Higher-order Actions in Deny-Guarantee Reasoning

Mike Dodds & Matthew Parkinson

overview

Temporal properties of interference are hard to reason about in rely-guarantee.

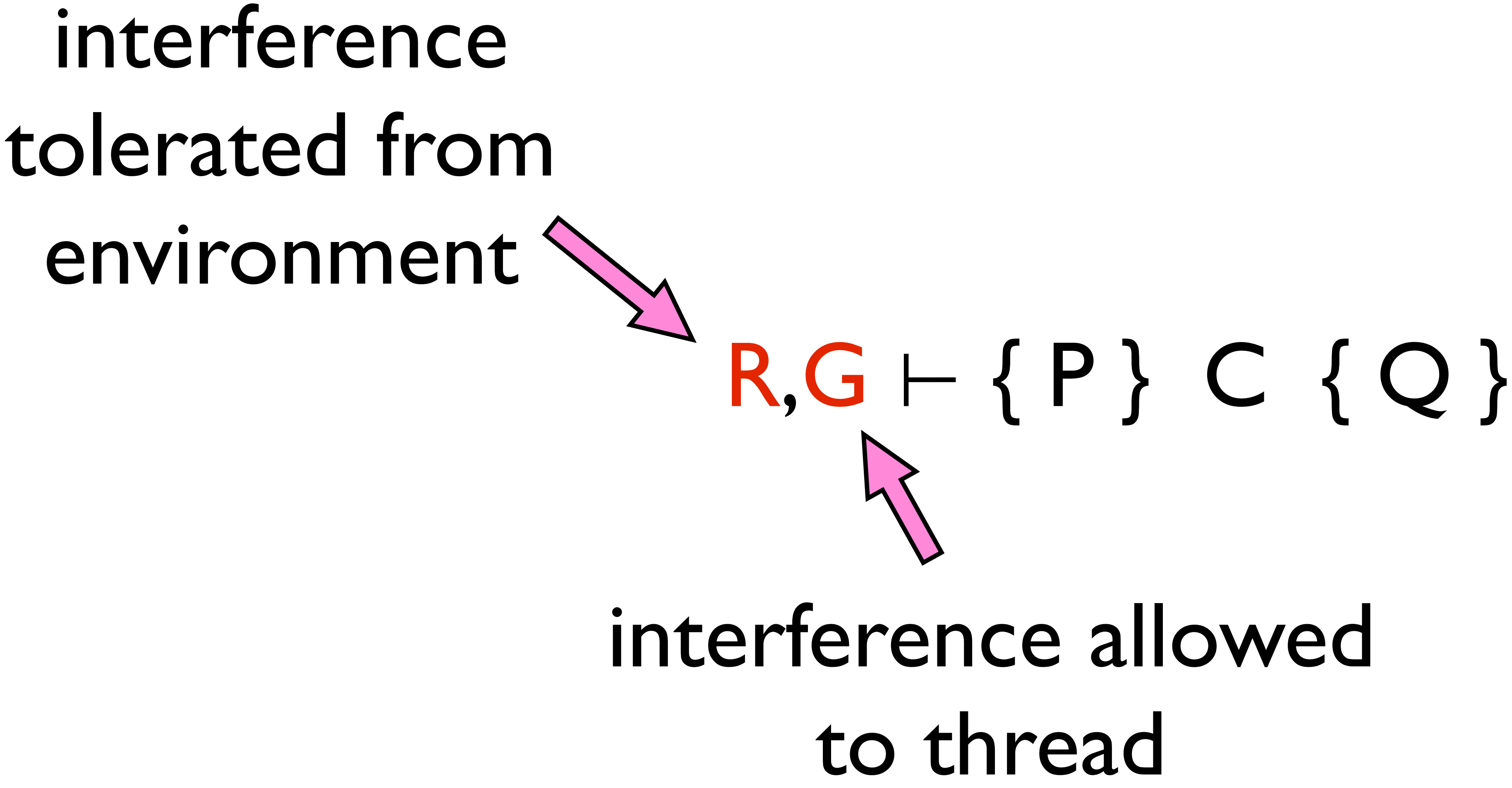
We define interference by splittable and joinable state.

Interference can rewrite interference, permitting or preventing future events.

Main examples: shared variable program, lock-coupling list.

