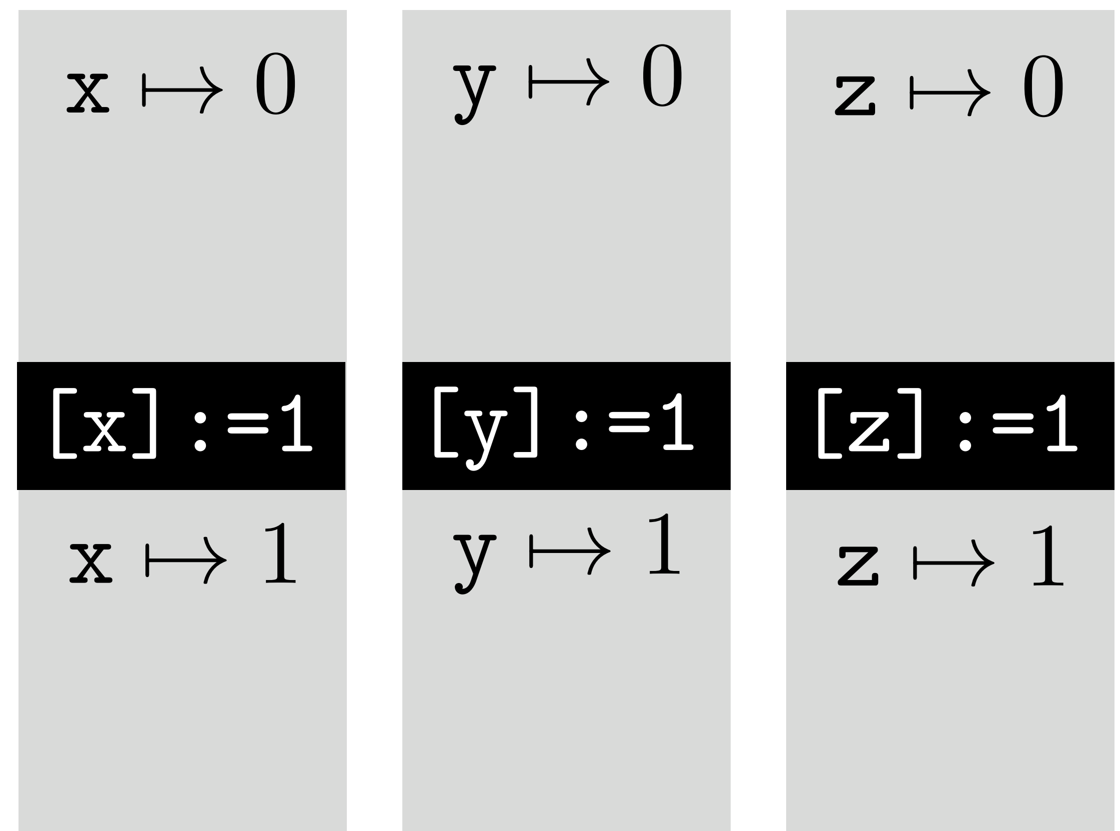
[y]:=1;

$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$

[z]:=1;

$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$

A proof outline

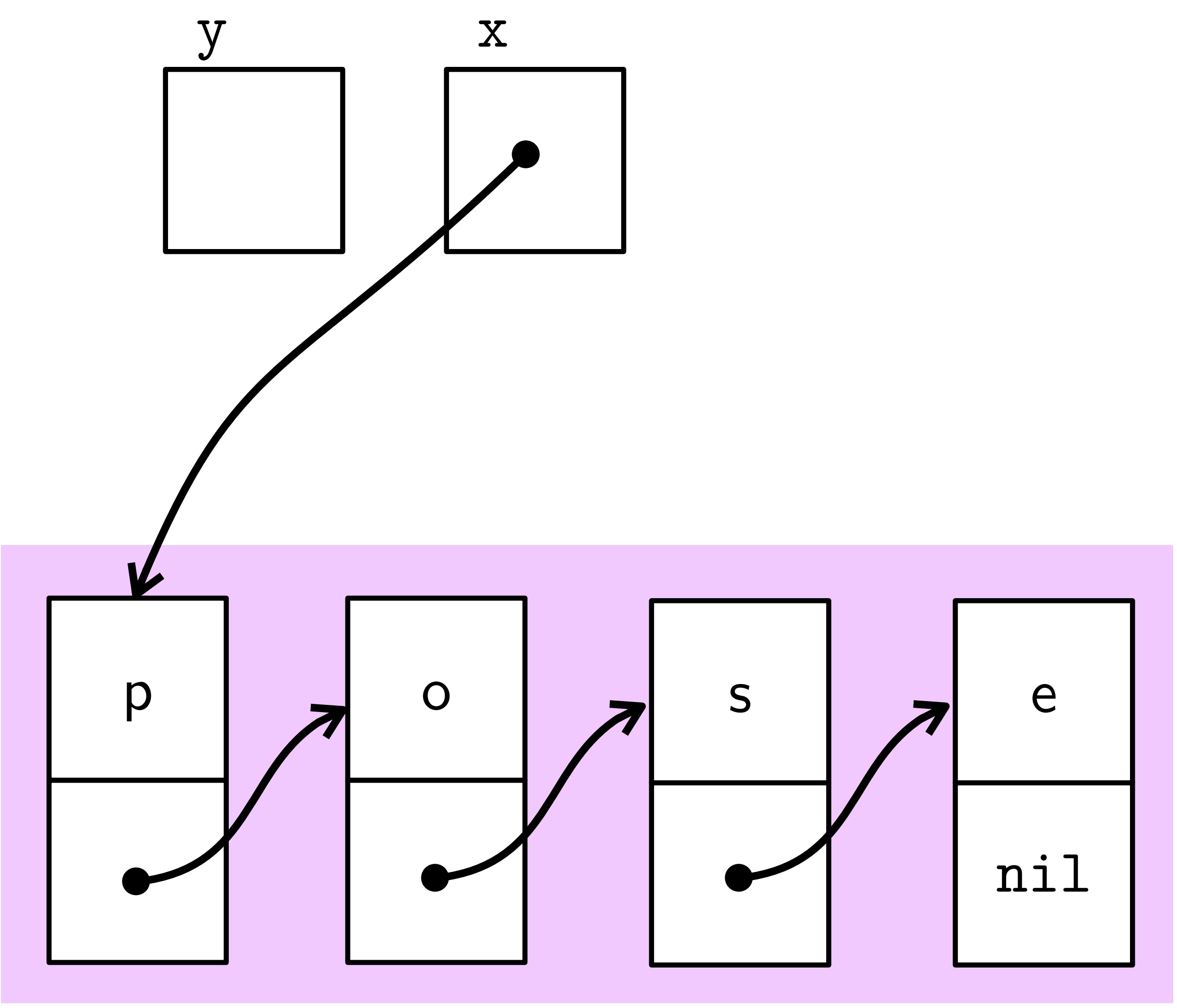


A ribbon proof

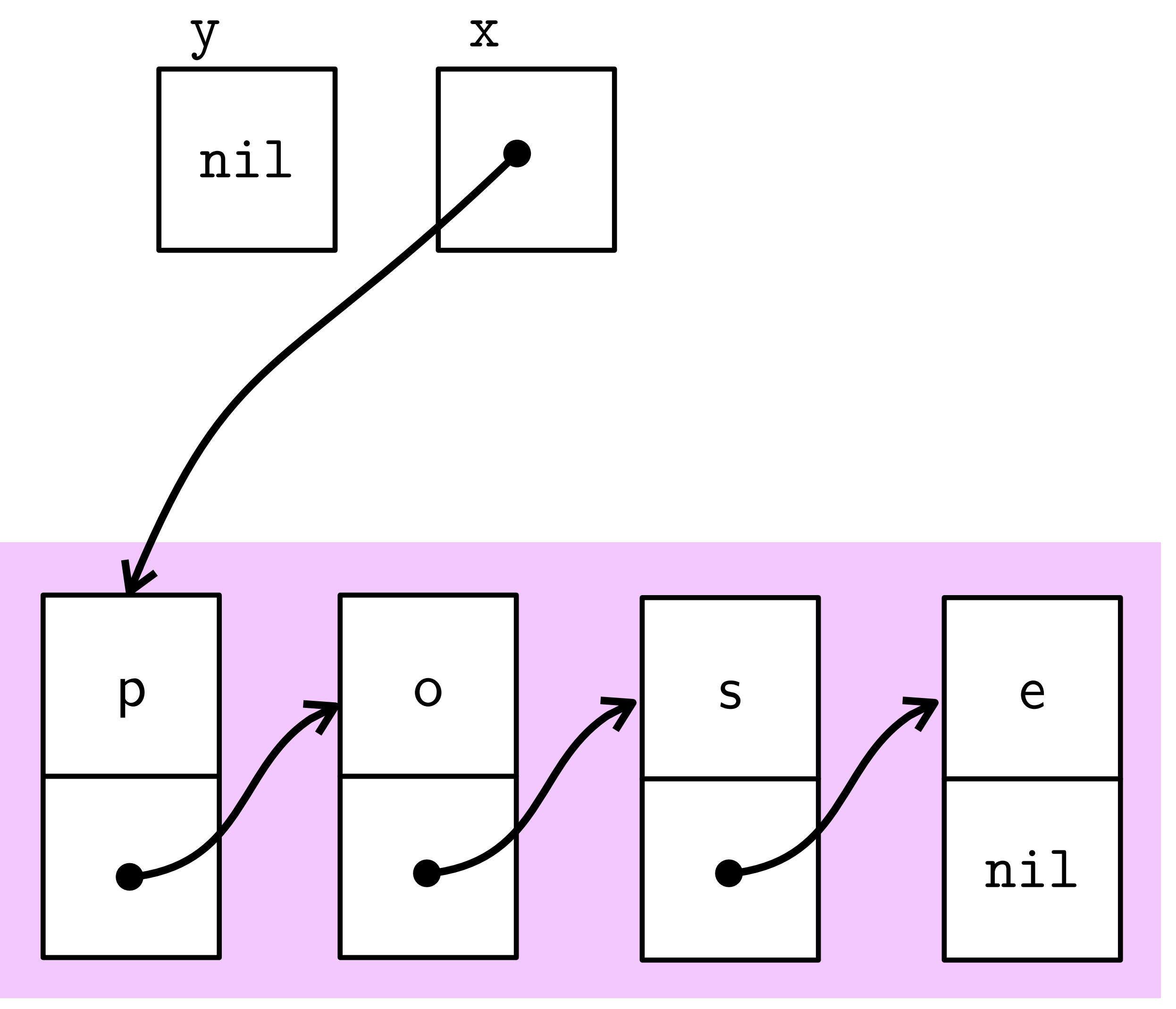
Example: in-place list reversal

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

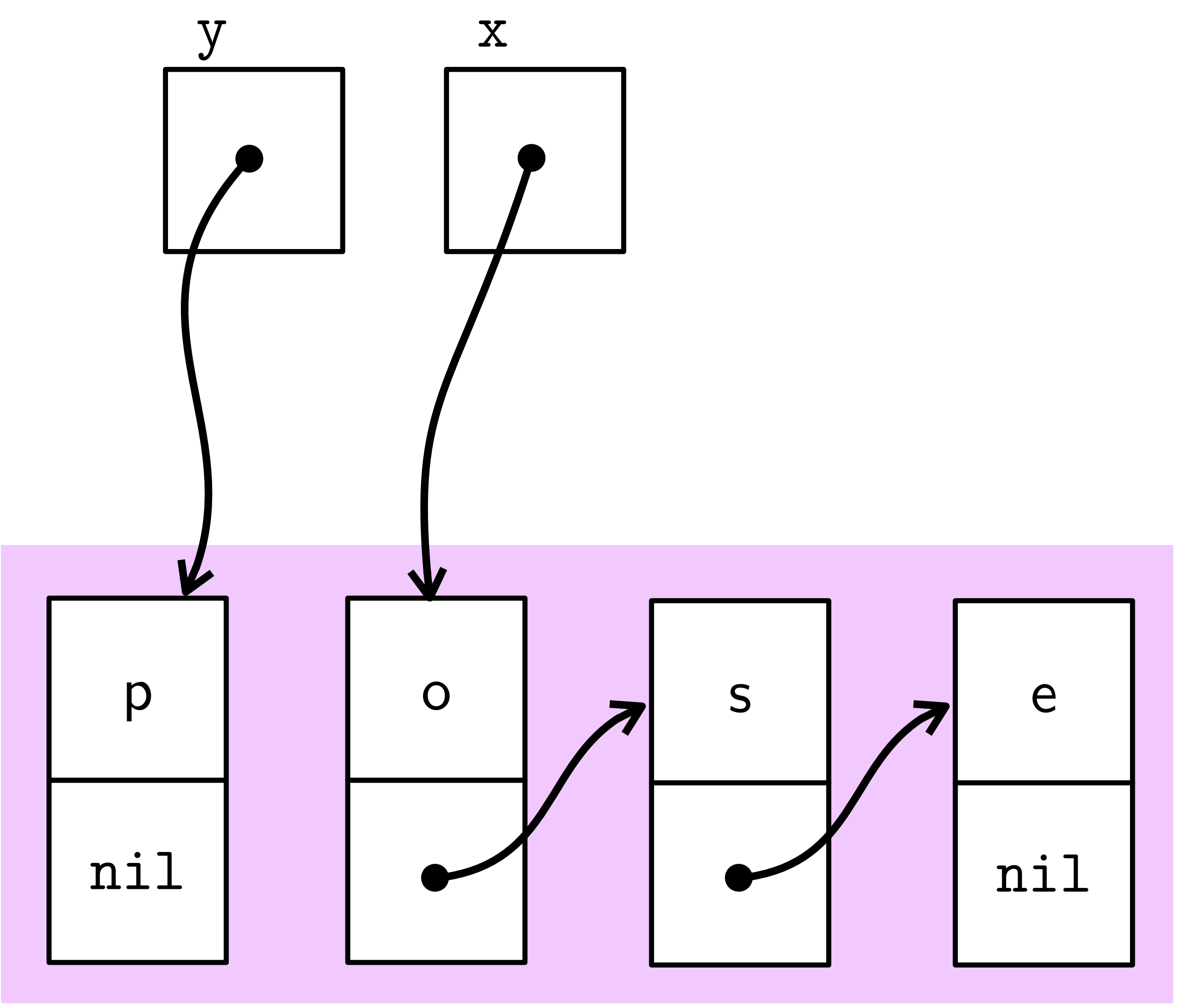
```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```



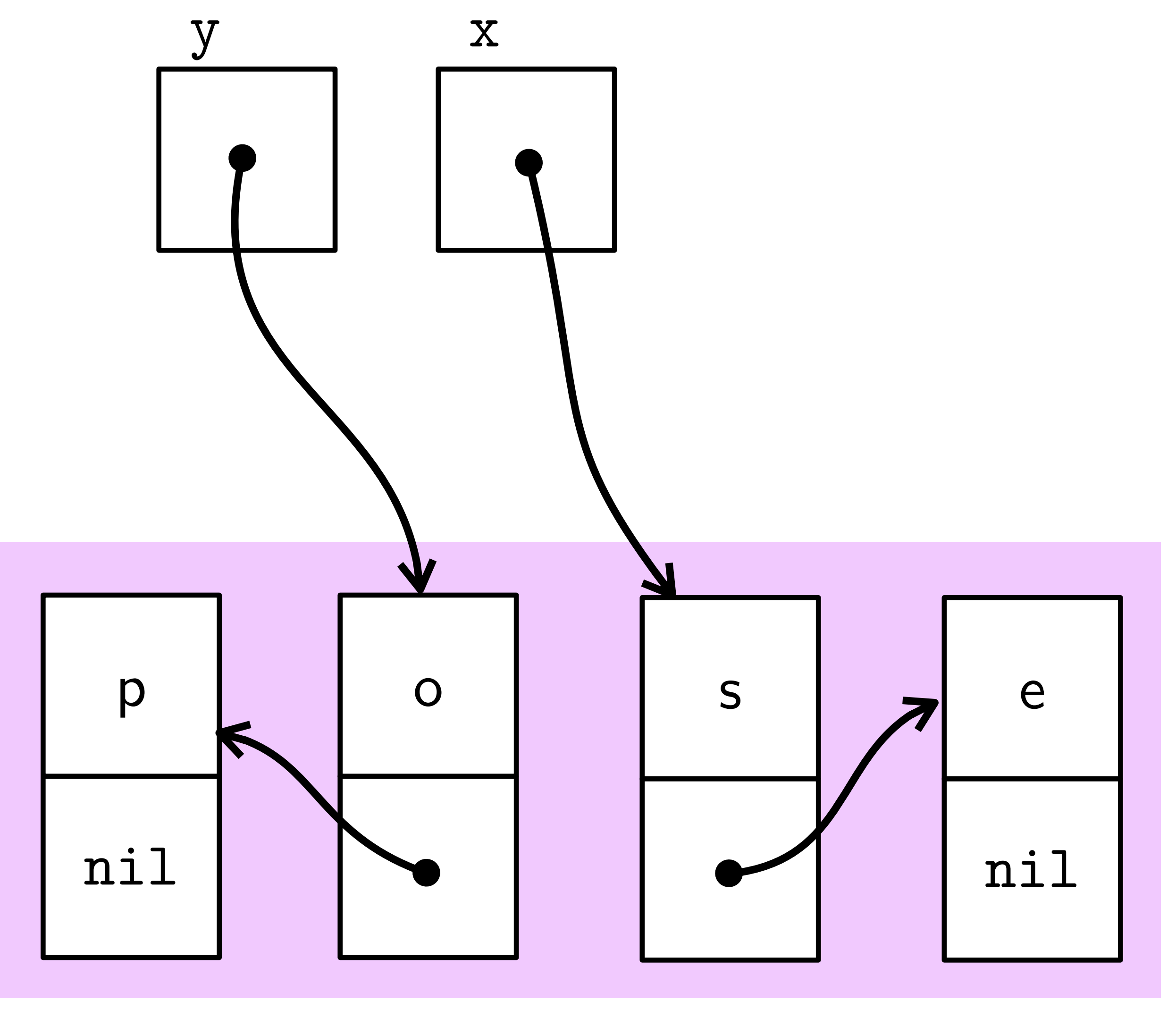
```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```



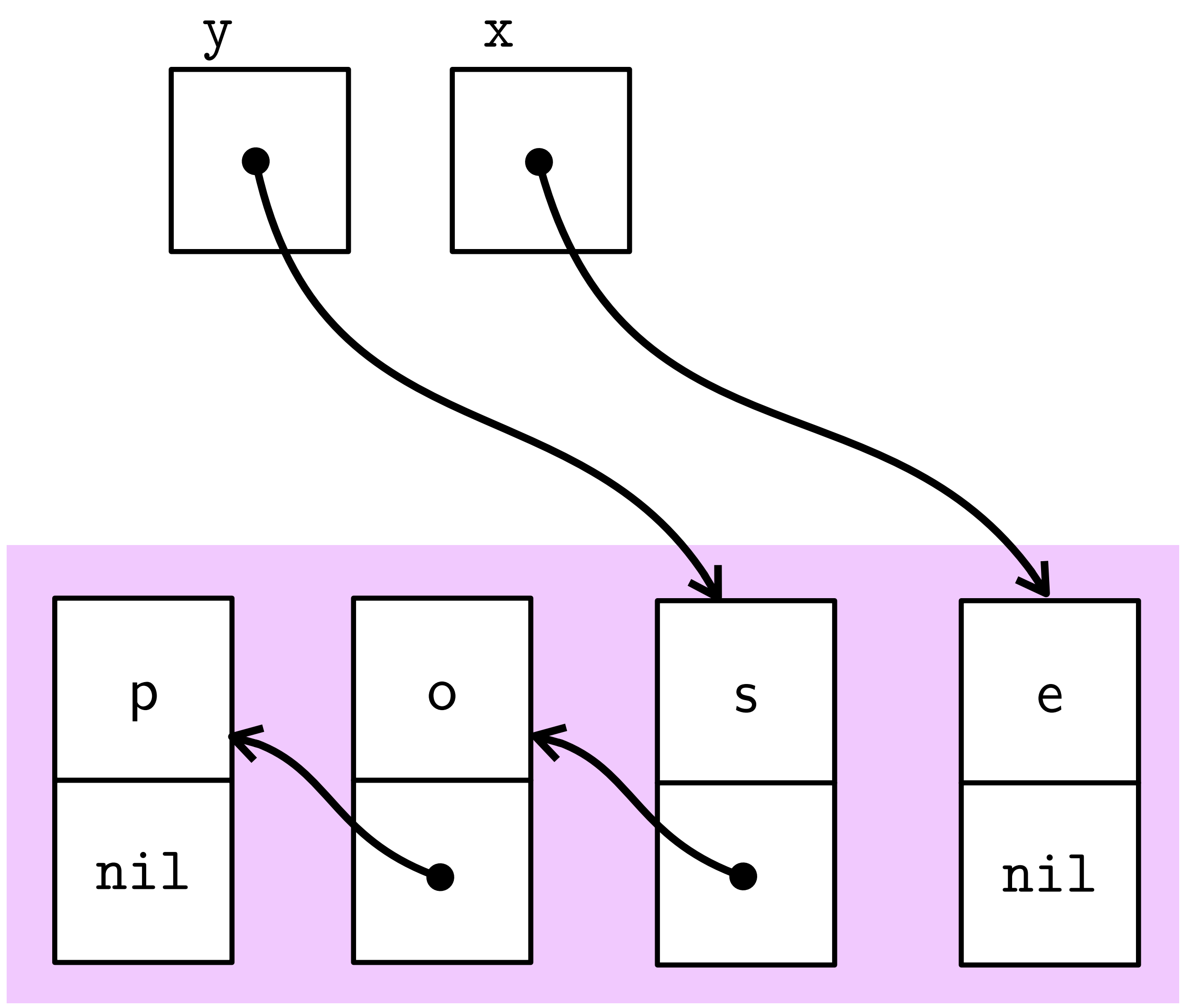
```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```



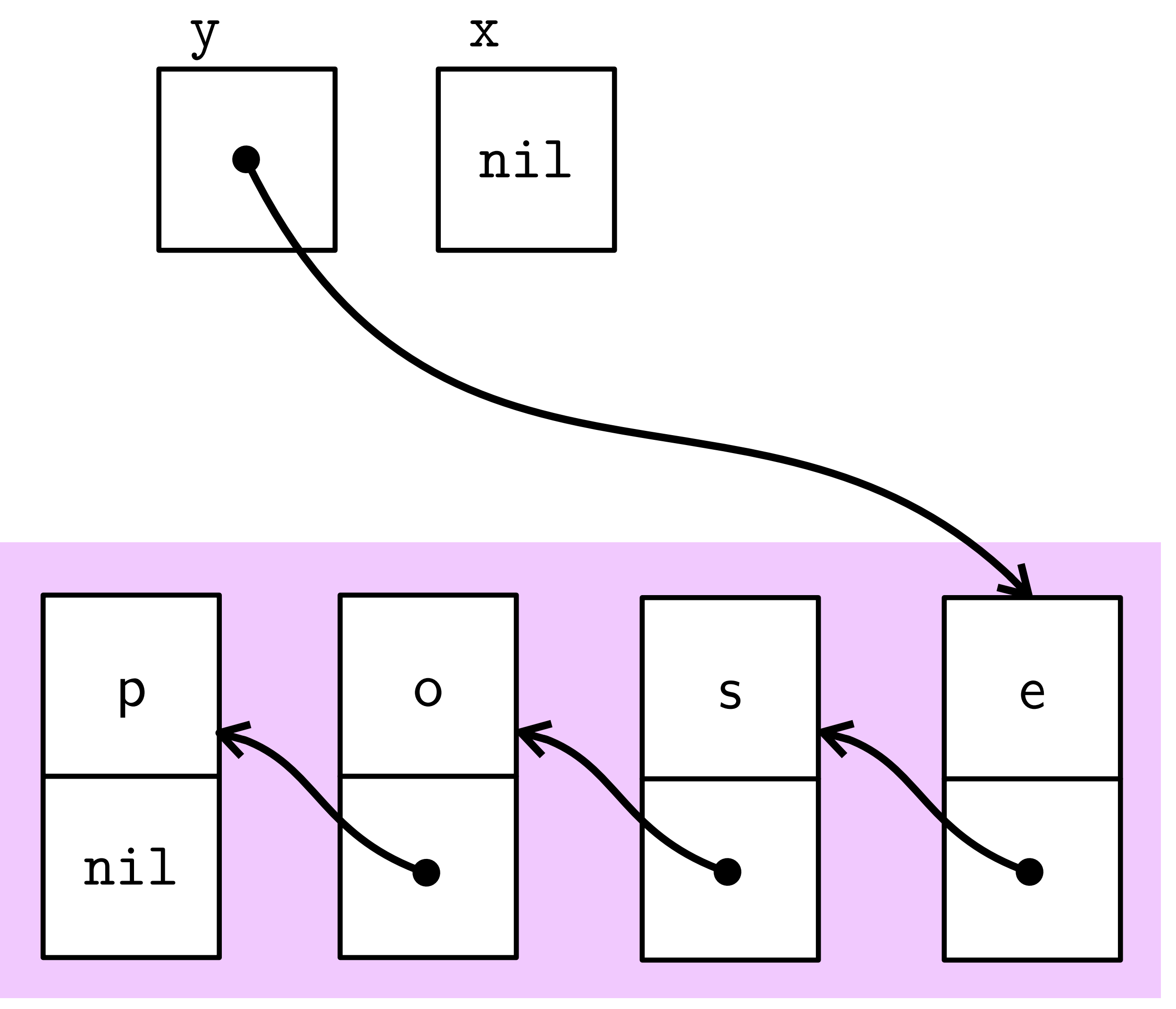
```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```



```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```



```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```



$list\ \alpha_0\ \mathtt{x}$

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

$list\ \alpha_0^{\dagger}\ \mathtt{y}$

$$list\ \epsilon\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil})$$

$$list\ (i \cdot \alpha')\ x \stackrel{\text{def}}{=} (\exists x'.\ x \mapsto i, x' * list\ \alpha'\ x')$$

$list\ \alpha_0\ \texttt{x}$

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

$list\ \alpha_0^{\dagger}\ \texttt{y}$

$$list\ x \stackrel{\text{def}}{=} (x \doteq \mathtt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

list x

$$list\,x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * list\,x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

list y

$$list\ x \stackrel{\text{def}}{=} (x \doteq \mathtt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

list x

```
y:=nil
```

list y

list y

$$\mathit{list}\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\ x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y

list x

```
y:=nil
```

list y

```
while (x!=nil) {
}
```

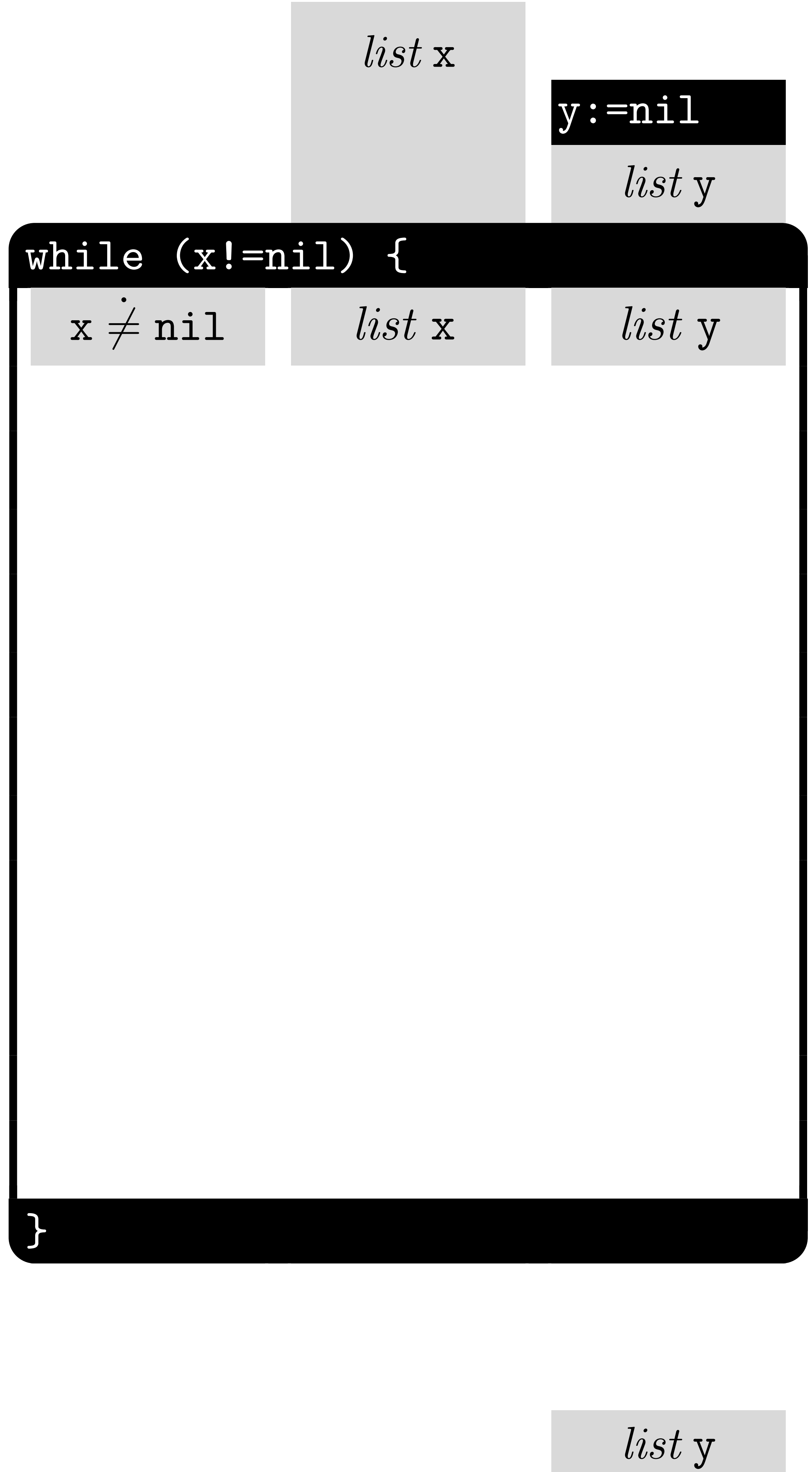
list y

$$list\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \lor (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

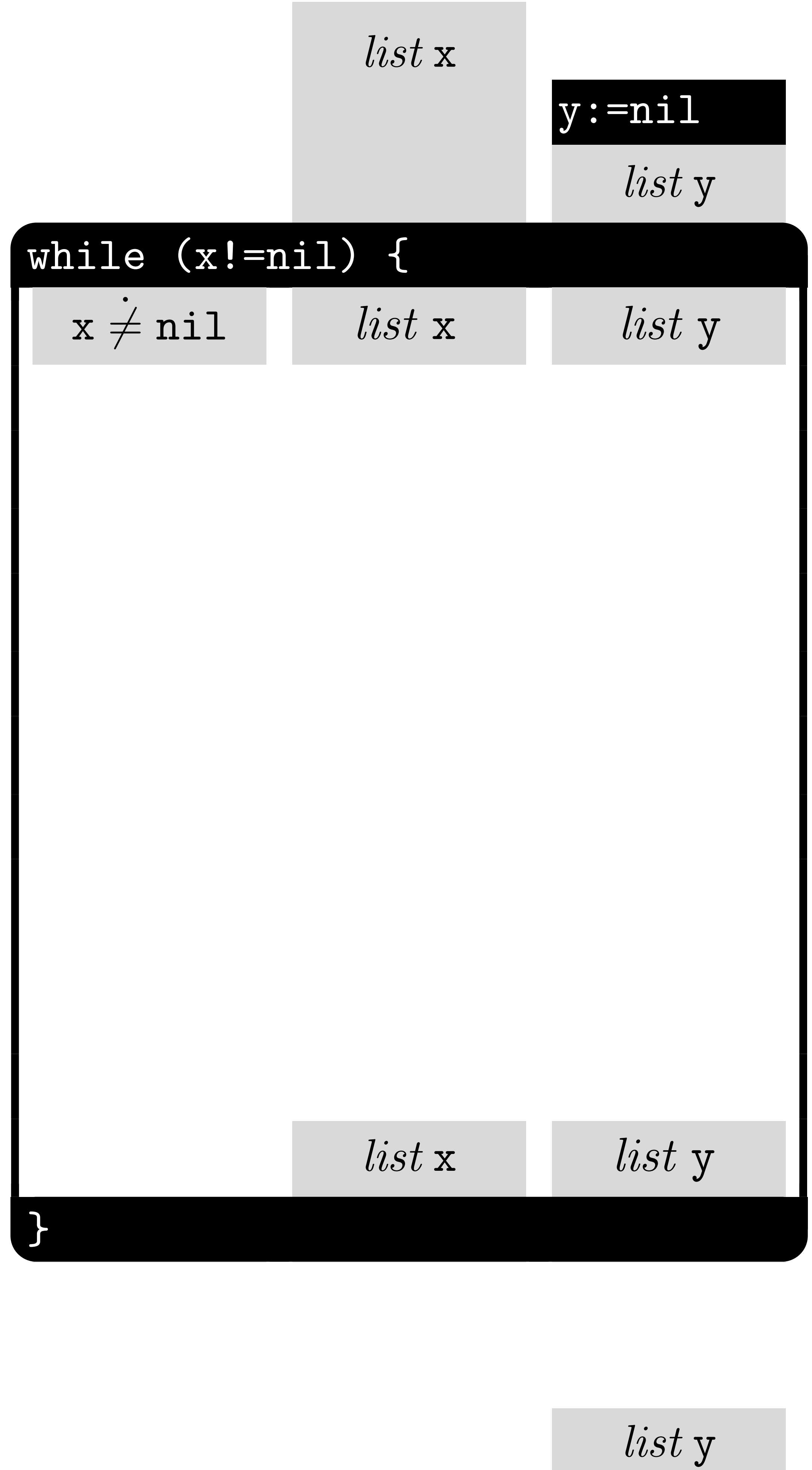


$$list\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y



$$list\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \lor (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

list x

`y:=nil`

list y

`while (x!=nil) {`

x $\dot{\neq}$ nil | *list* x | *list* y

list x | *list* y

`}`

x $\doteq$ nil | *list* x | *list* y

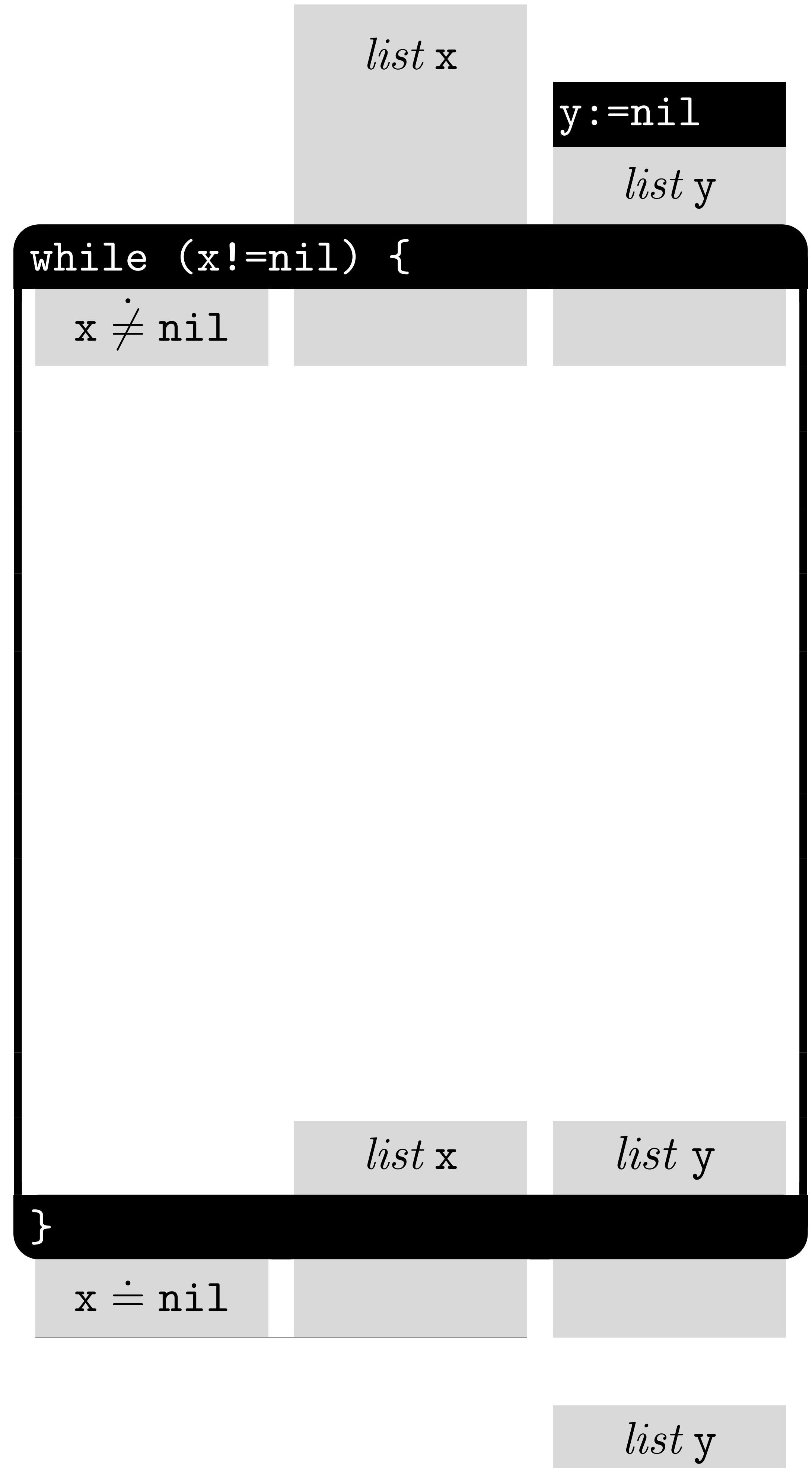
list y

$$\mathit{list}\; x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\; x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y



$$\mathit{list}\ x \stackrel{\text{def}}{=} (x \doteq \mathtt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\ x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y

list x

`y:=nil`

list y

`while (x!=nil) {`

$\mathtt{x} \not\doteq \mathtt{nil}$

Unfold *list* def

$\exists Z.\, \mathtt{x} \mapsto \_, Z * \mathit{list}\ Z$

list x

list y

`}`

$\mathtt{x} \doteq \mathtt{nil}$

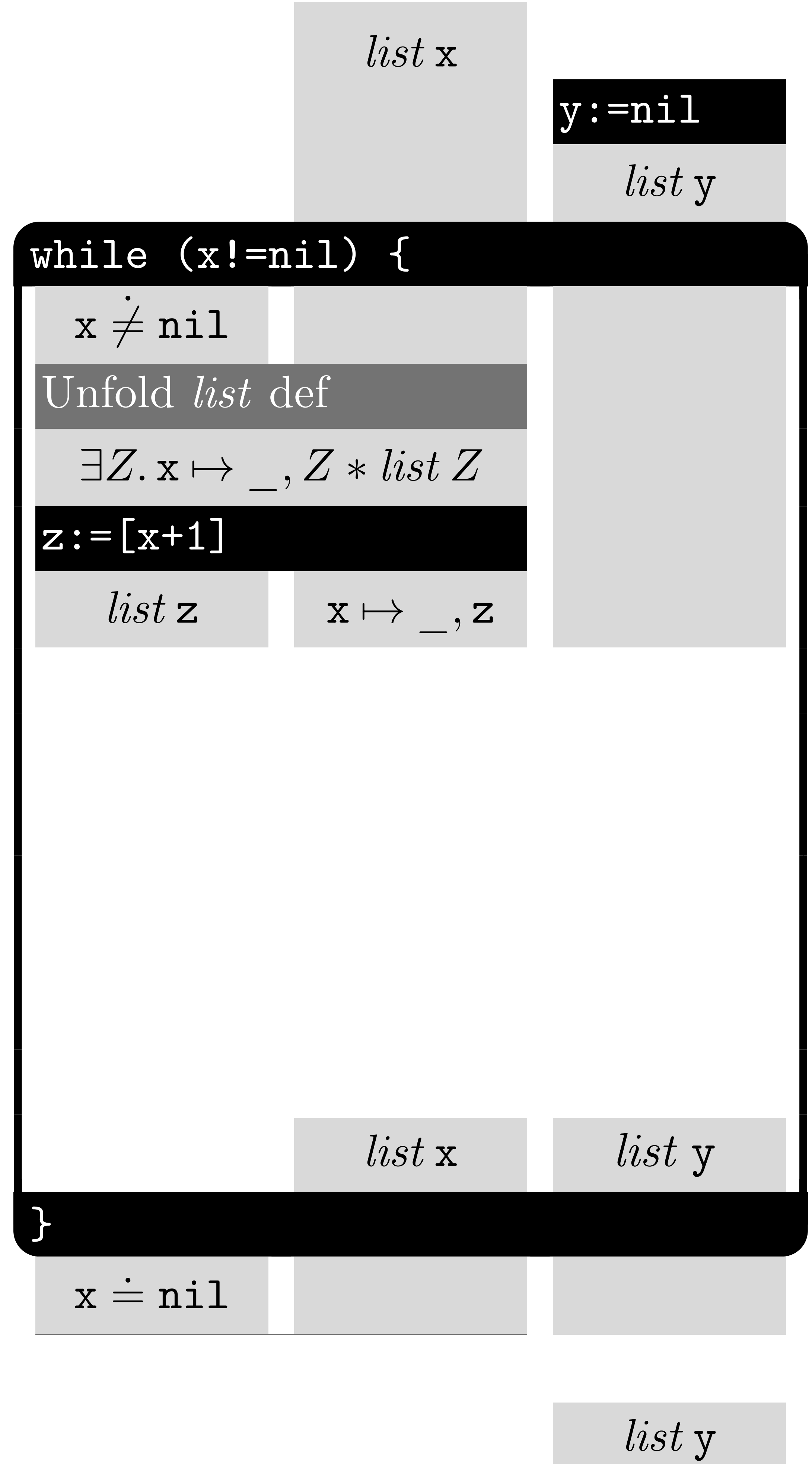
list y

$$list\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y



$$\mathit{list}\ x \stackrel{\text{def}}{=} (x \doteq \mathtt{nil}) \vee (\exists x'.\ x \mapsto \_, x' * \mathit{list}\ x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y