Interference in rely-guarantee is modelled by two sets of *Actions*.

$$\text{Actions} \overset{\mathbf{def}}{=} \text{States} \times \text{States}$$



interference tolerated from environment

$$R, G \vdash \{\, P \,\}\ C\ \{\, Q \,\}$$

interference allowed to thread

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

```
read(x) = {
  t1 := x;
  t2 := x;
  if (t2 < t1) error;
}
```

program:

```
  { true }
incr(x,l) || incr(x,l) || read(x);
  { true }
```

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

After this point, the variable x cannot be incremented by any other thread.

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

at this point, only the current thread is allowed to write to x.

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

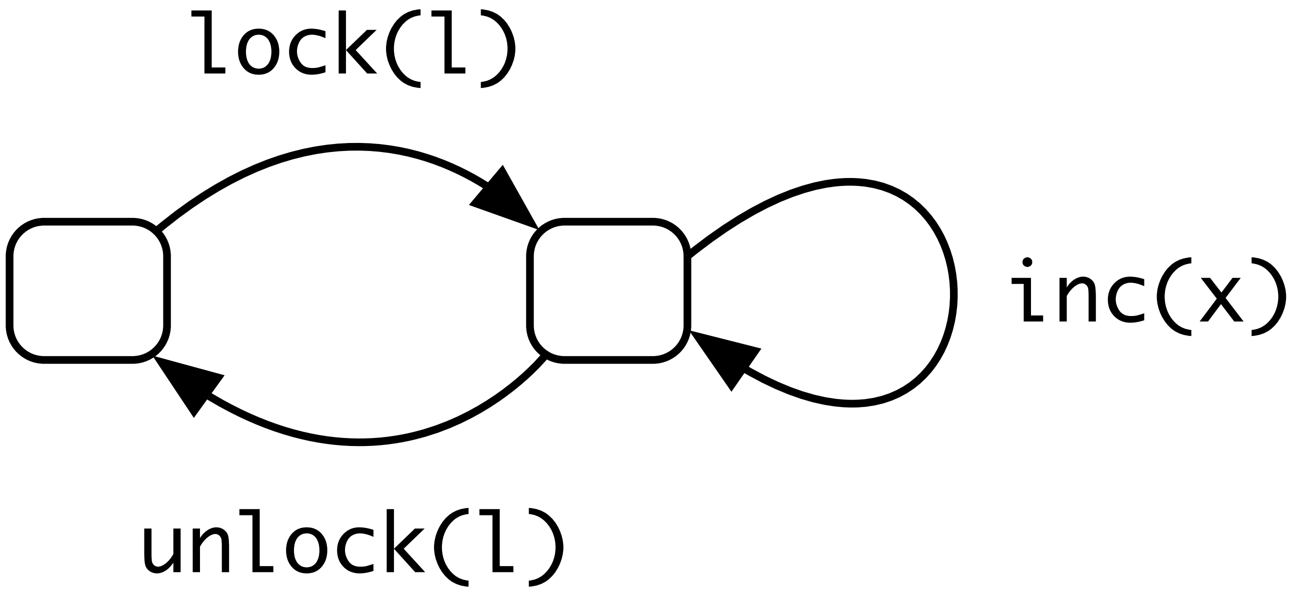
At this point x is released for other threads to increment.

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

Thread interference can’t be captured by a relation.

The fact that l is locked or not does not express whether x can be incremented.

interference is a state machine.



```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

```
read(x) = {
  t1 := x;
  t2 := x;
  if (t2 < t1) error;
}
```

Could move the variable x to protected local state...
(this is the RGSep solution)

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

```
read(x) = {
  t1 := x;
  t2 := x;
  if (t2 < t1) error;
}
```

Could move the variable x to protected local state...
(this is the RGSep solution)

No! The variable x needs to be readable by the `read(x)` thread

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

```
read(x) = {
  t1 := x;
  t2 := x;
  if (t2 < t1) error;
}
```

Could add an auxiliary variable to record which thread locked `l` ...

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