list x

`y:=nil`

list y

`while (x!=nil) {`

$\mathtt{x} \not\doteq \mathtt{nil}$

Unfold *list* def

$\exists Z.\ \mathtt{x} \mapsto \_, Z * \mathit{list}\ Z$

`z:=[x+1]`

list z

$\mathtt{x} \mapsto \_, \mathtt{z}$

`[x+1]:=y`

$\mathtt{x} \mapsto \_, \mathtt{y}$

list x

list y

`}`

$\mathtt{x} \doteq \mathtt{nil}$

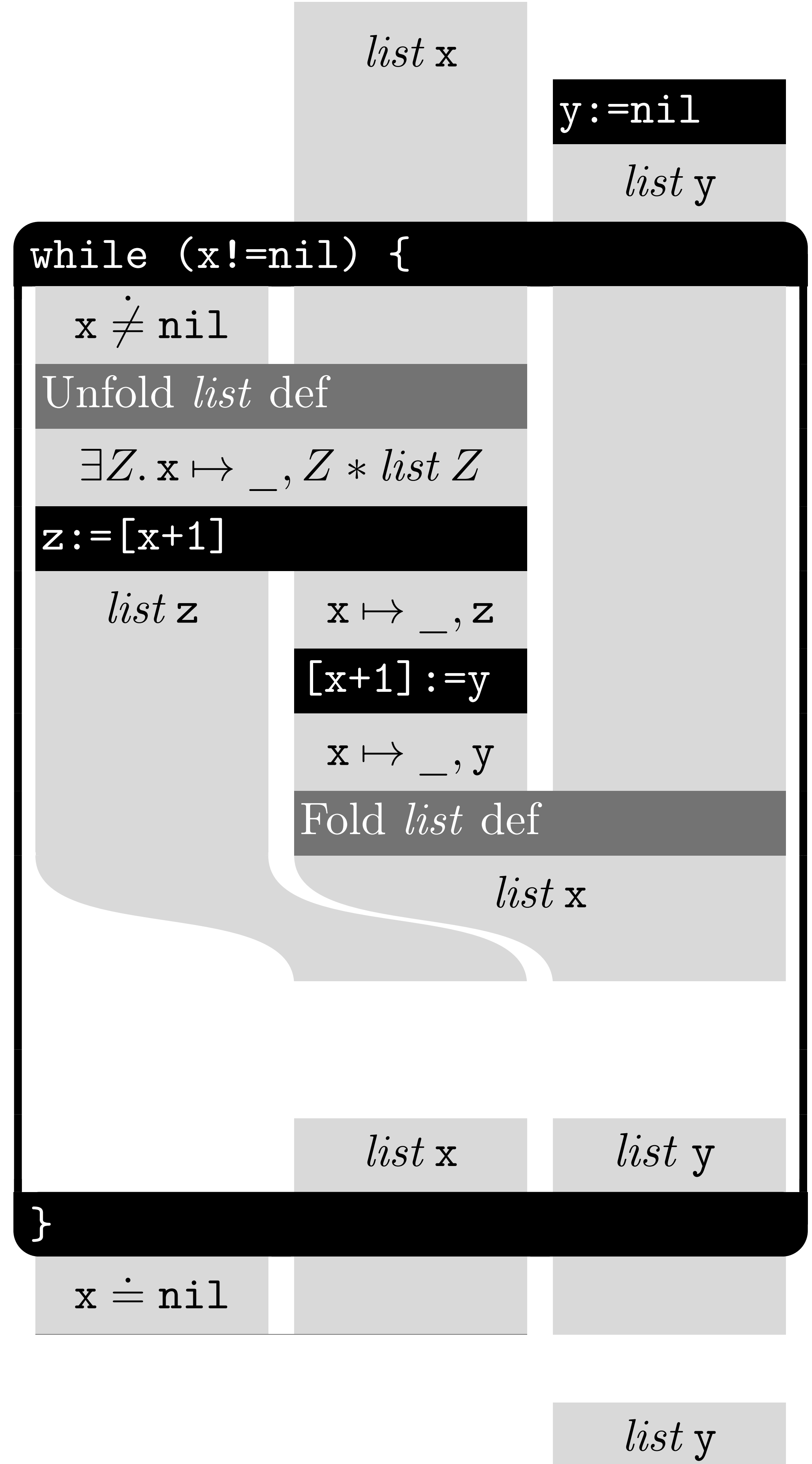
list y

$$\mathit{list}\, x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\, x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

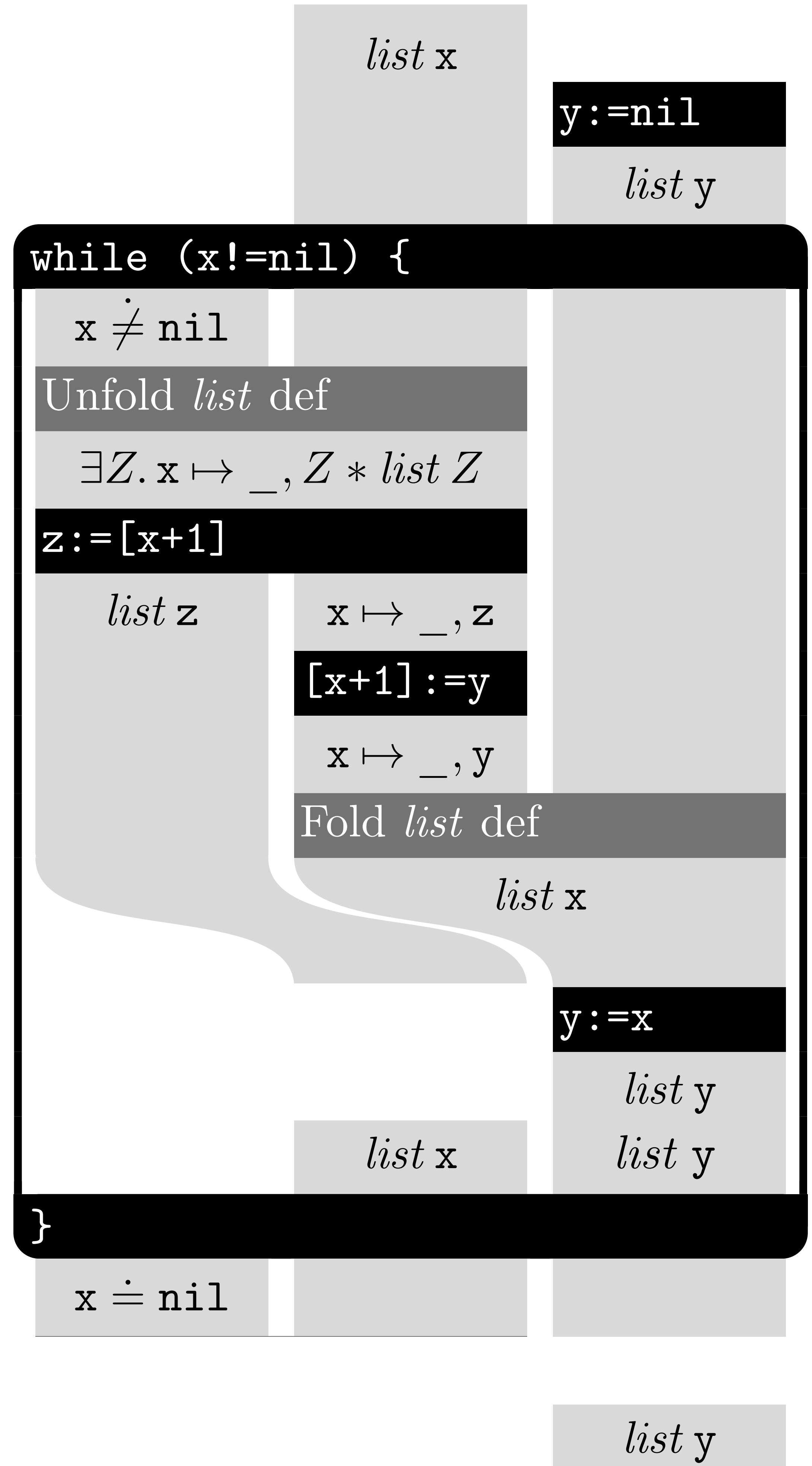


$$\mathit{list}\, x \stackrel{\text{def}}{=} (x \doteq \mathtt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\, x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y

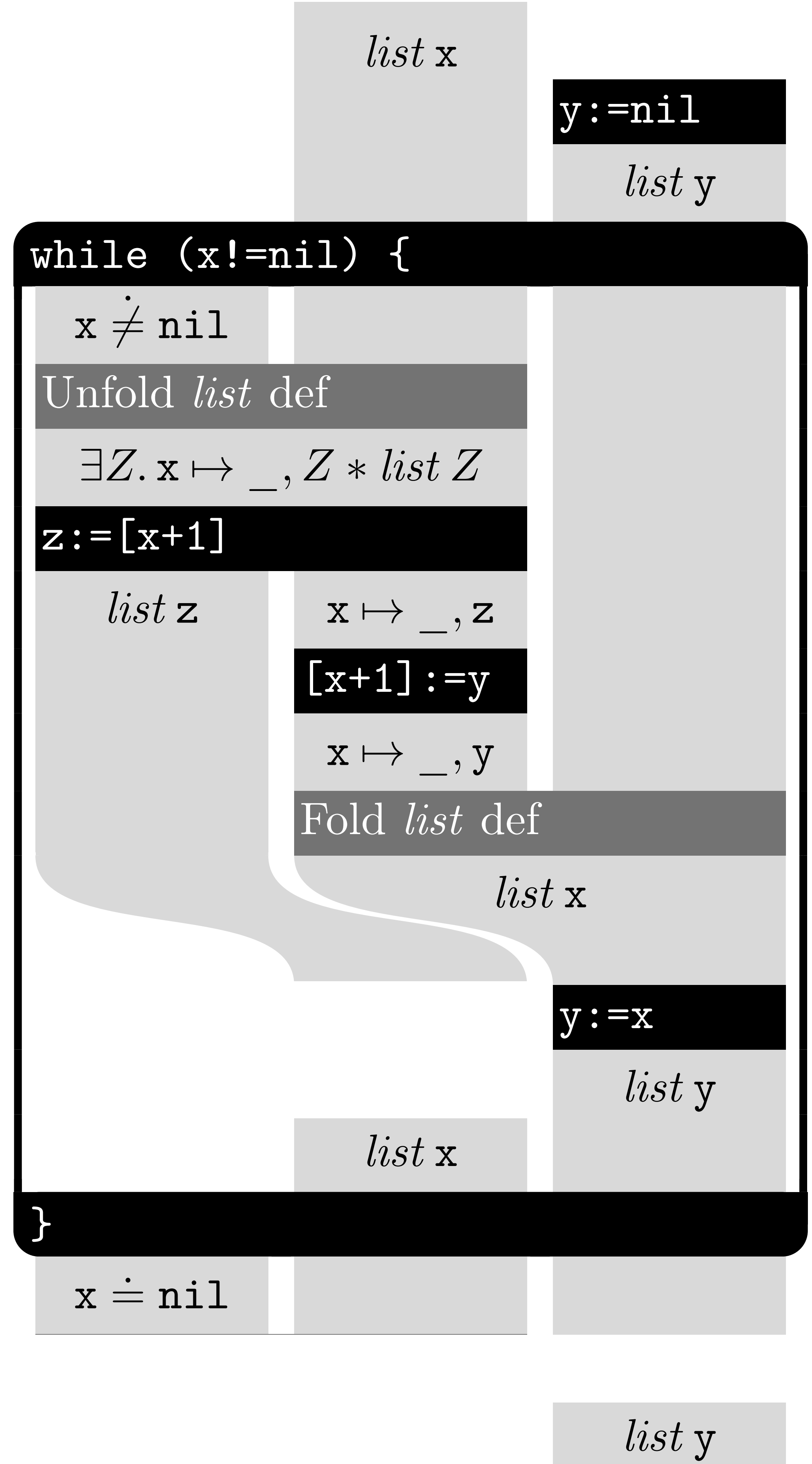


$$\mathit{list}\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\ x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

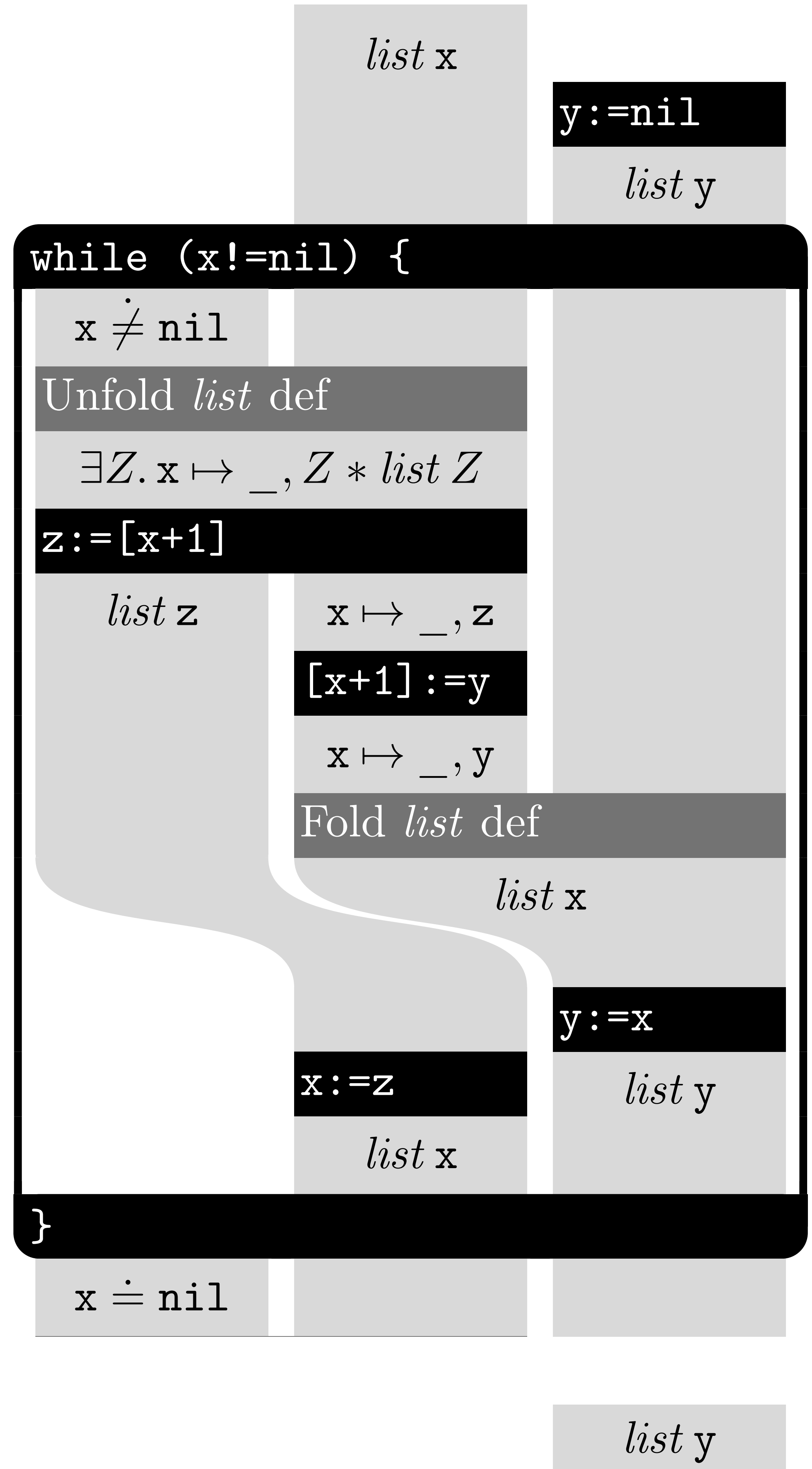


$$list\ x \stackrel{\mathrm{def}}{=} (x \doteq \mathtt{nil}) \lor (\exists x'.\, x \mapsto \_, x' * list\ x')$$

list x

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

list y

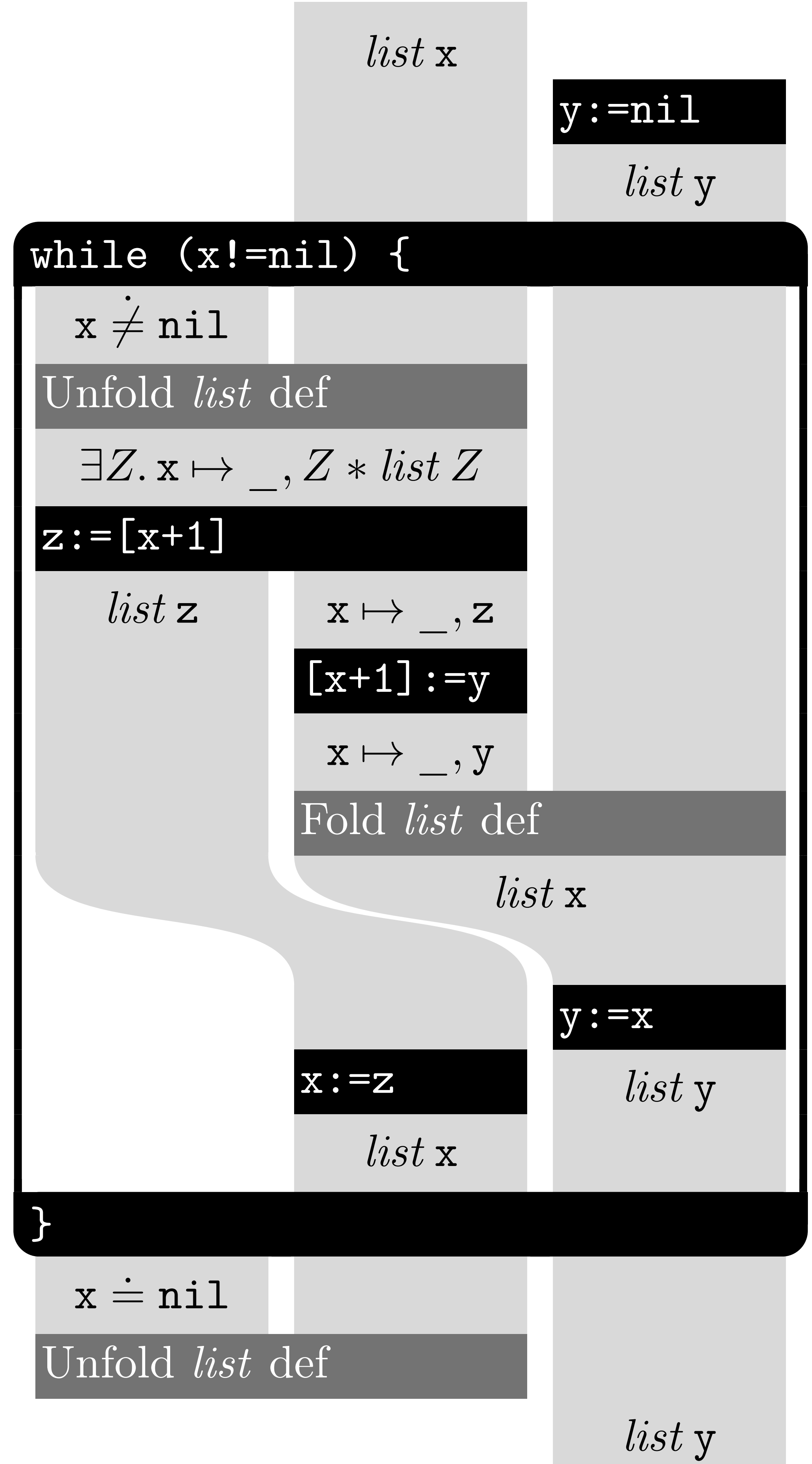


$$\mathit{list}\,x \stackrel{\text{def}}{=} (x \doteq \texttt{nil}) \vee (\exists x'.\, x \mapsto \_, x' * \mathit{list}\,x')$$

list x

```
y := nil;
while (x != nil) {
    z := [x+1];
    [x+1] := y;
    y := x;
    x := z;
}
```

list y



Dealing with quantifiers

$$list\ \epsilon\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil})$$

$$list\ (i \cdot \alpha')\ x \stackrel{\text{def}}{=} (\exists x'.\ x \mapsto i, x' * list\ \alpha'\ x')$$

$list\ \alpha_0\ \texttt{x}$

```
y := nil;
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

$list\ \alpha_0^{\dagger}\ \texttt{y}$

$$list\ \epsilon\ x \stackrel{\text{def}}{=} (x \doteq \texttt{nil})$$

$$list\ (i \cdot \alpha')\ x \stackrel{\text{def}}{=} (\exists x'.\ x \mapsto i, x' * list\ \alpha'\ x')$$

$$list\ \alpha_0\ \mathtt{x}$$

```
y := nil;
```

$$\exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha$$

```
while (x != nil) {
   z := [x+1];
   [x+1] := y;
   y := x;
   x := z;
}
```

$$list\ \alpha_0^\dagger\ \mathtt{y}$$

```
{list α₀ x}
y:=nil;
{list α₀ x * list ε y}
// Choose α := α₀ and β := ε
while {∃α, β. list α x * list β y * α₀ ≐ β† · α}
(x!=nil) {
  {∃α, β. x ≠ nil * list α x * list β y
   * α₀ ≐ β† · α}
  // Unfold list def
  {∃α, β. (∃α′, i, Z. x ↦ i, Z * list α′ z
   * α ≐ i · α′) * list β y * α₀ ≐ β† · α}
  // Choose α := α′
  {∃α, β, i, Z. x ↦ i, Z * list α Z
   * α₀ ≐ β† · (i · α) * list β y}
  z:=[x+1];
  {∃α, β, i. list α z * x ↦ i, z
   * α₀ ≐ β† · (i · α) * list β y}
  // Reassociate i
  {∃α, β, i. list α z * x ↦ i, z
   * α₀ ≐ (i · β)† · α * list β y}
  [x+1]:=y;
  {∃α, β, i. list α z * x ↦ i, y
   * α₀ ≐ (i · β)† · α * list β y}
  // Fold list def
  {∃α, β, i. list α z * list (i · β) x
   * α₀ ≐ (i · β)† · α}
  // Choose β := (i · β)
  {∃α, β. list α z * list β x * α₀ ≐ β† · α}
  y:=x;
  {∃α, β. list α z * list β y * α₀ ≐ β† · α}
  x:=z;
  {∃α, β. list α x * list β y * α₀ ≐ β† · α}
}
{∃α, β. x ≐ nil * list α x * list β y
 * α₀ ≐ β† · α}
// Unfold list def
{∃α, β. α ≐ ε * list β y * α₀ ≐ β† · α}
// Concatenate empty sequence
{∃β. list β y * α₀ ≐ β†}
// Fold list def
{list α₀† y}
```

```
y:=nil;
```

$\{list\ \alpha_0\ \mathtt{x} * list\ \epsilon\ \mathtt{y}\}$

`// Choose` $\alpha := \alpha_0$ `and` $\beta := \epsilon$

`while` $\{\exists\alpha,\beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$

`(x!=nil) {`

$$\left\{\begin{array}{l}\exists\alpha,\beta.\ \mathtt{x} \mathrel{\dot{\neq}} \mathtt{nil} * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$$

`// Unfold` *list* `def`

$$\left\{\begin{array}{l}\exists\alpha,\beta.\ (\exists\alpha', i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha'\ \mathtt{z} \\ * \alpha \doteq i \cdot \alpha') * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$$

`// Choose` $\alpha := \alpha'$

$$\left\{\begin{array}{l}\exists\alpha,\beta,i,Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha\ Z \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y}\end{array}\right\}$$

```
z:=[x+1];
```

$$\left\{\begin{array}{l}\exists\alpha,\beta,i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y}\end{array}\right\}$$

`// Reassociate` i

$$\left\{\begin{array}{l}\exists\alpha,\beta,i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y}\end{array}\right\}$$

```
[x+1]:=y;
```

$$\left\{\begin{array}{l}\exists\alpha,\beta,i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{y} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y}\end{array}\right\}$$

`// Fold` *list* `def`

$$\left\{\begin{array}{l}\exists\alpha,\beta,i.\ list\ \alpha\ \mathtt{z} * list\ (i \cdot \beta)\ \mathtt{x} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha\end{array}\right\}$$

`// Choose` $\beta := (i \cdot \beta)$

```
y:=nil;
```

$\{list\ \alpha_0\ \mathtt{x} * list\ \epsilon\ \mathtt{y}\}$

// Choose $\alpha := \alpha_0$ and $\beta := \epsilon$

while $\{\exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$

(x!=nil) {

$$\left\{\begin{array}{l}\exists \alpha, \beta.\ \mathtt{x} \mathrel{\dot{\neq}} \mathtt{nil} * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$$

// Unfold *list* def

$$\left\{\begin{array}{l}\exists \alpha, \beta. (\exists \alpha', i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha'\ \mathtt{z} \\ * \alpha = i \cdot \alpha') * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$$

// Choose $\alpha := \alpha'$

$$\left\{\begin{array}{l}\exists \alpha, \beta, i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha\ Z \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y}\end{array}\right\}$$

```
z:=[x+1];
```

$$\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y}\end{array}\right\}$$

// Reassociate i

$$\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y}\end{array}\right\}$$