```
read(x) = {
  t1 := x;
  t2 := x;
  if (t2 < t1) error;
}
```

Could add an auxiliary variable to record which thread locked `l` ...

Ugly, doesn't really capture the semantics of the algorithm in the proof.

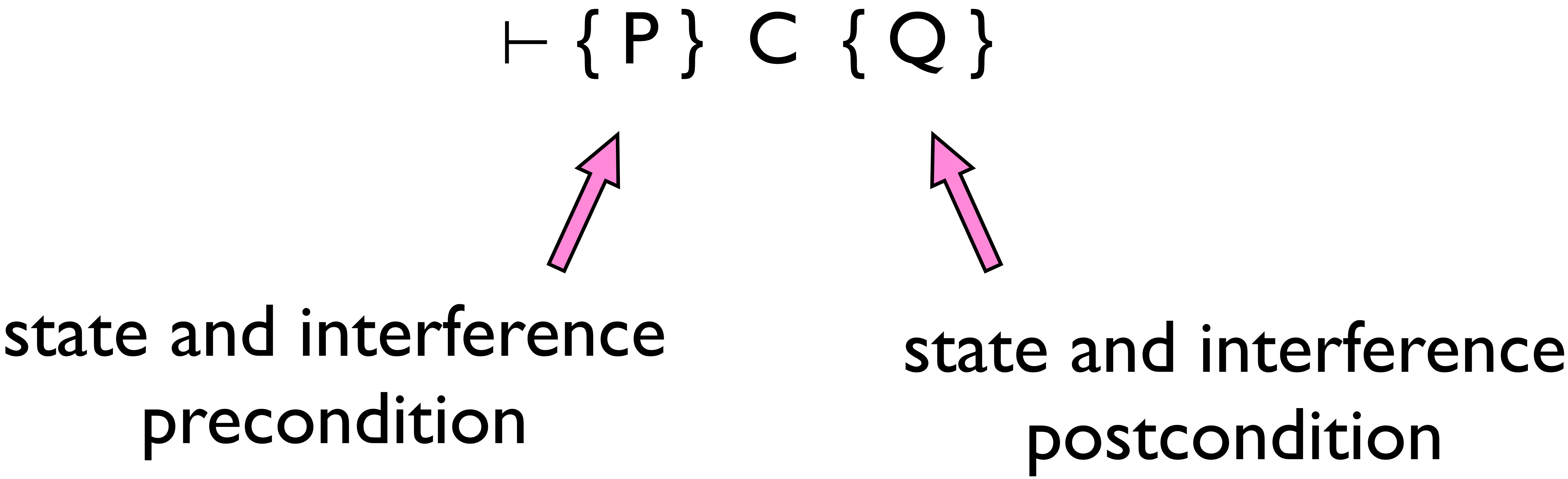
```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

```
read(x) = {
  t1 := x;
  t2 := x;
  if (t2 < t1) error;
}
```

What is really going on?

In deny-guarantee, interference is captured by *permissions*, which express both rely and guarantee.

Permissions are treated like normal state, so judgements are now of the form:



$$\vdash \{\, P \,\}\; C\; \{\, Q \,\}$$

state and interference precondition

state and interference postcondition

To perform an action, the thread must have sufficient permission.

Actions can be *denied*, meaning they cannot occur.

Just like state in RGSep, permissions can be shared or local.

- shared permissions cannot be used by any thread.
- local permissions can only be used by the owner thread.

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

Actions capture state update and permission update.

Need *lock*, *unlock* and *increase* actions.

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

Actions capture state update and permission update.

Need *lock*, *unlock* and *increase* actions.

$$\textsc{Inc}(x): \qquad x = n \wedge m > n \quad \rightsquigarrow \quad x = m$$

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

Actions capture state update and permission update.

Need *lock*, *unlock* and *increase* actions.

$$\textsc{Inc}(x): \quad x = n \wedge m > n \quad \rightsquigarrow \quad x = m$$

$$\textsc{Lock}(l): \quad l = 0 * [\textsc{Inc}(x)]_1 \quad \rightsquigarrow \quad l = 1$$

```
incr(x,l) = {
  n := *;
  lock(l);
  t := x;
  x := t + n;
  if (x != t+n) error;
  unlock(l);
}
```

Actions capture state update and permission update.