```
[x+1]:=y;
```

$$\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{y} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y}\end{array}\right\}$$

// Fold *list* def

$$\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * list\ (i \cdot \beta)\ \mathtt{x} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha\end{array}\right\}$$

// Choose $\beta := (i \cdot \beta)$

$\{\mathit{list}\,\alpha_0\,\mathtt{x}\}$
`y:=nil;`
$\{\mathit{list}\,\alpha_0\,\mathtt{x} * \mathit{list}\,\epsilon\,\mathtt{y}\}$
`// Choose` $\alpha := \alpha_0$ `and` $\beta := \epsilon$
`while` $\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`(x!=nil) {`
$\left\{\begin{array}{l}\exists\alpha,\beta.\,\mathtt{x} \not\doteq \mathtt{nil} * \mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$
`// Unfold` *list* `def`
$\left\{\begin{array}{l}\exists\alpha,\beta.\,(\exists\alpha',i,Z.\,\mathtt{x} \mapsto i,Z * \mathit{list}\,\alpha'\,\mathtt{z} \\ * \alpha \doteq i\cdot\alpha') * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$
`// Choose` $\alpha := \alpha'$
$\left\{\begin{array}{l}\exists\alpha,\beta,i,Z.\,\mathtt{x} \mapsto i,Z * \mathit{list}\,\alpha\,Z \\ * \alpha_0 \doteq \beta^\dagger \cdot (i\cdot\alpha) * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$
`z:=[x+1];`
$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathtt{x} \mapsto i,\mathtt{z} \\ * \alpha_0 \doteq \beta^\dagger \cdot (i\cdot\alpha) * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$
`// Reassociate` i
$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathtt{x} \mapsto i,\mathtt{z} \\ * \alpha_0 \doteq (i\cdot\beta)^\dagger \cdot \alpha * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$
`[x+1]:=y;`
$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathtt{x} \mapsto i,\mathtt{y} \\ * \alpha_0 \doteq (i\cdot\beta)^\dagger \cdot \alpha * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$
`// Fold` *list* `def`
$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathit{list}\,(i\cdot\beta)\,\mathtt{x} \\ * \alpha_0 \doteq (i\cdot\beta)^\dagger \cdot \alpha\end{array}\right\}$
`// Choose` $\beta := (i\cdot\beta)$
$\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathit{list}\,\beta\,\mathtt{x} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`y:=x;`
$\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`x:=z;`
$\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`}`
$\left\{\begin{array}{l}\exists\alpha,\beta.\,\mathtt{x} \doteq \mathtt{nil} * \mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$
`// Unfold` *list* `def`
$\{\exists\alpha,\beta.\,\alpha \doteq \epsilon * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`// Concatenate empty sequence`
$\{\exists\beta.\,\mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger\}$
`// Fold` *list* `def`
$\{\mathit{list}\,\alpha_0^\dagger\,\mathtt{y}\}$

$\{\mathit{list}\,\alpha_0\,\mathtt{x}\}$
`y:=nil;`
$\{\mathit{list}\,\alpha_0\,\mathtt{x} * \mathit{list}\,\epsilon\,\mathtt{y}\}$
`// Choose` $\alpha := \alpha_0$ `and` $\beta := \epsilon$
`while` $\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`(x!=nil) {`

$\left\{\begin{array}{l}\exists\alpha,\beta.\,\mathtt{x} \not\doteq \mathtt{nil} * \mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$

`// Unfold` *list* `def`

$\left\{\begin{array}{l}\exists\alpha,\beta.\,(\exists\alpha',i,Z.\,\mathtt{x} \mapsto i,Z * \mathit{list}\,\alpha'\,\mathtt{z} \\ * \alpha \doteq i \cdot \alpha') * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$

`// Choose` $\alpha := \alpha'$

$\left\{\begin{array}{l}\exists\alpha,\beta,i,Z.\,\mathtt{x} \mapsto i,Z * \mathit{list}\,\alpha\,Z \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$

`z:=[x+1];`

$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathtt{x} \mapsto i,\mathtt{z} \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$

`// Reassociate` i

$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathtt{x} \mapsto i,\mathtt{z} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$

`[x+1]:=y;`

$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathtt{x} \mapsto i,\mathtt{y} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * \mathit{list}\,\beta\,\mathtt{y}\end{array}\right\}$

`// Fold` *list* `def`

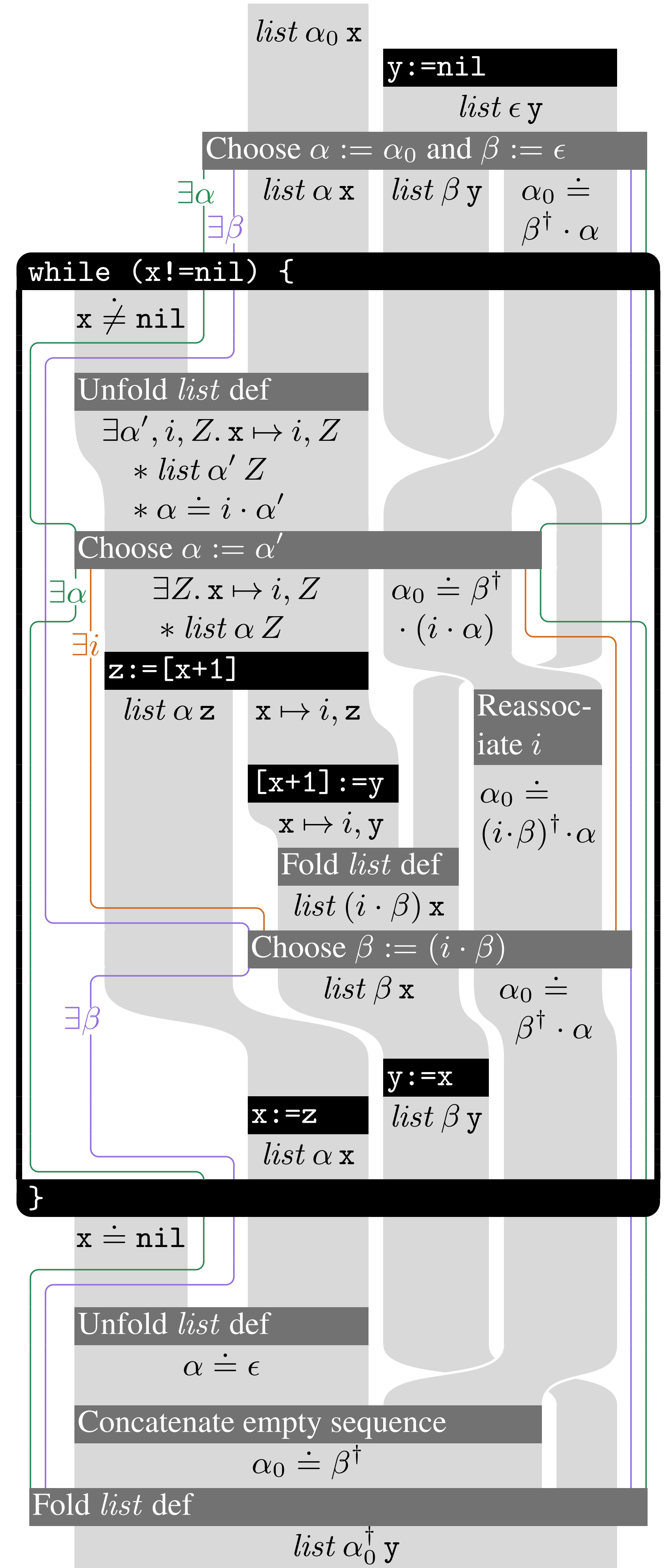
$\left\{\begin{array}{l}\exists\alpha,\beta,i.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathit{list}\,(i \cdot \beta)\,\mathtt{x} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha\end{array}\right\}$

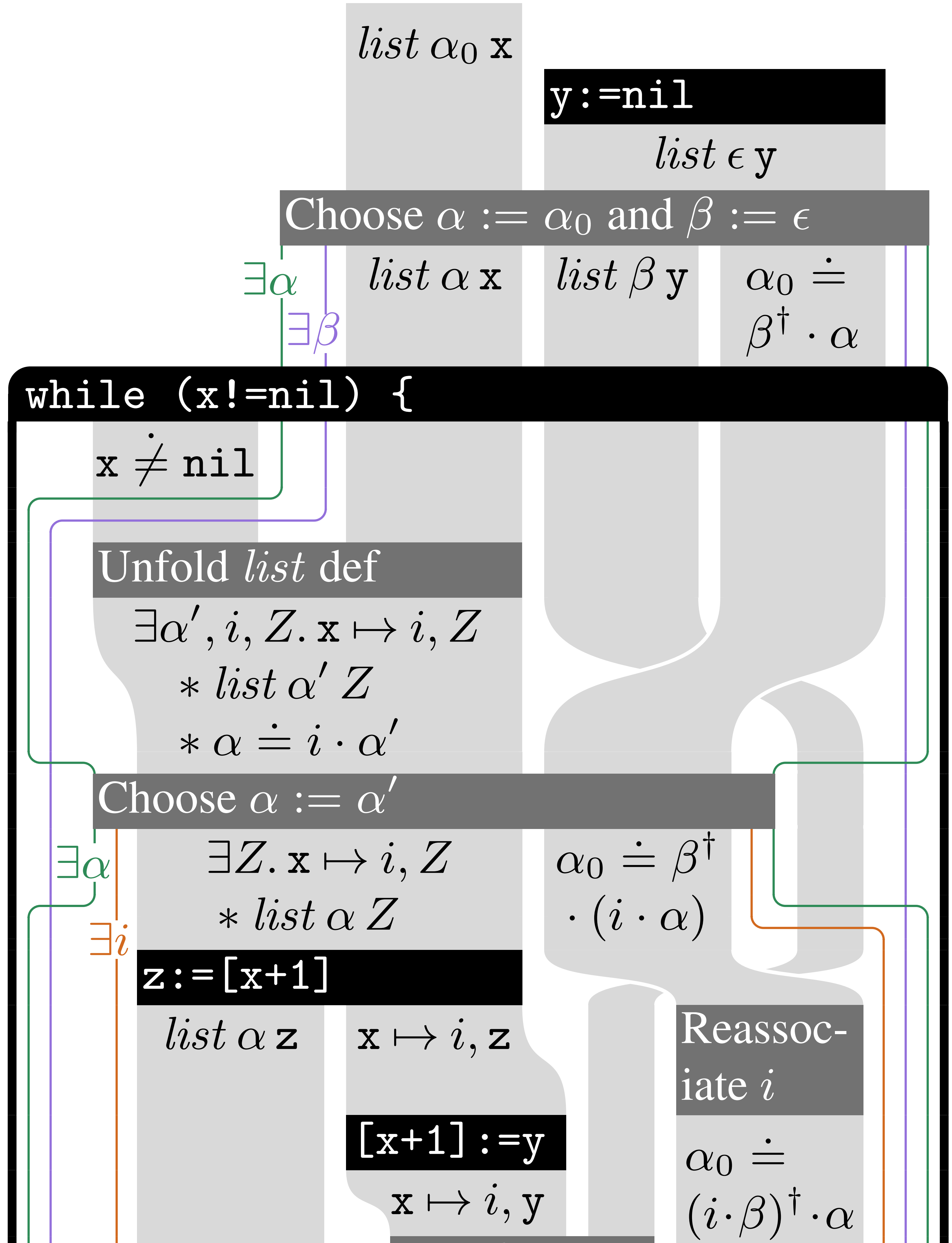
`// Choose` $\beta := (i \cdot \beta)$

$\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathit{list}\,\beta\,\mathtt{x} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`y:=x;`
$\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{z} * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`x:=z;`
$\{\exists\alpha,\beta.\,\mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`}`

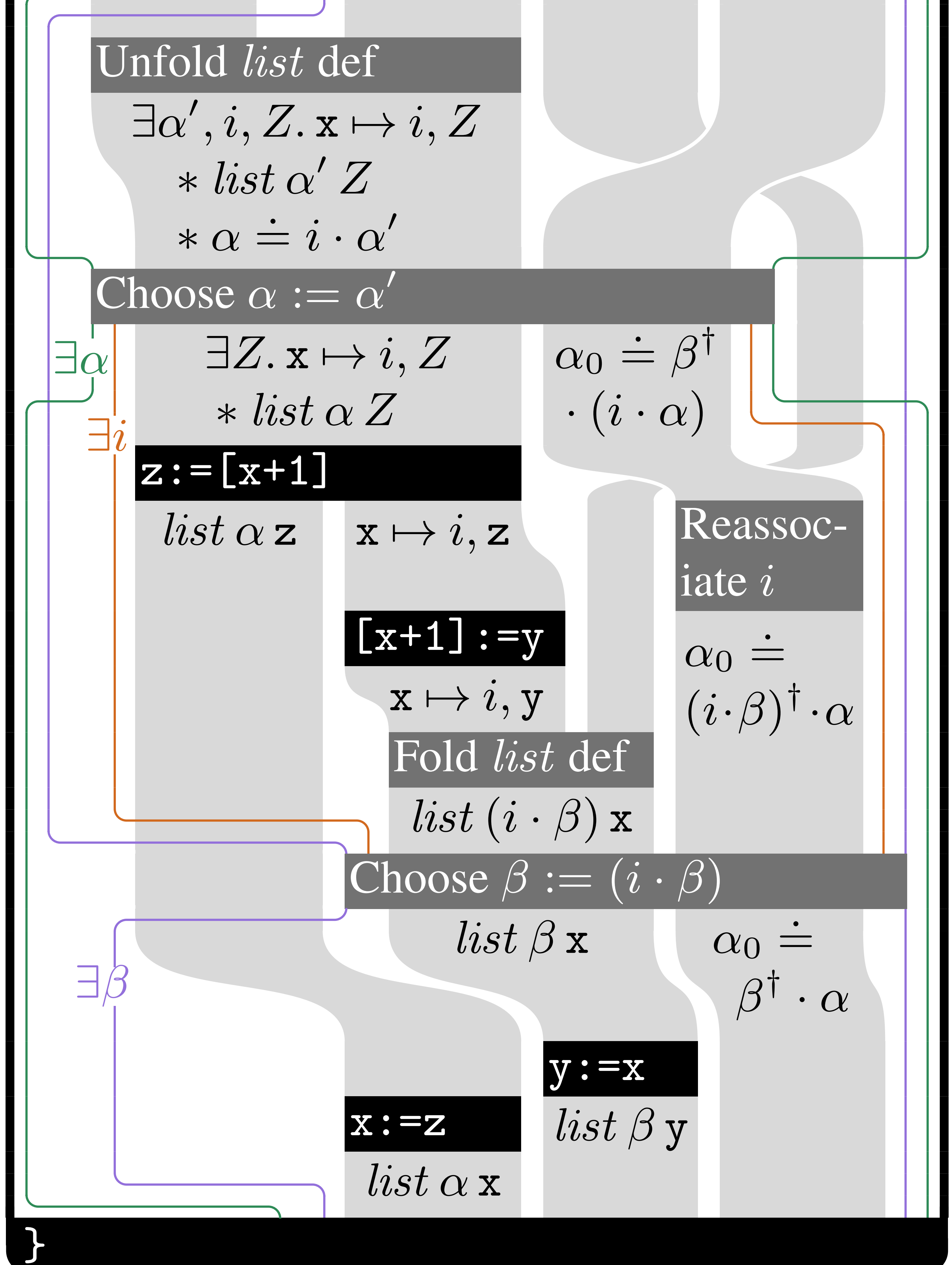
$\left\{\begin{array}{l}\exists\alpha,\beta.\,\mathtt{x} \doteq \mathtt{nil} * \mathit{list}\,\alpha\,\mathtt{x} * \mathit{list}\,\beta\,\mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$

`// Unfold` *list* `def`
$\{\exists\alpha,\beta.\,\alpha \doteq \epsilon * \mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha\}$
`// Concatenate empty sequence`
$\{\exists\beta.\,\mathit{list}\,\beta\,\mathtt{y} * \alpha_0 \doteq \beta^\dagger\}$
`// Fold` *list* `def`
$\{\mathit{list}\,\alpha_0^\dagger\,\mathtt{y}\}$

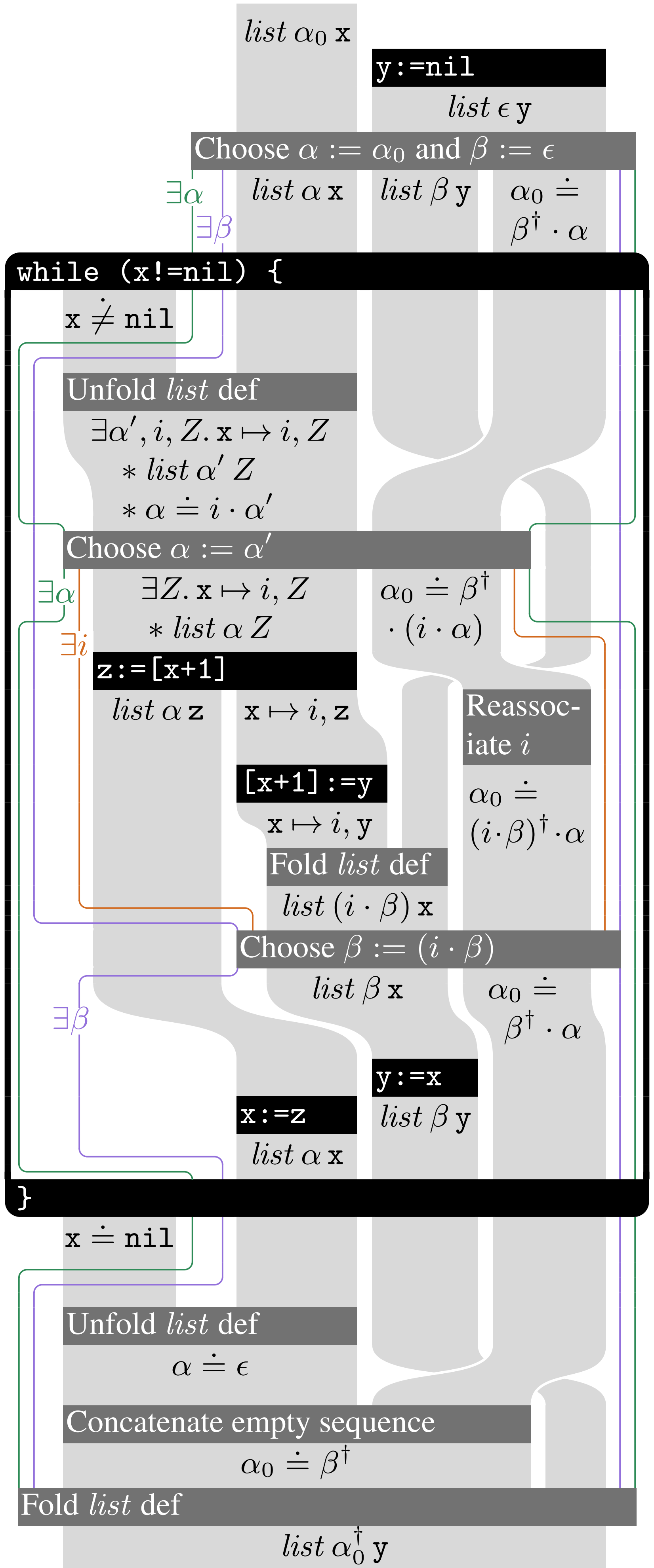


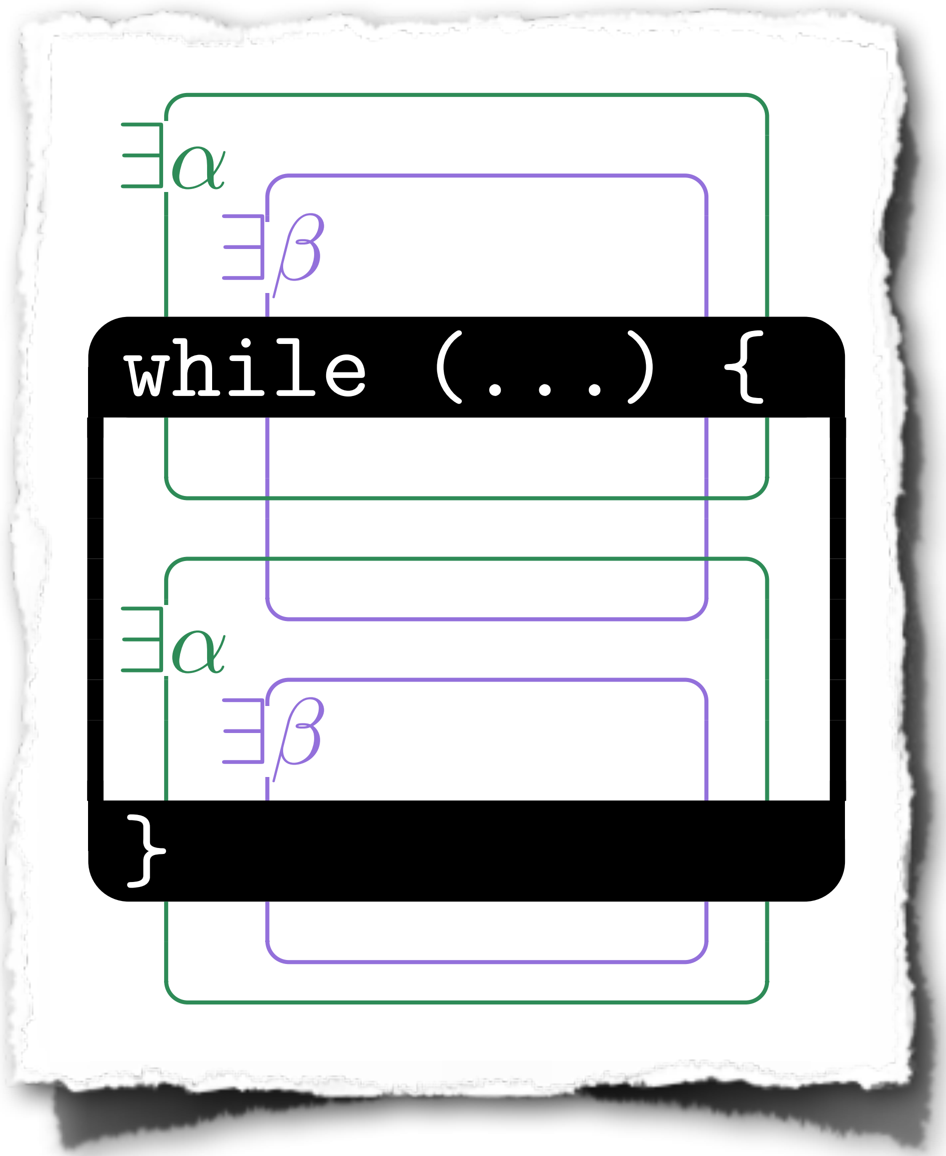


list α0 x
y:=nil
list ϵ y
Choose α := α0 and β := ϵ
∃α
∃β
list α x
list β y
α0 ≐ β† · α
while (x!=nil) {
x ≠ nil
Unfold list def
∃α′, i, Z. x ↦ i, Z
∗ list α′ Z
∗ α ≐ i · α′
Choose α := α′
∃α
∃i
∃Z. x ↦ i, Z
∗ list α Z
α0 ≐ β† · (i · α)
z:=[x+1]
list α z
x ↦ i, z
Reassoc-
iate i
[x+1]:=y
x ↦ i, y
α0 ≐ (i·β)† · α

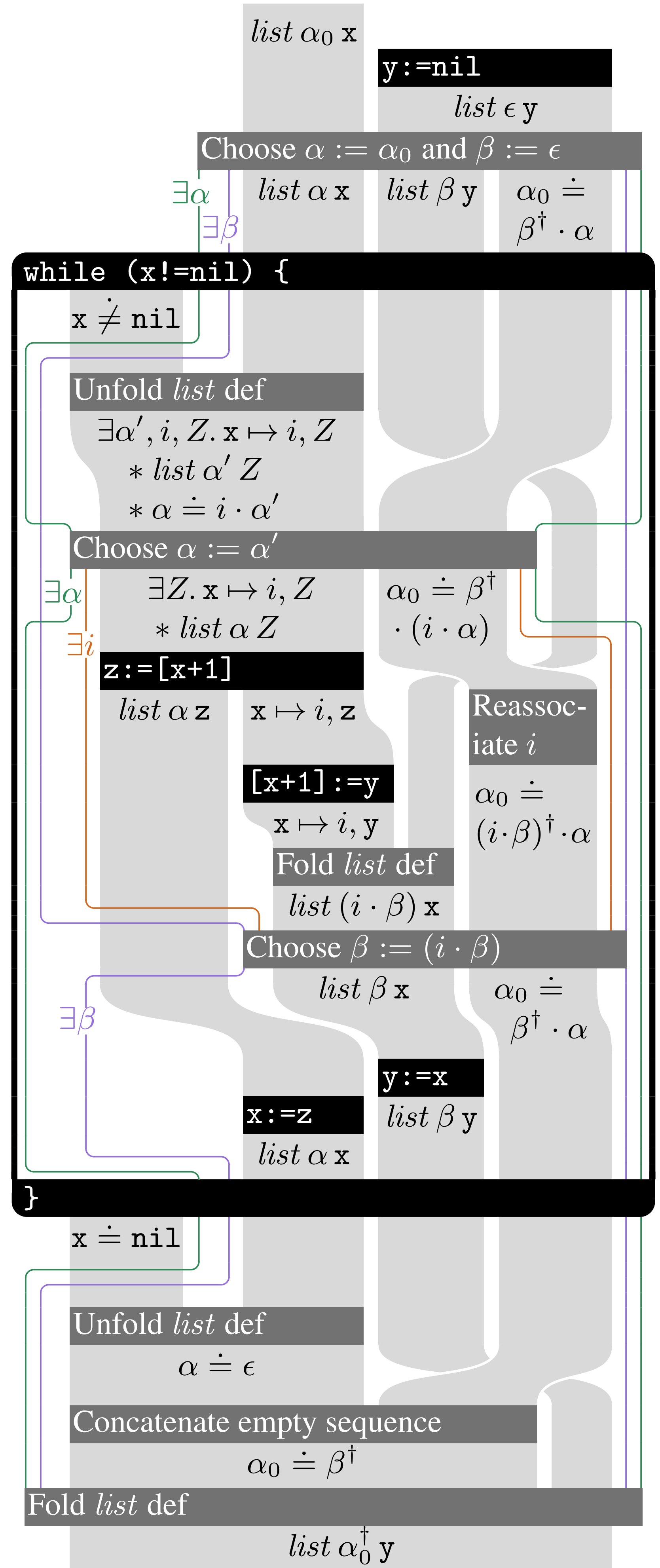


Unfold *list* def
$\exists \alpha', i, Z.\, \mathtt{x} \mapsto i, Z$
$* \; list\; \alpha'\; Z$
$* \; \alpha \doteq i \cdot \alpha'$
Choose $\alpha := \alpha'$
$\exists \alpha$
$\exists Z.\, \mathtt{x} \mapsto i, Z$
$* \; list\; \alpha\; Z$
$\alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha)$
$\exists i$
z:=[x+1]
$list\; \alpha\; \mathtt{z}$
$\mathtt{x} \mapsto i, \mathtt{z}$
Reassociate i
[x+1]:=y
$\mathtt{x} \mapsto i, \mathtt{y}$
$\alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha$
Fold *list* def
$list\; (i \cdot \beta)\; \mathtt{x}$
Choose $\beta := (i \cdot \beta)$
$list\; \beta\; \mathtt{x}$
$\alpha_0 \doteq \beta^\dagger \cdot \alpha$
$\exists \beta$
y:=x
$list\; \beta\; \mathtt{y}$
x:=z
$list\; \alpha\; \mathtt{x}$
}


list α_0 x
y:=nil
list ϵ y
Choose $\alpha := \alpha_0$ and $\beta := \epsilon$
$\exists\alpha$
$\exists\beta$
list α x
list β y
$\alpha_0 \doteq \beta^\dagger \cdot \alpha$
while (x!=nil) {
x $\dot{\neq}$ nil
Unfold list def
$\exists \alpha', i, Z.$ x $\mapsto i, Z$ $*$ list α' Z $* \alpha \doteq i \cdot \alpha'$
Choose $\alpha := \alpha'$
$\exists\alpha$
$\exists i$
$\exists Z.$ x $\mapsto i, Z$ $*$ list α Z
$\alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha)$
z:=[x+1]
list α z
x $\mapsto i$, z
Reassociate i
[x+1]:=y
x $\mapsto i$, y
$\alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha$
Fold list def
list $(i \cdot \beta)$ x
Choose $\beta := (i \cdot \beta)$
$\exists\beta$
list β x
$\alpha_0 \doteq \beta^\dagger \cdot \alpha$
y:=x
list β y
x:=z
list α x
}
x $\doteq$ nil
Unfold list def
$\alpha \doteq \epsilon$
Concatenate empty sequence
$\alpha_0 \doteq \beta^\dagger$
Fold list def
list $\alpha_0^\dagger$ y

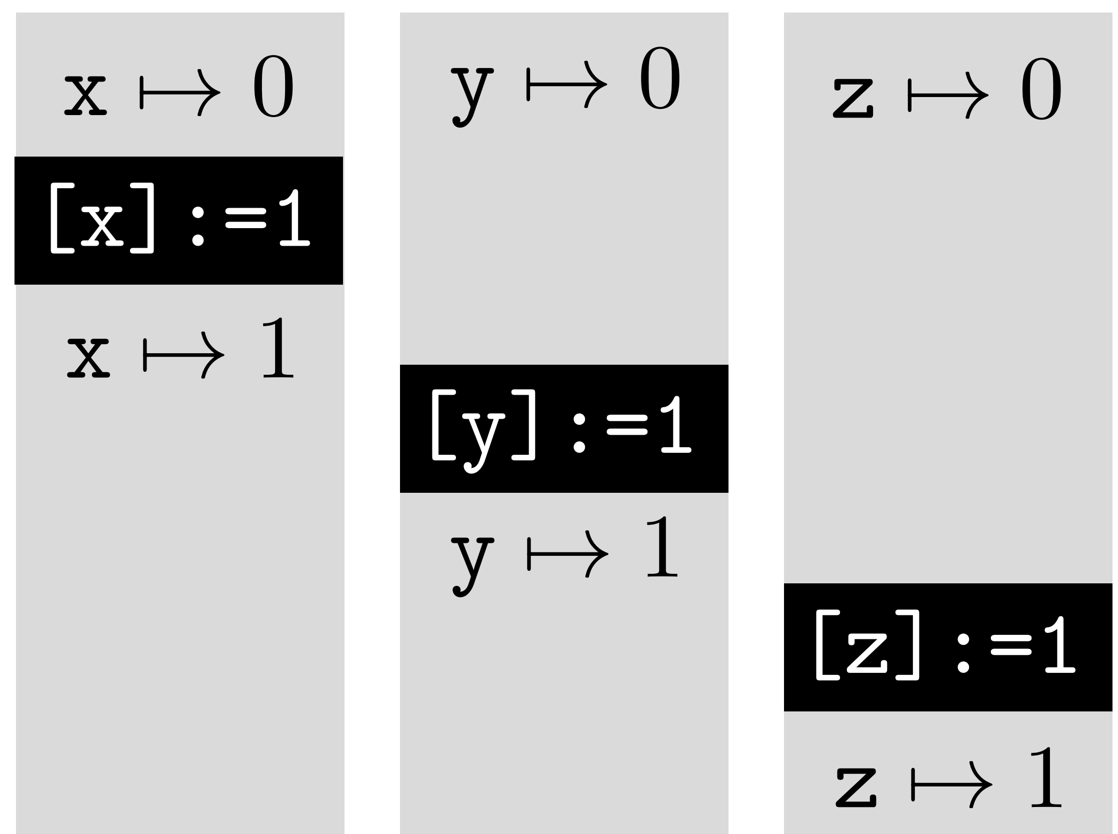


∃α
∃β
while (...) {
∃α
∃β
}

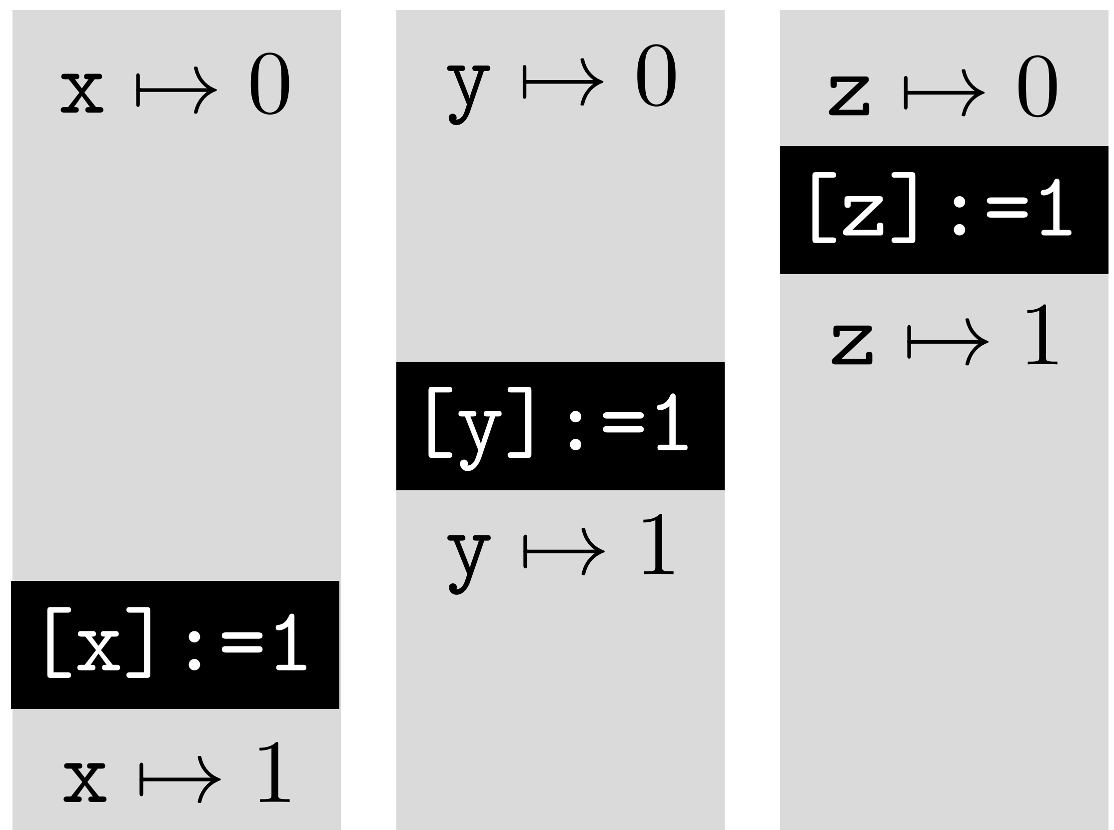


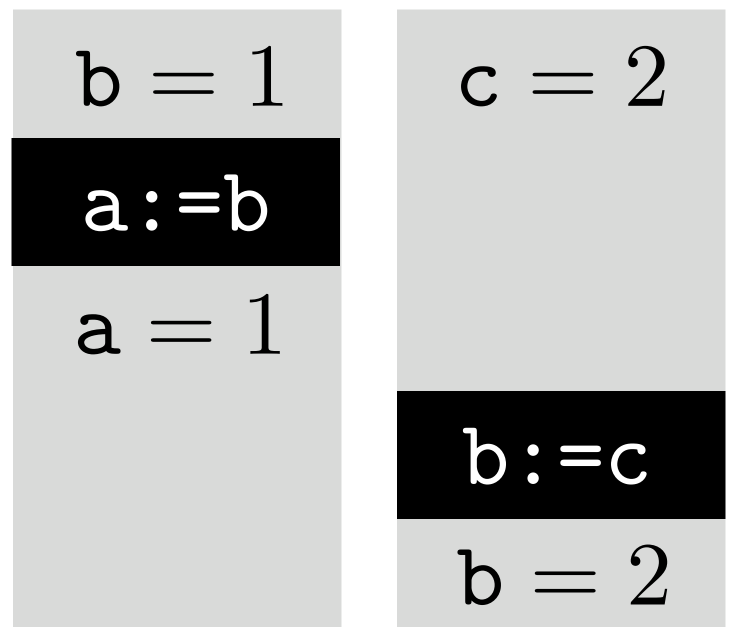
list α₀ x
y:=nil
list ϵ y
Choose α := α₀ and β := ϵ
∃α
∃β
list α x