Need *lock*, *unlock* and *increase* actions.

$$\begin{array}{rlcl}
\textsc{Inc}(x): & x = n \wedge m > n & \rightsquigarrow & x = m \\
\textsc{Lock}(l): & l = 0 * [\textsc{Inc}(x)]_1 & \rightsquigarrow & l = 1 \\
\textsc{Unlock}(l): & l = 1 & \rightsquigarrow & l = 0 * [\textsc{Inc}(x)]_1
\end{array}$$

$$\left\{ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
lock(l);


t := x;


x := t + n;


if (x != t+n) error;


unlock(l);
```

$$\left\{ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
lock(l);

t := x;

x := t +

if (x != t+n) error;

unlock(l);
```

We write shared state as boxed and local state as unboxed

$$\left\{ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    lock(l);
```

$$\left\{ \boxed{l = 1} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    t := x;


    x := t + n;


    if (x != t+n) error;


    unlock(l);
```

$$\left\{ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    lock(l);
```

$$\left\{ \boxed{l = 1} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    t := x;
```

$$\left\{ \boxed{l = 1 * x = t} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    x := t + n;


    if (x != t+n) error;


    unlock(l);
```

$$\left\{ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    lock(l);
```

$$\left\{ \boxed{l = 1} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    t := x;
```

$$\left\{ \boxed{l = 1 * x = t} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    x := t + n;
```

$$\left\{ \boxed{l = 1 * x = (t + n)} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    if (x != t+n) error;


    unlock(l);
```

$$\left\{ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    lock(l);
```

$$\left\{ \boxed{l = 1} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    t := x;
```

$$\left\{ \boxed{l = 1 * x = t} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    x := t + n;
```

$$\left\{ \boxed{l = 1 * x = (t + n)} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    if (x != t+n) error;
```

$$\left\{ \boxed{l = 1 * x = (t + n)} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})} \right\}$$

```
    unlock(l);
```

$$\left\{\ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})}\right\}$$

```
    lock(l);
```

$$\left\{\ \boxed{l = 1} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})}\right\}$$

```
    t := x;
```

$$\left\{\ \boxed{l = 1 * x = t} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})}\right\}$$

```
    x := t + n;
```

$$\left\{\ \boxed{l = 1 * x = (t + n)} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})}\right\}$$

```
    if (x != t+n) error;
```

$$\left\{\ \boxed{l = 1 * x = (t + n)} * [\textsc{Inc}(x)]_1 * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})}\right\}$$

```
    unlock(l);
```

$$\left\{\ \boxed{[\textsc{Inc}(x)]_1} * [\textsc{Lock}(l)]_{(\mathbf{g},\frac{1}{2})} * [\textsc{Unlock}(l)]_{(\mathbf{g},\frac{1}{2})}\right\}$$

semantics of deny-guarantee

interference semantics

Actions are purely syntactic.

$$\mathsf{Actions} \overset{\text{def}}{=} \mathsf{Names} \times \mathsf{Locs}^*$$

Permission gives each action a level of permission

$$\mathsf{PermDG} \overset{\textbf{def}}{=} \mathsf{Actions} \rightarrow \mathsf{FractionDG}$$

interference semantics

Actions are purely syntactic.

$$\begin{aligned} \mathsf{Actions} &\overset{\mathrm{def}}{=} \mathsf{Names} \times \mathsf{Locs}^{*} \\ \mathsf{PermDG} &\overset{\mathbf{def}}{=} \mathsf{Actions} \rightarrow \mathsf{FractionDG} \end{aligned}$$

interference semantics

Actions are purely syntactic.

$$\begin{aligned} \mathsf{Actions} &\overset{\text{def}}{=} \mathsf{Names} \times \mathsf{Locs}^* \\ \mathsf{PermDG} &\overset{\mathbf{def}}{=} \mathsf{Actions} \rightarrow \mathsf{FractionDG} \end{aligned}$$

Level of permission recorded by FractionDG