list β y
α₀ ≐ β† · α
while (x!=nil) {
x ≠ nil
Unfold list def
∃α′, i, Z. x ↦ i, Z
∗ list α′ Z
∗ α ≐ i · α′
Choose α := α′
∃α
∃i
∃Z. x ↦ i, Z
∗ list α Z
α₀ ≐ β† · (i · α)
z:=[x+1]
list α z
x ↦ i, z
Reassociate i
[x+1]:=y
x ↦ i, y
α₀ ≐ (i·β)†·α
Fold list def
list (i · β) x
Choose β := (i · β)
∃β
list β x
α₀ ≐ β† · α
y:=x
list β y
x:=z
list α x
}
x ≐ nil
Unfold list def
α ≐ ϵ
Concatenate empty sequence
α₀ ≐ β†
Fold list def
list α₀† y

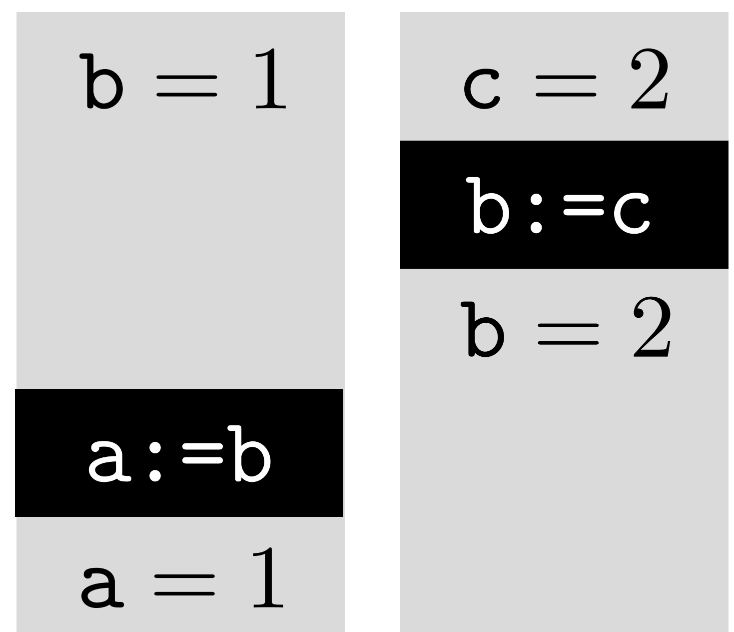
Dealing with program variables



x ↦ 0
[x]:=1
x ↦ 1
y ↦ 0
[y]:=1
y ↦ 1
z ↦ 0
[z]:=1
z ↦ 1

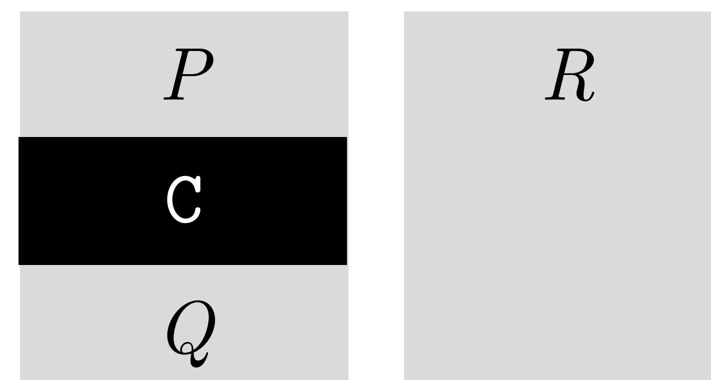

x ↦ 0
[x]:=1
x ↦ 1
y ↦ 0
[y]:=1
y ↦ 1
z ↦ 0
[z]:=1
z ↦ 1

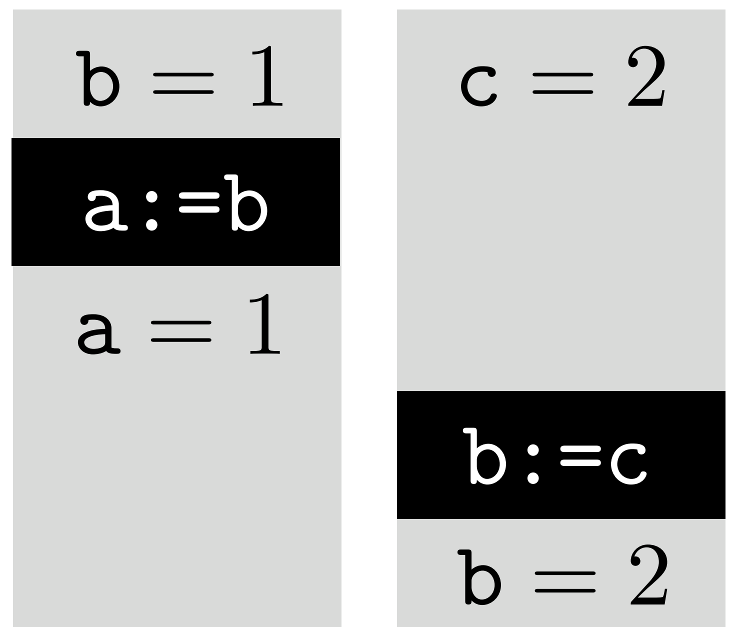

b = 1
a:=b
a = 1
c = 2
b:=c
b = 2

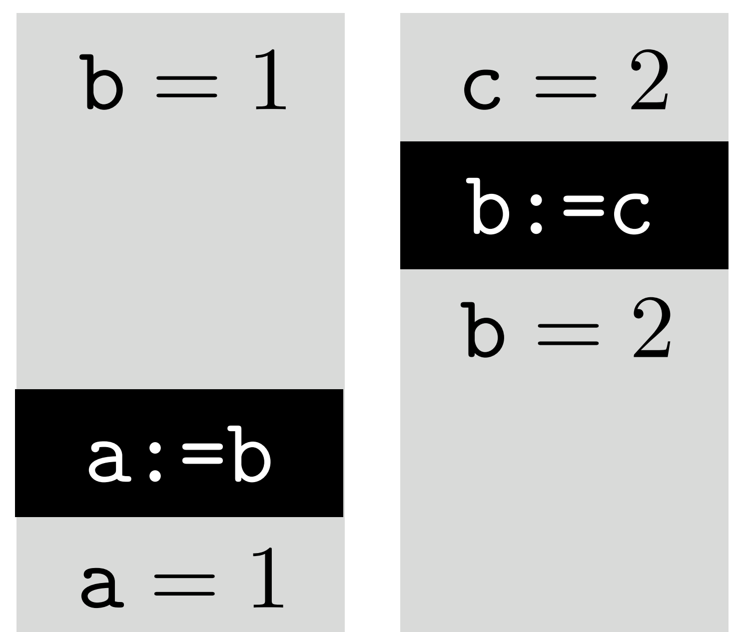

b = 1
a:=b
a = 1
c = 2
b:=c
b = 2

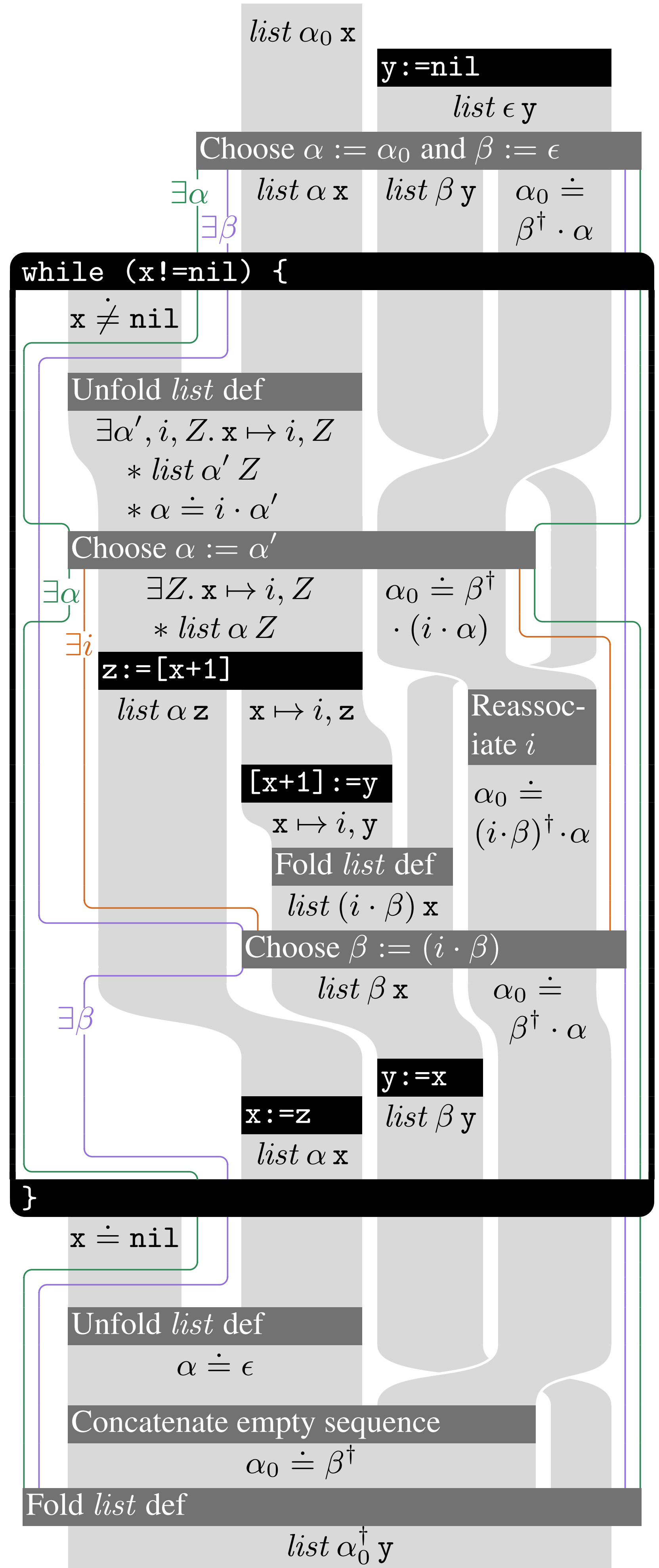
$$\frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{P * R\}\ \mathtt{C}\ \{Q * R\}}$$

providing $\mathit{fv}(R) \cap \mathit{modified}(\mathtt{C}) = \{\}$

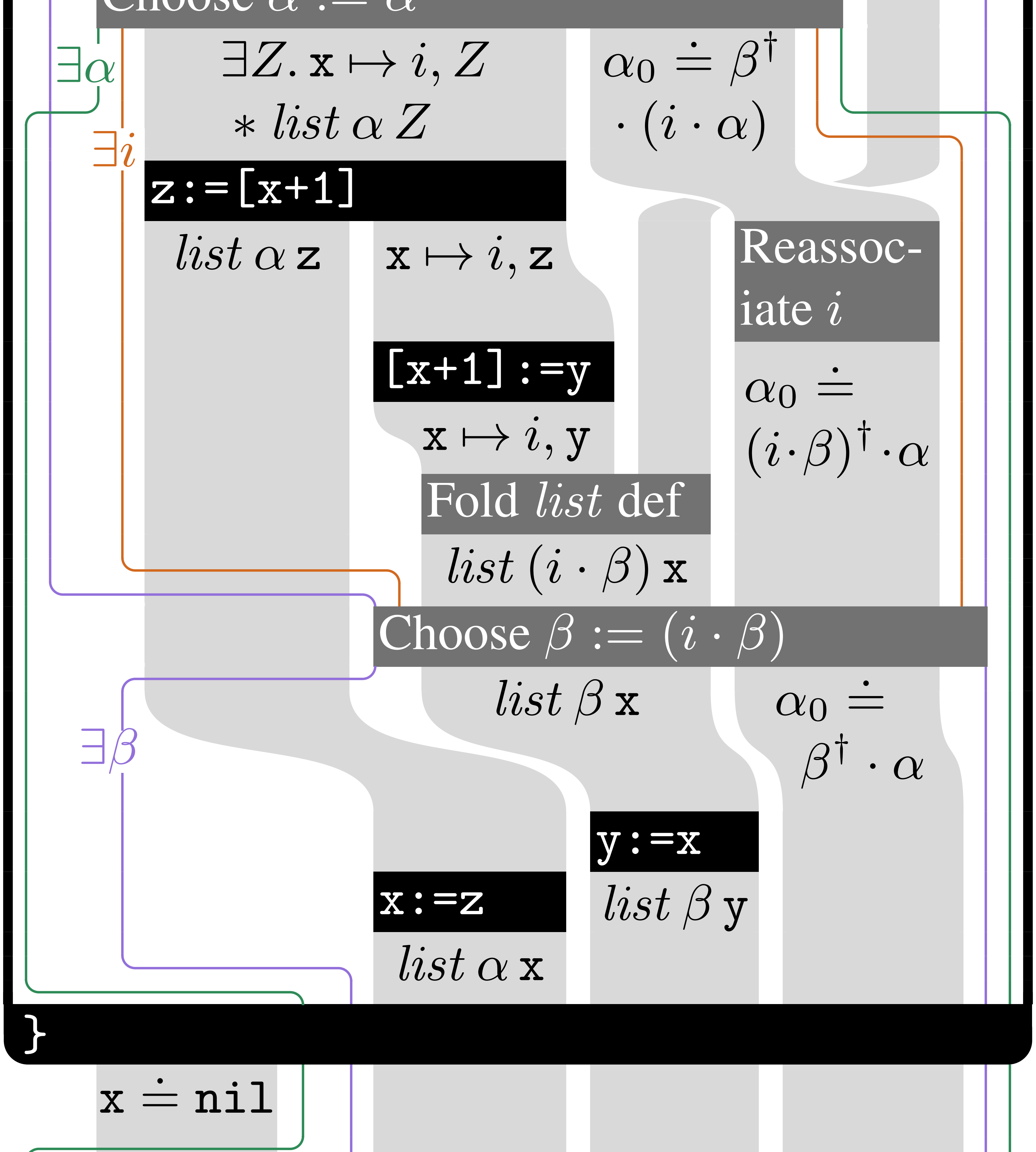


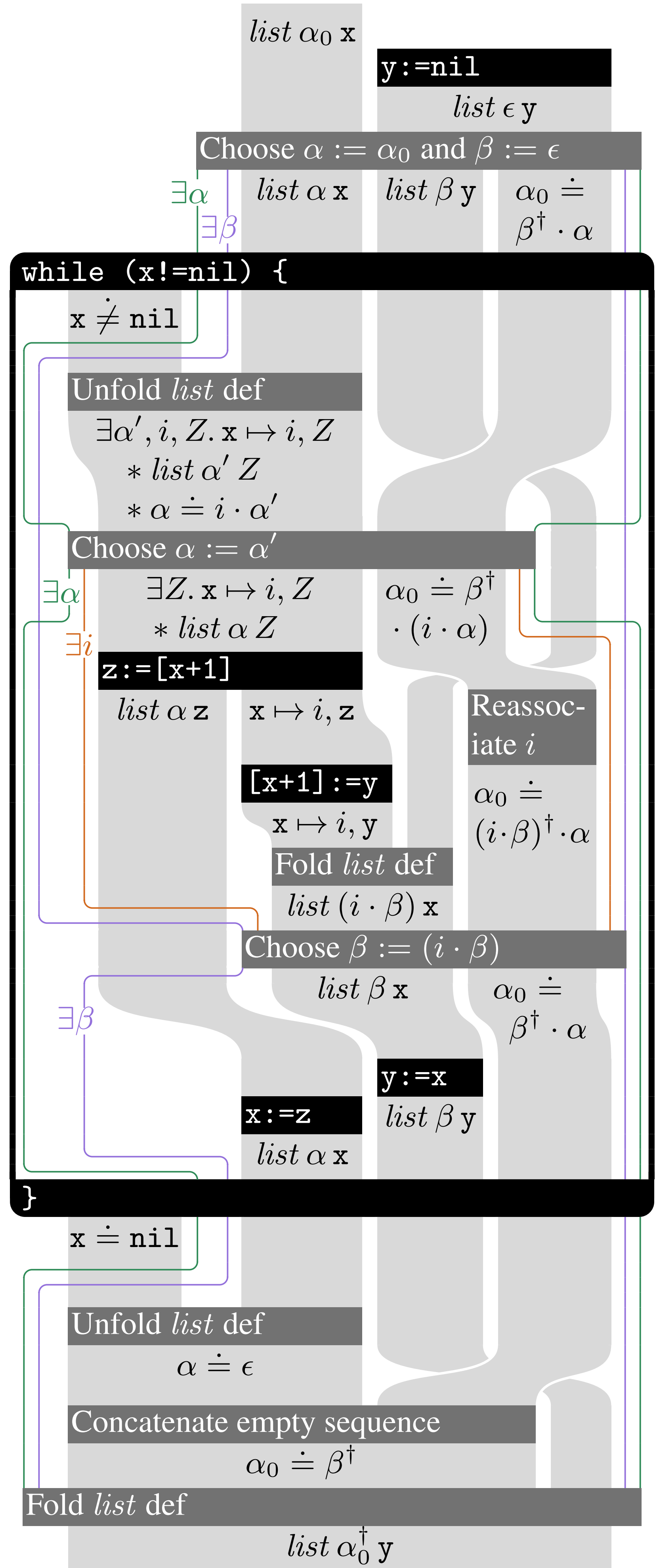

b = 1
a:=b
a = 1
c = 2
b:=c
b = 2


b = 1
a:=b
a = 1
c = 2
b:=c
b = 2



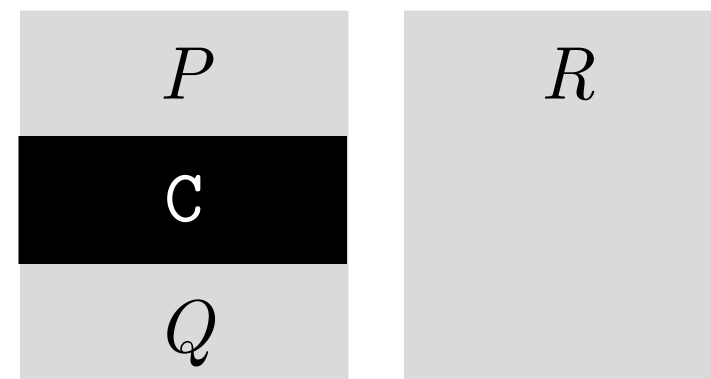
list α0 x
y:=nil
list ϵ y
Choose α := α0 and β := ϵ
∃α
∃β
list α x
list β y
α0 ≐ β† · α
while (x!=nil) {
x ≠ nil
Unfold list def
∃α′, i, Z. x ↦ i, Z
∗ list α′ Z
∗ α ≐ i · α′
Choose α := α′
∃α
∃i
∃Z. x ↦ i, Z
∗ list α Z
α0 ≐ β† · (i · α)
z:=[x+1]
list α z
x ↦ i, z
Reassociate i
[x+1]:=y
x ↦ i, y
α0 ≐ (i·β)†·α
Fold list def
list (i · β) x
Choose β := (i · β)
list β x
α0 ≐ β† · α
∃β
y:=x
list β y
x:=z
list α x
}
x ≐ nil
Unfold list def
α ≐ ϵ
Concatenate empty sequence
α0 ≐ β†
Fold list def
list α0† y


Choose $\alpha := \alpha$
$\exists\alpha$
$\exists i$
$\exists Z.\, \mathtt{x} \mapsto i, Z * \mathit{list}\, \alpha\, Z$
$\alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha)$
z:=[x+1]
$\mathit{list}\, \alpha\, \mathtt{z}$
$\mathtt{x} \mapsto i, \mathtt{z}$
Reassoc-iate i
[x+1]:=y
$\alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha$
$\mathtt{x} \mapsto i, \mathtt{y}$
Fold *list* def
$\mathit{list}\, (i \cdot \beta)\, \mathtt{x}$
Choose $\beta := (i \cdot \beta)$
$\exists\beta$
$\mathit{list}\, \beta\, \mathtt{x}$
$\alpha_0 \doteq \beta^\dagger \cdot \alpha$
y:=x
$\mathit{list}\, \beta\, \mathtt{y}$
x:=z
$\mathit{list}\, \alpha\, \mathtt{x}$
}
$\mathtt{x} \doteq \mathtt{nil}$


list α₀ x
y:=nil
list ϵ y
Choose α := α₀ and β := ϵ
∃α
∃β
list α x
list β y
α₀ ≐ β† · α
while (x!=nil) {
x ≠ nil
Unfold list def
∃α′, i, Z. x ↦ i, Z
∗ list α′ Z
∗ α ≐ i · α′
Choose α := α′
∃α
∃i
∃Z. x ↦ i, Z
∗ list α Z
α₀ ≐ β† · (i · α)
z:=[x+1]
list α z
x ↦ i, z
Reassoc-iate i
[x+1]:=y
x ↦ i, y
α₀ ≐ (i·β)†·α
Fold list def
list (i · β) x
Choose β := (i · β)
list β x
α₀ ≐ β† · α
∃β
y:=x
list β y
x:=z
list α x
}
x ≐ nil
Unfold list def
α ≐ ϵ
Concatenate empty sequence
α₀ ≐ β†
Fold list def
list α₀† y

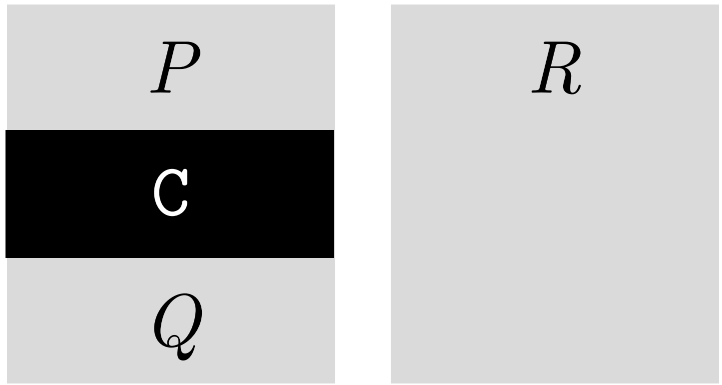
$$\frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{P * R\}\ \mathtt{C}\ \{Q * R\}}$$

providing $fv(R) \cap modified(\mathtt{C}) = \{\}$



$$\frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{P * R\}\ \mathtt{C}\ \{Q * R\}}$$

~~providing $fv(R) \cap modified(\mathtt{C}) = \{\}$~~



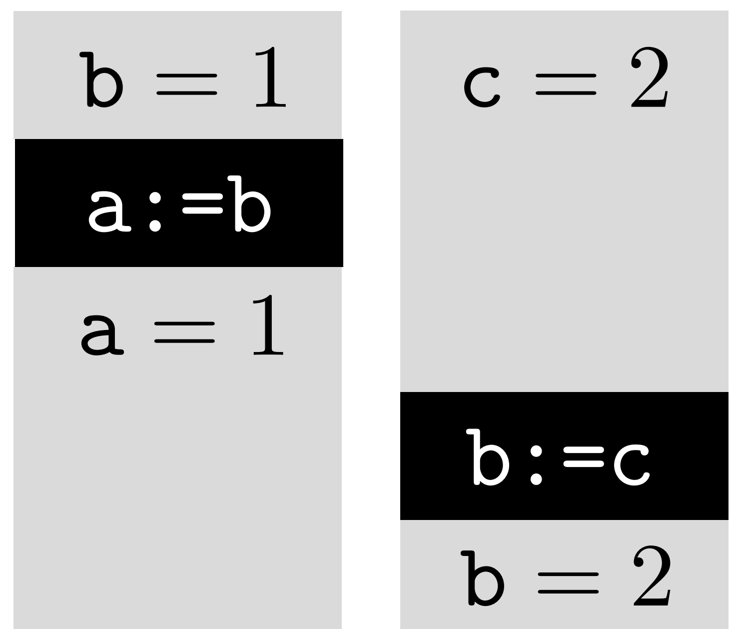
$$\frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{P * R\}\ \mathtt{C}\ \{Q * R\}}$$

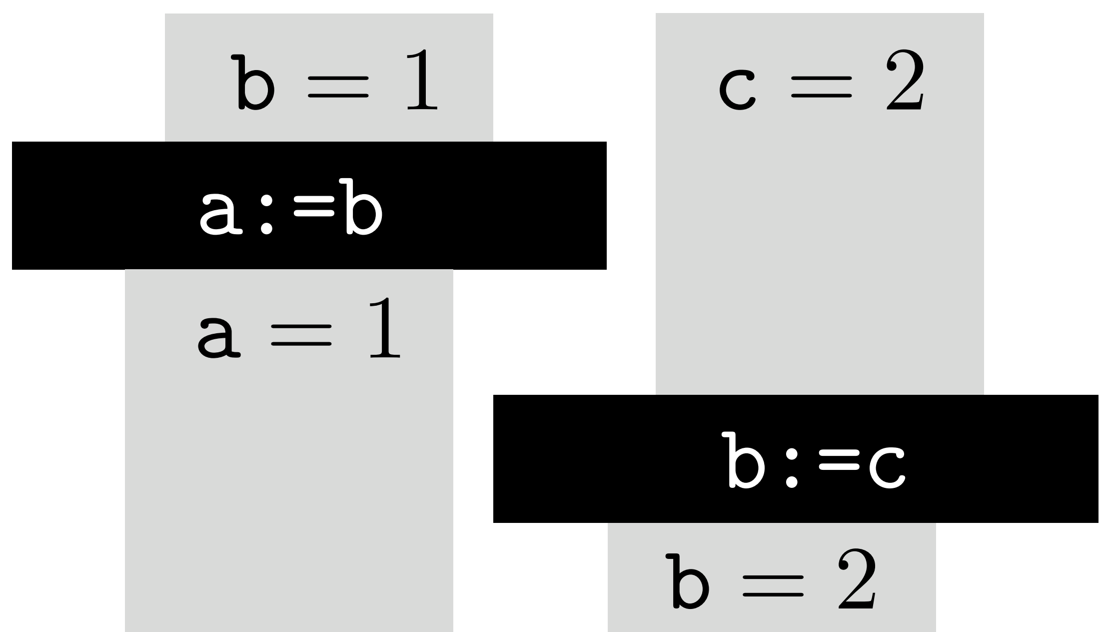
~~providing $fv(R) \cap modified(\mathtt{C}) = \{\}$~~

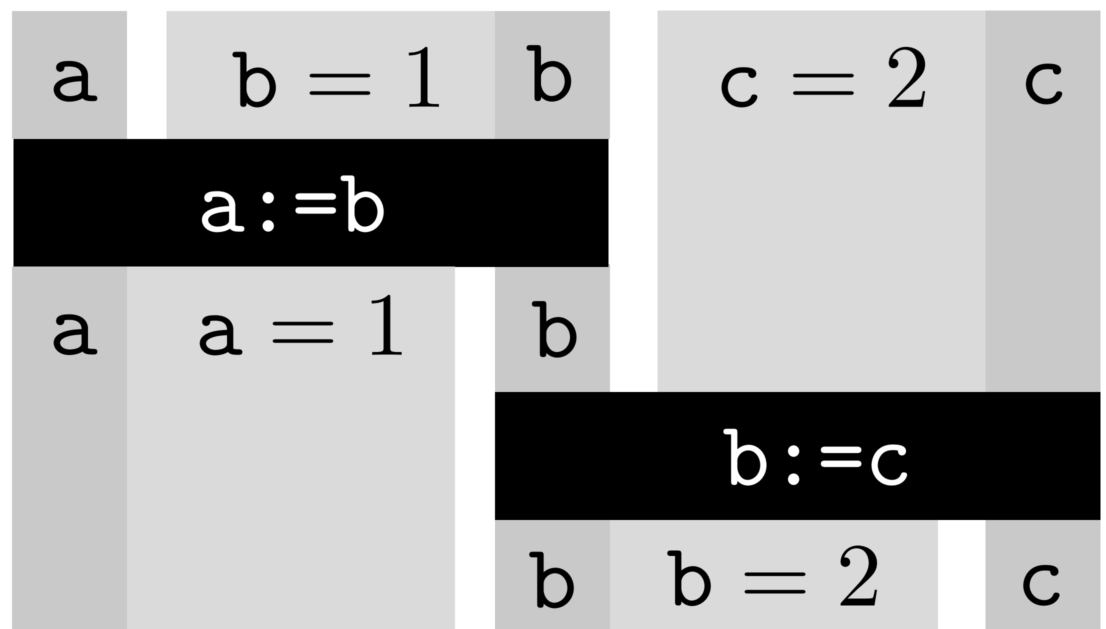
Electronic Notes in Theoretical Computer Science 155 (2006)

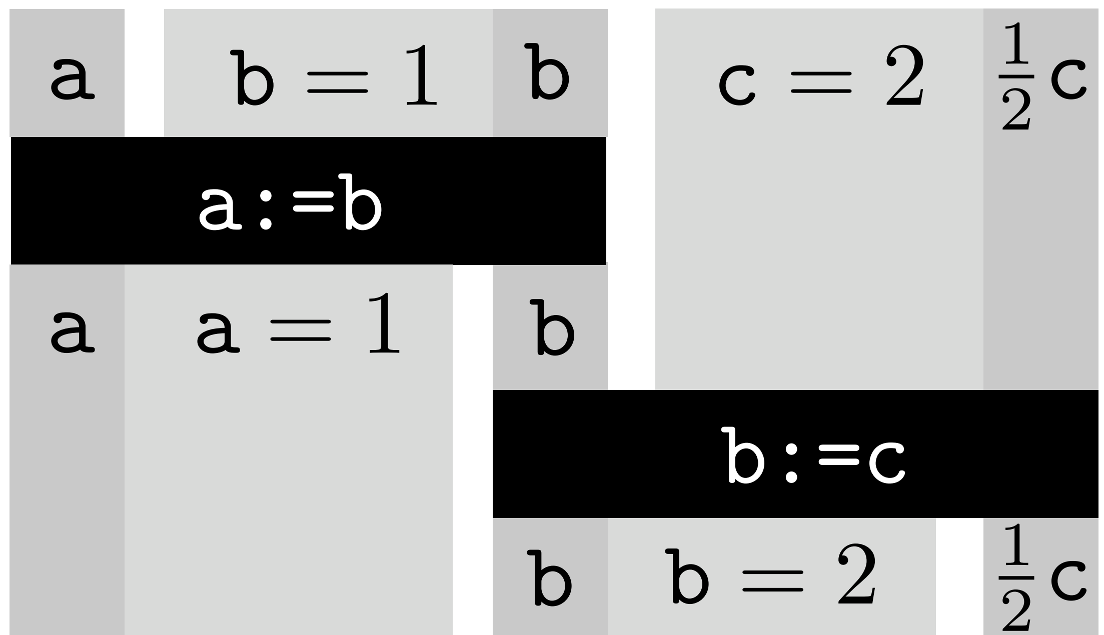
Variables as Resource in Separation Logic

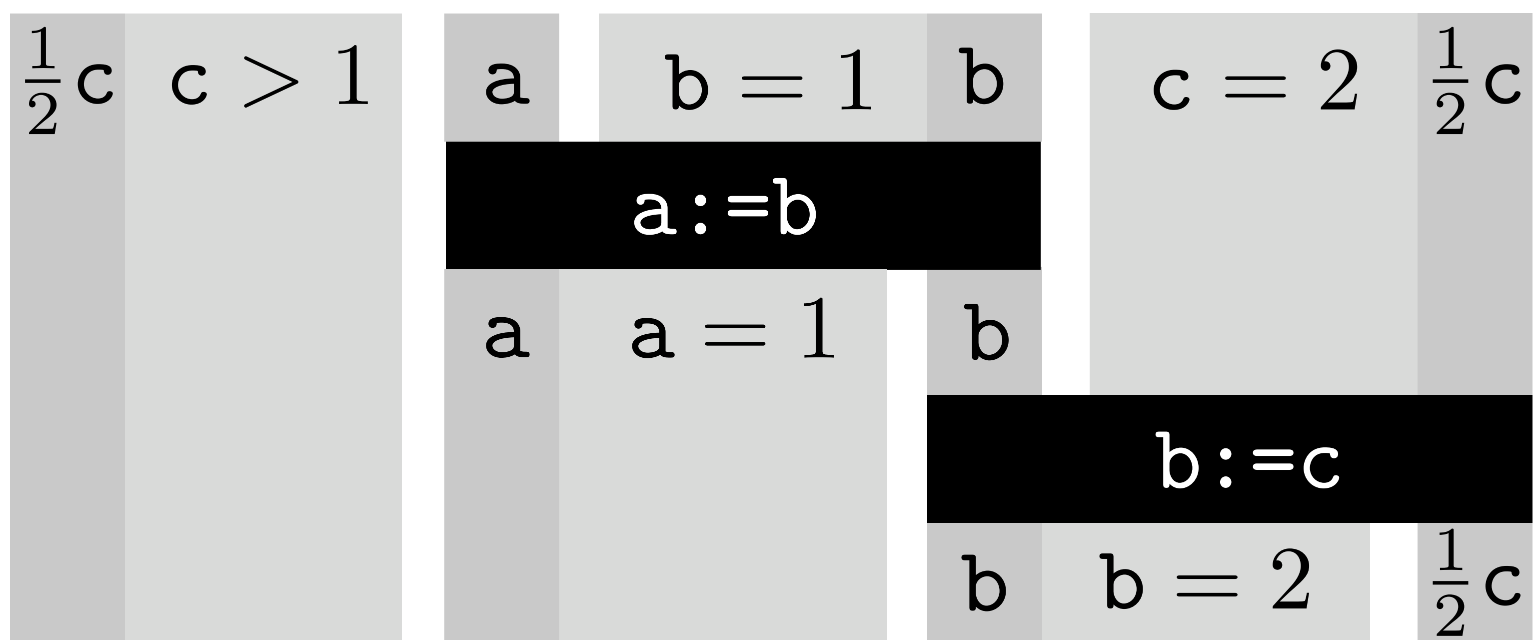
Richard Bornat Cristiano Calcagno Hongseok Yang

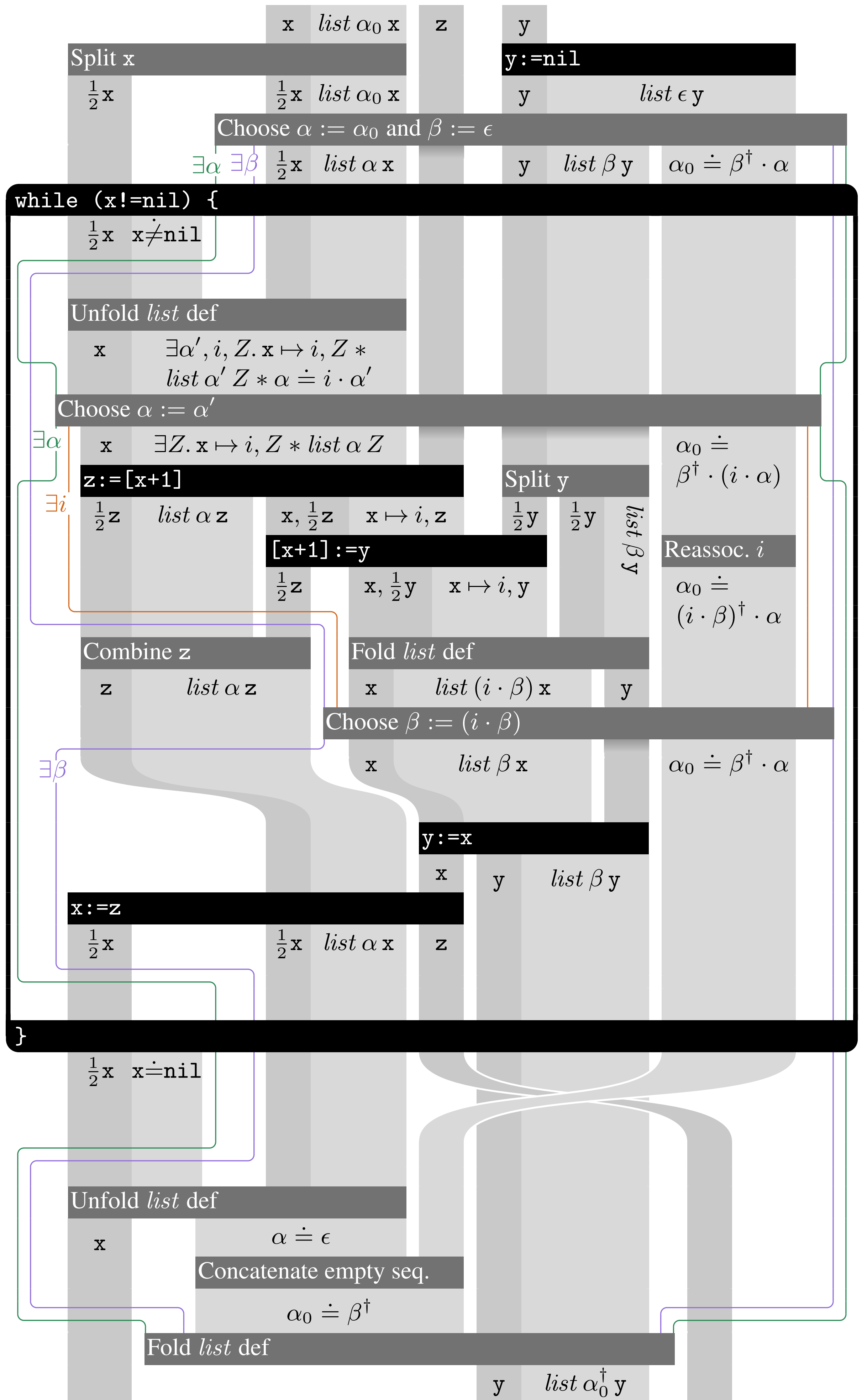

b = 1
a:=b
a = 1
c = 2
b:=c
b = 2


b = 1
c = 2
a:=b
a = 1
b:=c
b = 2

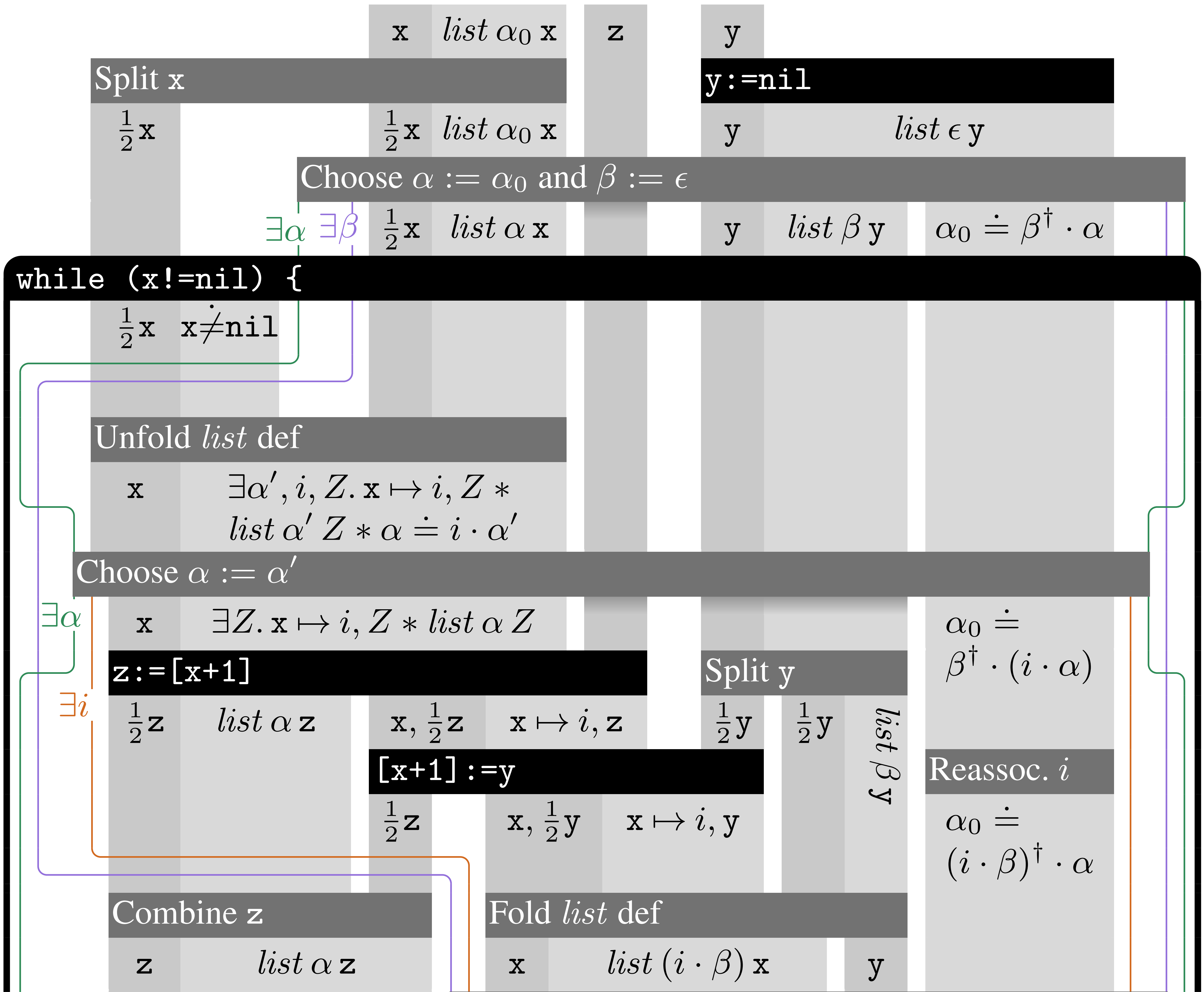