$$\begin{aligned} \mathsf{FractionDG} \overset{\mathbf{def}}{=} & \quad \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ & \cup \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ & \cup \{0, 1\} \end{aligned}$$

interference semantics

$$\begin{aligned}\mathsf{FractionDG} \quad \overset{\mathbf{def}}{=} \quad & \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ \cup \;& \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ \cup \;& \{0, 1\}\end{aligned}$$

interference semantics

$$\begin{aligned}\mathsf{FractionDG} \quad \overset{\mathbf{def}}{=} \quad & \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ \cup \; & \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ \cup \; & \{0,1\}\end{aligned}$$

interference semantics

$$\begin{aligned}\text{FractionDG} \quad \overset{\textbf{def}}{=} \quad & \{(\text{deny}, k) \mid k \in (0,1)\} \\ \cup\ & \{(\text{guar}, k) \mid k \in (0,1)\} \\ \cup\ & \{0, 1\}\end{aligned}$$

Join elements of FractionDG by addition.

$$0 \oplus p = p \qquad\qquad 1 \oplus 0 = 1$$

$$(\text{guar}, k) \oplus (\text{guar}, k') = \begin{cases} (\text{guar}, k + k') & \text{if } k + k' < 1 \\ 1 & \text{if } k + k' = 1 \end{cases}$$

$$(\text{deny}, k) \oplus (\text{deny}, k') = \begin{cases} (\text{deny}, k + k') & \text{if } k + k' < 1 \\ 1 & \text{if } k + k' = 1 \end{cases}$$

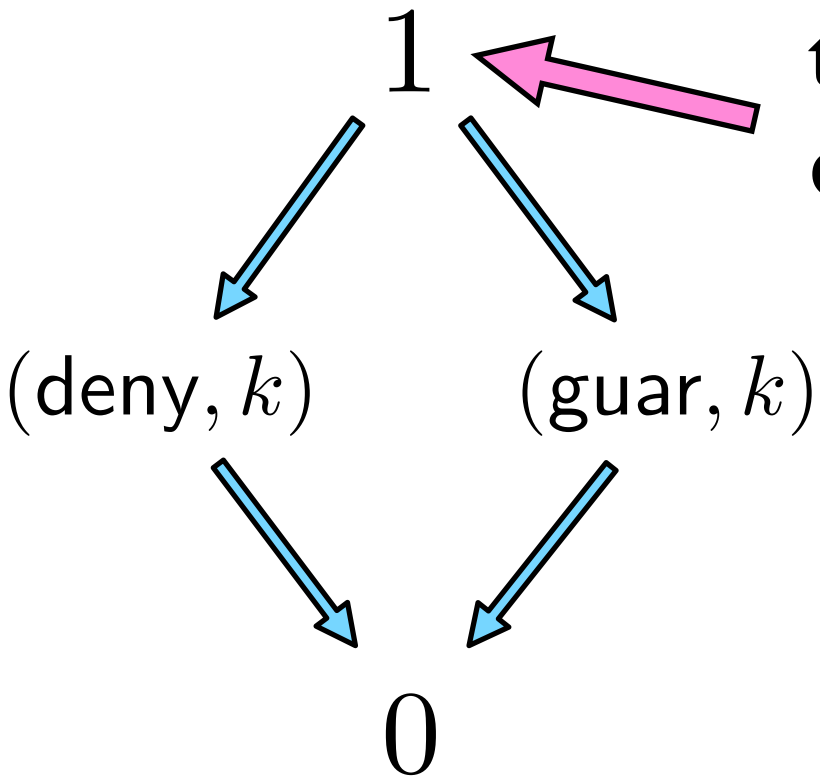
interference semantics

$$\begin{aligned}
\mathsf{FractionDG} \quad \overset{\mathbf{def}}{=} \quad & \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\
\cup \;& \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\
\cup \;& \{0, 1\}
\end{aligned}$$

$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
(\mathsf{deny}, k) \arrow[dr] & & (\mathsf{guar}, k) \arrow[dl] \\
 & 0 &
\end{tikzcd}$$

interference semantics

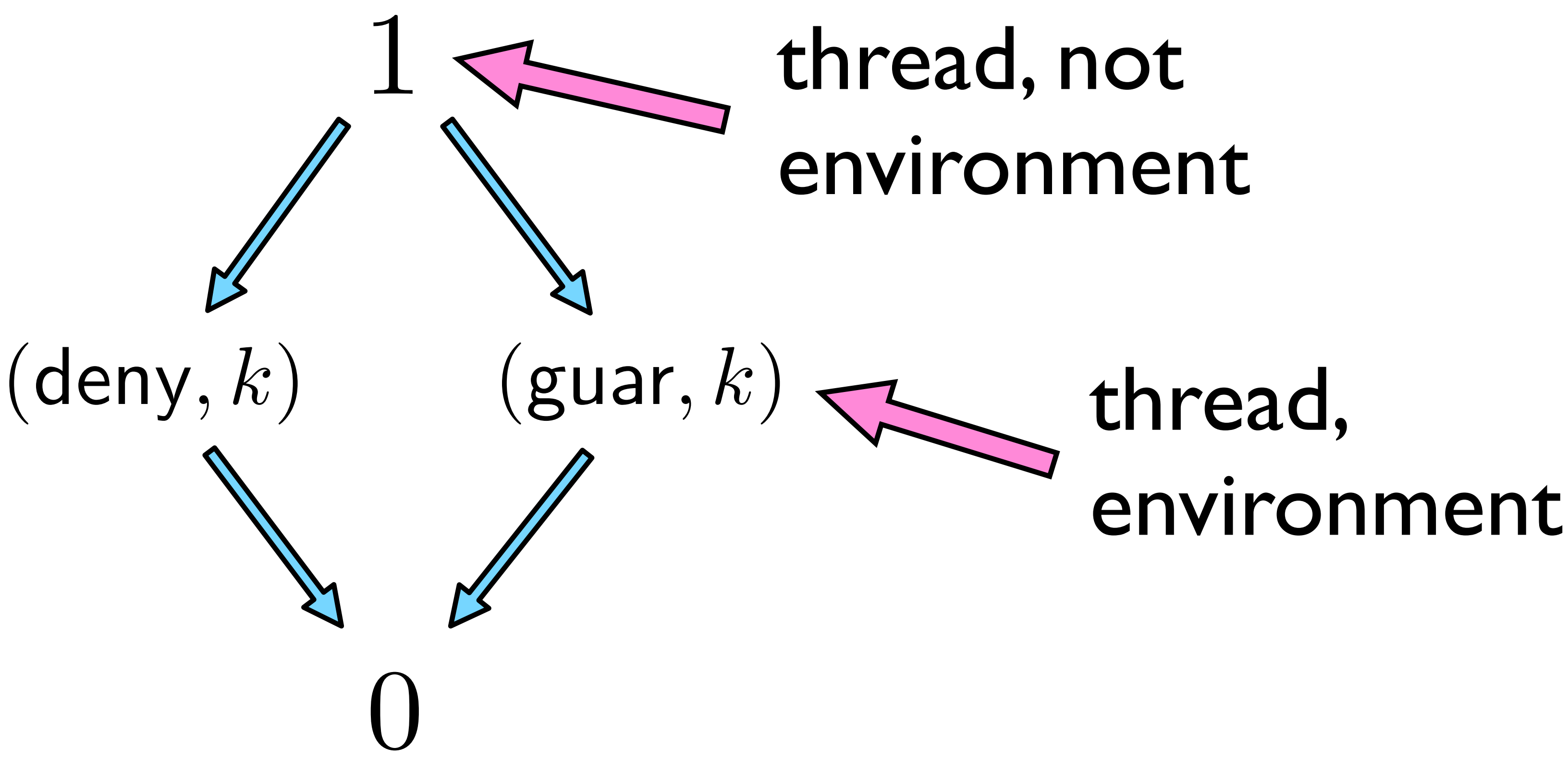
$$\begin{aligned} \mathsf{FractionDG} \overset{\mathbf{def}}{=} & \quad \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ & \cup \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ & \cup \{0, 1\} \end{aligned}$$



thread, not environment

interference semantics