a
b = 1
b
c = 2
c
a:=b
a
a = 1
b
b:=c
b
b = 2
c

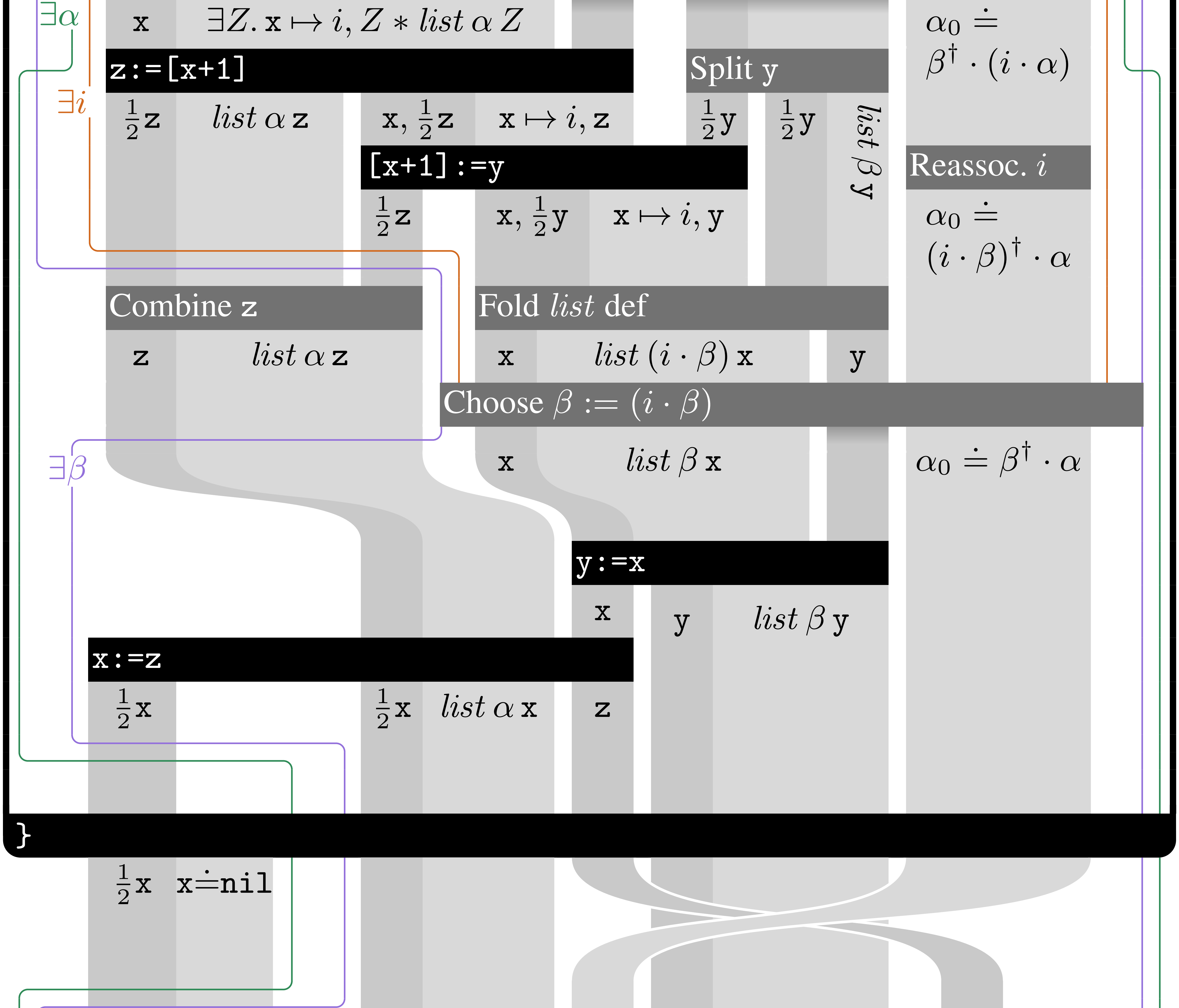

a
b = 1
b
c = 2
½ c
a:=b
a
a = 1
b
b:=c
b
b = 2
½ c

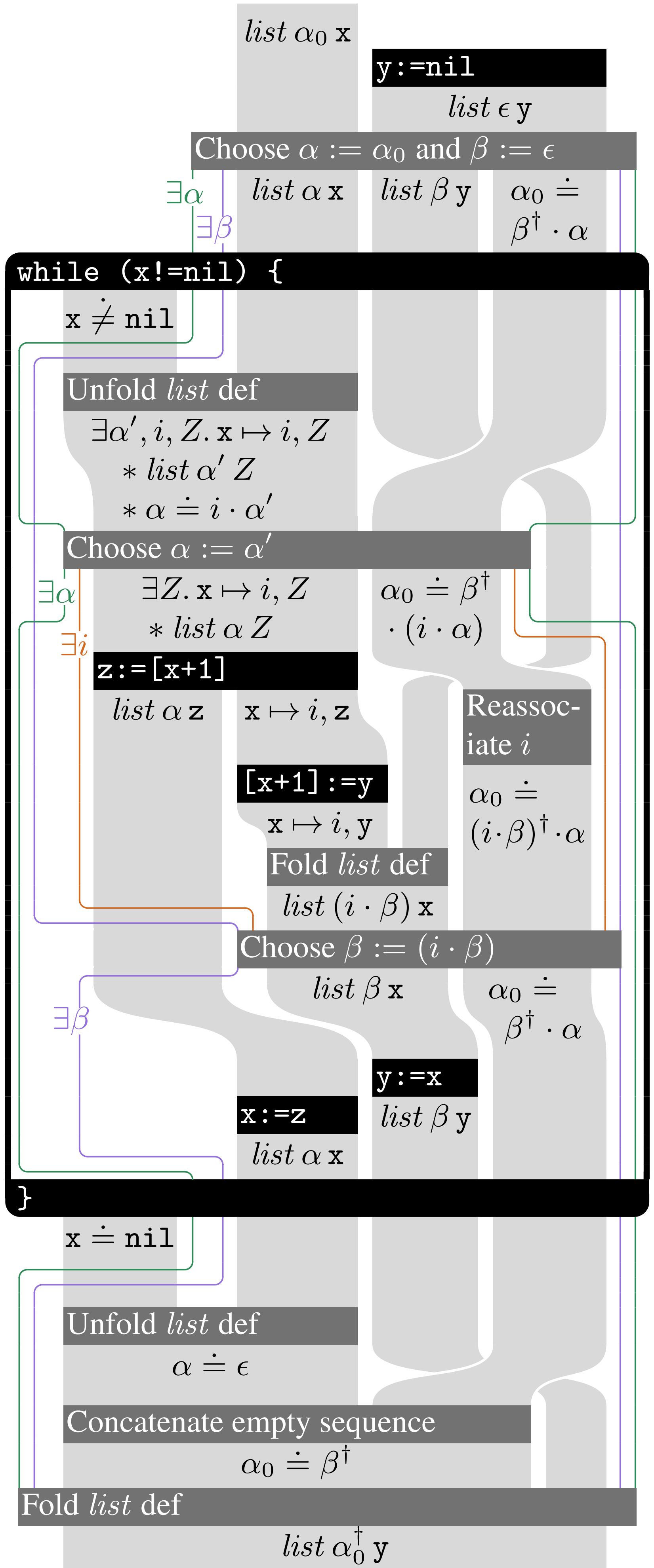
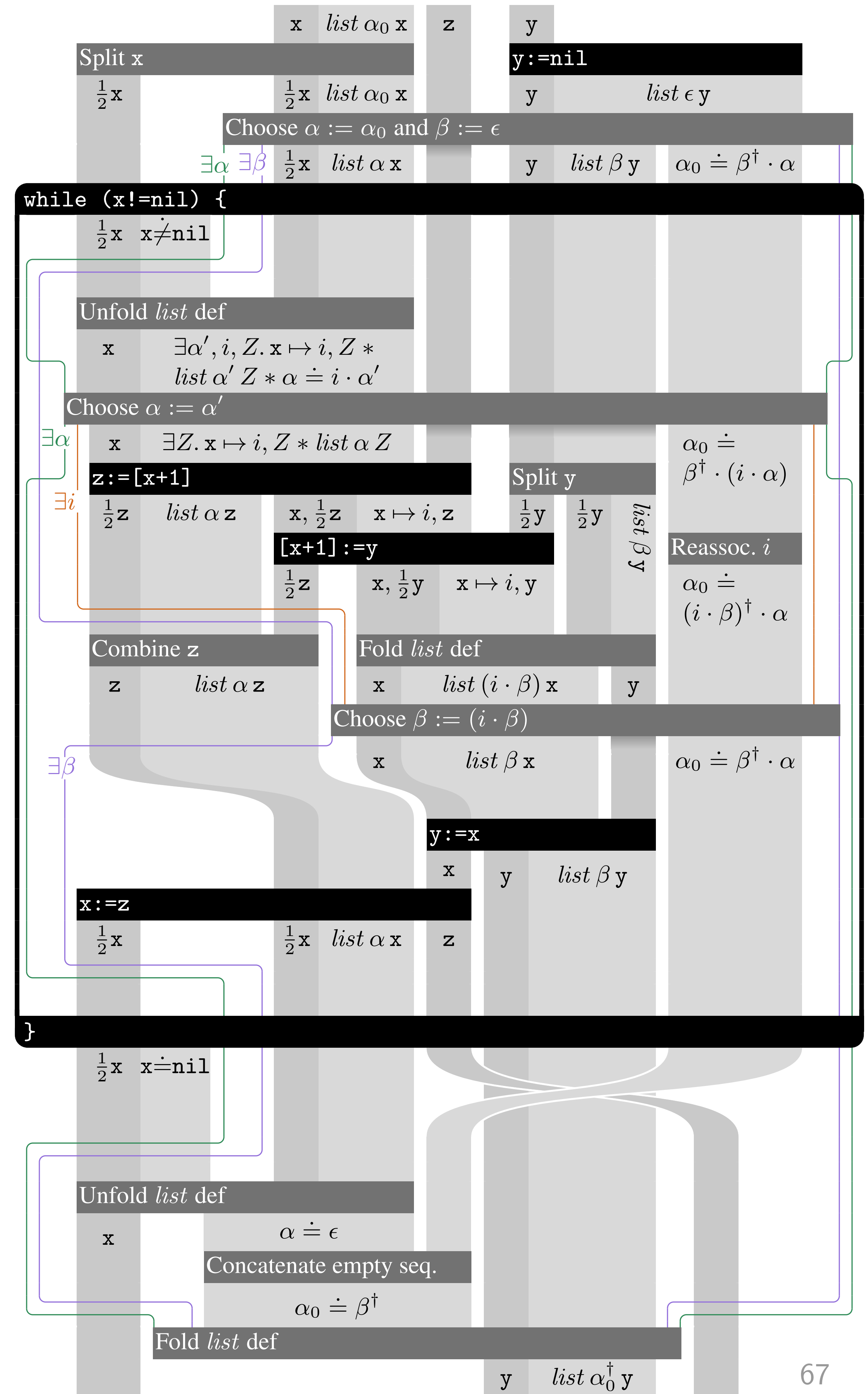

$\frac{1}{2}$c
c > 1
a
b $= 1$
b
c $= 2$
$\frac{1}{2}$c
a:=b
a
a $= 1$
b
b:=c
b
b $= 2$
$\frac{1}{2}$c



x
list α0 x
z
y
Split x
y:=nil
½x
½x list α0 x
y list ϵ y
Choose α := α0 and β := ϵ
∃α ∃β
½x list α x
y list β y
α0 ≐ β† · α
while (x!=nil) {
½x x≠nil
Unfold list def
x
∃α′, i, Z. x ↦ i, Z ∗ list α′ Z ∗ α ≐ i · α′
Choose α := α′
∃α
x ∃Z. x ↦ i, Z ∗ list α Z
α0 ≐ β† · (i · α)
z:=[x+1]
Split y
∃i
½z list α z
x, ½z x ↦ i, z
½y ½y
list β y
[x+1]:=y
Reassoc. i
½z
x, ½y x ↦ i, y
α0 ≐ (i · β)† · α
Combine z
Fold list def
z list α z
x list (i · β) x
y
Choose β := (i · β)
∃β
x list β x
α0 ≐ β† · α
y:=x
x
y list β y
x:=z
½x
½x list α x
z
}
½x x≐nil
Unfold list def
x
α ≐ ϵ
Concatenate empty seq.
α0 ≐ β†
Fold list def
y list α0† y


x
$list\ \alpha_0$ x
z
y
Split x
y:=nil
$\frac{1}{2}$x
$\frac{1}{2}$x
$list\ \alpha_0$ x
y
$list\ \epsilon$ y
Choose $\alpha := \alpha_0$ and $\beta := \epsilon$
$\exists\alpha$ $\exists\beta$
$\frac{1}{2}$x
$list\ \alpha$ x
y
$list\ \beta$ y
$\alpha_0 \doteq \beta^\dagger \cdot \alpha$
while (x!=nil) {
$\frac{1}{2}$x
x$\dot{\neq}$nil
Unfold $list$ def
x
$\exists\alpha', i, Z.$ x $\mapsto i, Z *$
$list\ \alpha'\ Z * \alpha \doteq i \cdot \alpha'$
Choose $\alpha := \alpha'$
$\exists\alpha$
x
$\exists Z.$ x $\mapsto i, Z * list\ \alpha\ Z$
$\alpha_0 \doteq$
$\beta^\dagger \cdot (i \cdot \alpha)$
z:=[x+1]
Split y
$\exists i$
$\frac{1}{2}$z
$list\ \alpha$ z
x, $\frac{1}{2}$z
x $\mapsto i$, z
$\frac{1}{2}$y
$\frac{1}{2}$y
$list\ \beta$ y
[x+1]:=y
Reassoc. i
$\frac{1}{2}$z
x, $\frac{1}{2}$y
x $\mapsto i$, y
$\alpha_0 \doteq$
$(i \cdot \beta)^\dagger \cdot \alpha$
Combine z
Fold $list$ def
z
$list\ \alpha$ z
x
$list\ (i \cdot \beta)$ x
y


∃α
x
∃Z. x ↦ i, Z ∗ list α Z
α0 ≐ β† · (i · α)
z:=[x+1]
Split y
∃i
½z
list α z
x, ½z
x ↦ i, z
½y
½y
list β y
[x+1]:=y
Reassoc. i
½z
x, ½y
x ↦ i, y
α0 ≐ (i · β)† · α
Combine z
Fold list def
z
list α z
x
list (i · β) x
y
Choose β := (i · β)
∃β
x
list β x
α0 ≐ β† · α
y:=x
x
y
list β y
x:=z
½x
½x
list α x
z
}
½x
x≐nil


list α₀ x
y:=nil
list ϵ y
Choose α := α₀ and β := ϵ
∃α
∃β
list α x
list β y
α₀ ≐ β† · α
while (x!=nil) {
x ≠ nil
Unfold list def
∃α′, i, Z. x ↦ i, Z
∗ list α′ Z
∗ α ≐ i · α′
Choose α := α′
∃α
∃i
∃Z. x ↦ i, Z
∗ list α Z
α₀ ≐ β†
· (i · α)
z:=[x+1]
list α z
x ↦ i, z
Reassoc-
iate i
[x+1]:=y
x ↦ i, y
α₀ ≐
(i·β)†·α
Fold list def
list (i · β) x
Choose β := (i · β)
∃β
list β x
α₀ ≐
β† · α
y:=x
x:=z
list β y
list α x
}
x ≐ nil
Unfold list def
α ≐ ϵ
Concatenate empty sequence
α₀ ≐ β†
Fold list def
list α₀† y

x
list α₀ x
z
y
Split x
y:=nil
½x
½x
list α₀ x
y
list ϵ y
Choose α := α₀ and β := ϵ
∃α
∃β
½x
list α x
y
list β y
α₀ ≐ β† · α
while (x!=nil) {
½x
x≠nil
Unfold list def
x
∃α′, i, Z. x ↦ i, Z ∗
list α′ Z ∗ α ≐ i · α′
Choose α := α′
∃α
x
∃Z. x ↦ i, Z ∗ list α Z
α₀ ≐
β† · (i · α)
z:=[x+1]
Split y
∃i
½z
list α z
x, ½z
x ↦ i, z
½y
½y
list β y
Reassoc. i
[x+1]:=y
½z
x, ½y
x ↦ i, y
α₀ ≐
(i · β)† · α
Combine z
Fold list def
z
list α z
x
list (i · β) x
y
Choose β := (i · β)
∃β
x
list β x
α₀ ≐ β† · α
y:=x
x
y
list β y
x:=z
½x
½x
list α x
z
}
½x
x≐nil
Unfold list def
x
α ≐ ϵ
Concatenate empty seq.
α₀ ≐ β†
Fold list def
y
list α₀† y

Future directions

Where now?

- [x] Define two-dimensional syntax of ribbon proofs, a formal semantics, and a collection of proof rules
- [ ] Graphical user interface for constructing and checking ribbon proofs
- [ ] Application to more exotic program logics
- [ ] Connections to bigraphs, string diagrams, proof nets

Ribbon proofs are...

- an alternative to **proof outlines**
- **readable**, **flexible**, and **attractive**
- applicable to **separation logic** (and descendants)
- less **repetitive** than proof outlines, so more **scalable**