$$\begin{aligned} \mathsf{FractionDG} \overset{\mathbf{def}}{=} & \quad \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ & \cup \ \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ & \cup \ \{0, 1\} \end{aligned}$$

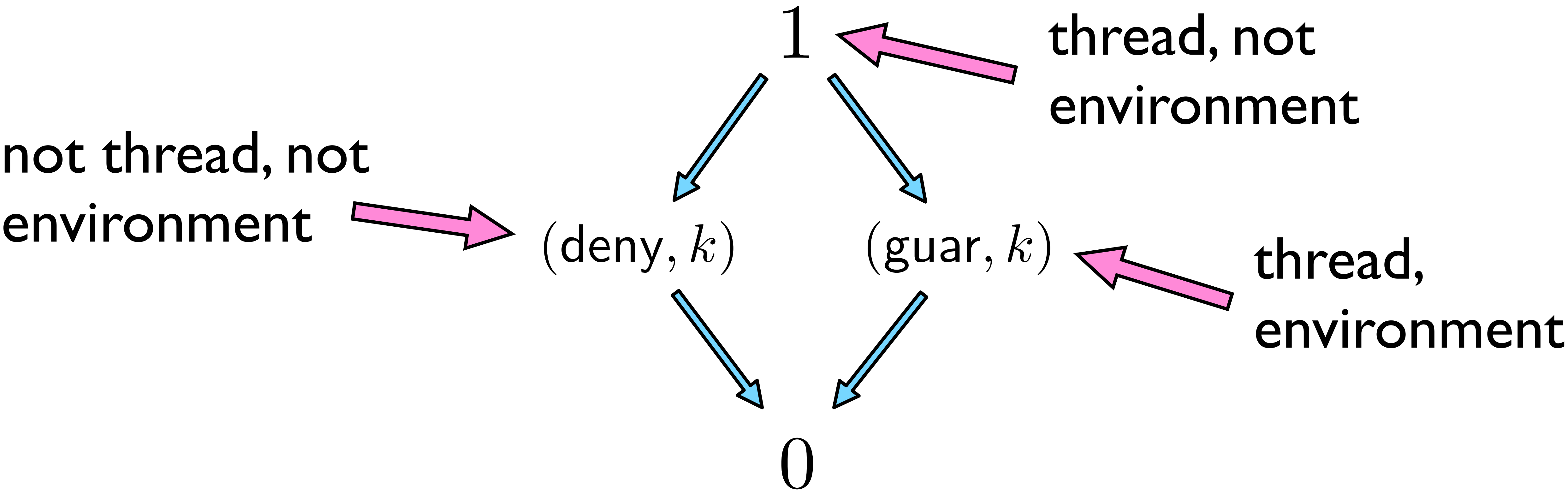


thread, not environment

thread, environment

interference semantics

$$\begin{aligned}\mathsf{FractionDG} \quad \overset{\mathbf{def}}{=} \quad & \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ \cup\ & \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ \cup\ & \{0, 1\}\end{aligned}$$



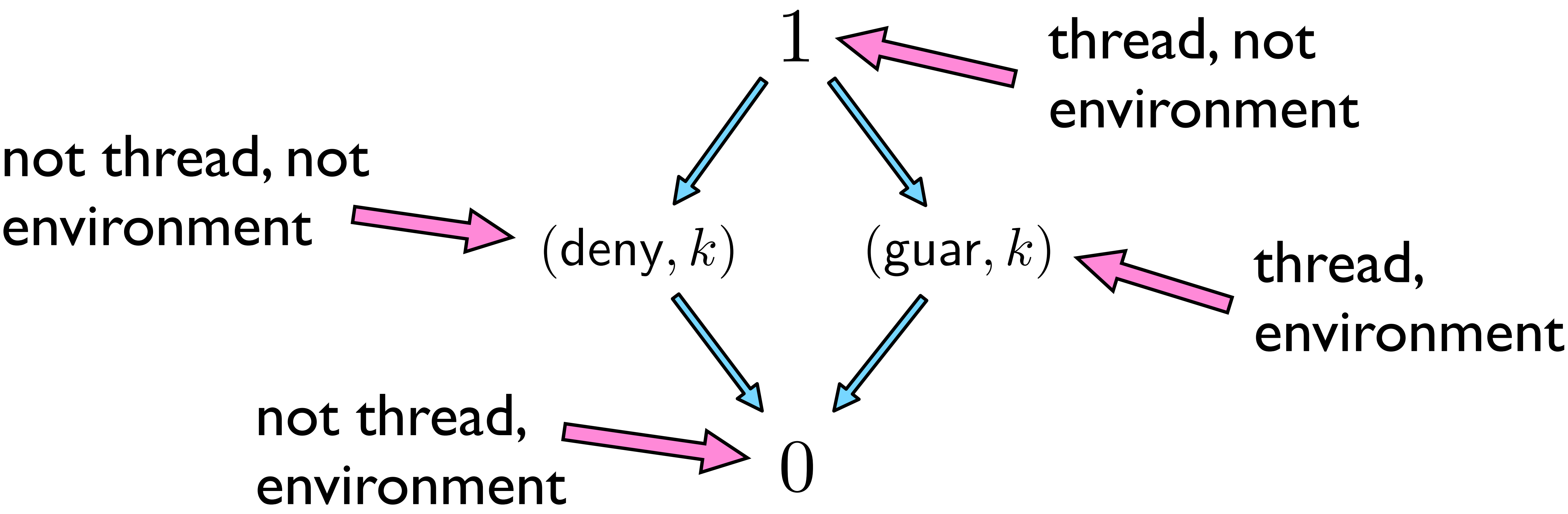
thread, not environment

not thread, not environment

thread, environment

interference semantics

$$\begin{aligned}\mathsf{FractionDG} \overset{\mathbf{def}}{=} &\quad \{(\mathsf{deny}, k) \mid k \in (0,1)\} \\ &\cup\ \{(\mathsf{guar}, k) \mid k \in (0,1)\} \\ &\cup\ \{0, 1\}\end{aligned}$$



1 ← thread, not environment

not thread, not environment → (deny, k)

(guar, k) ← thread, environment

not thread, environment → 0

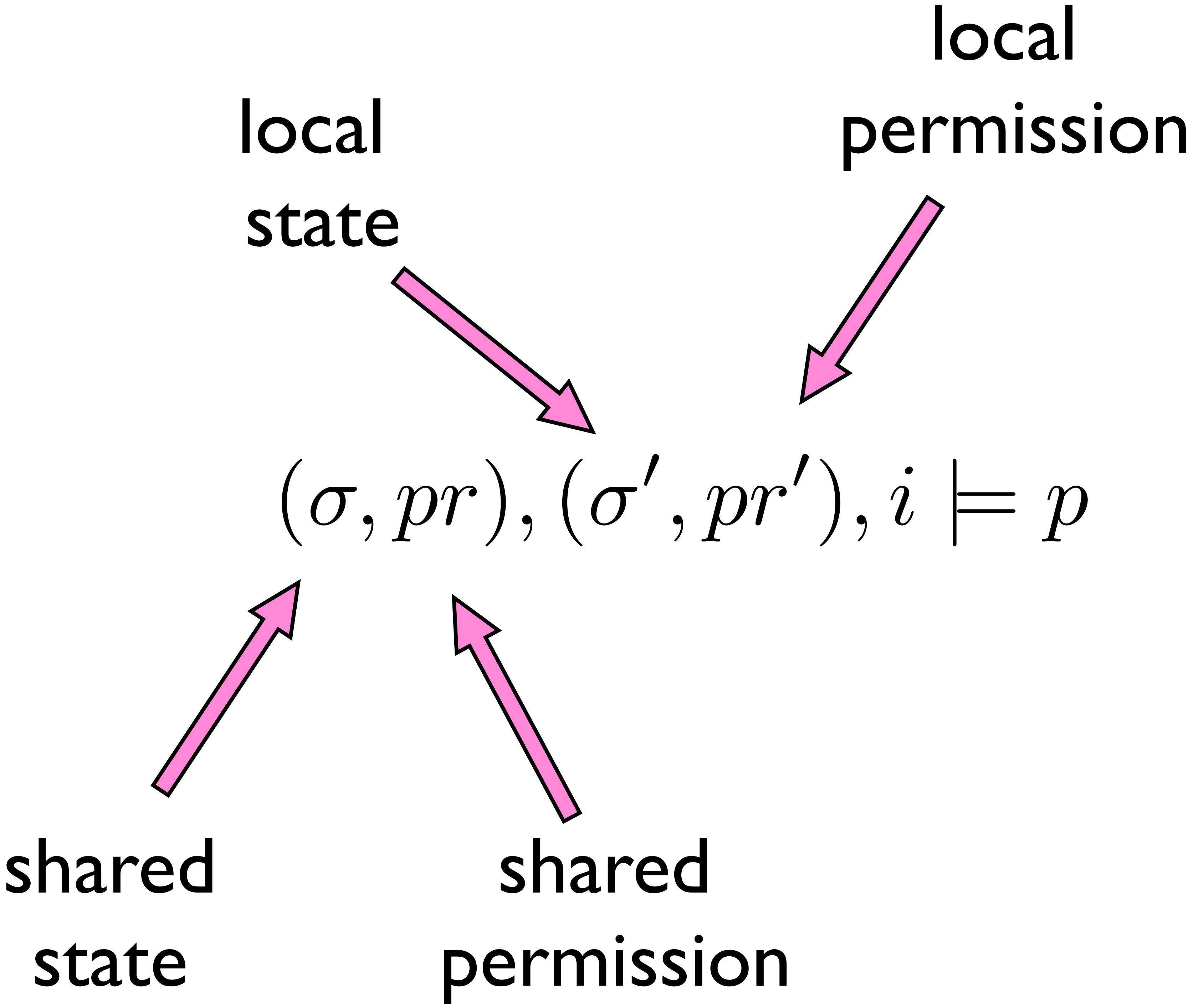
interference semantics

The semantics of an action is defined by an action environment.

$$\begin{aligned} \mathsf{Worlds} &\stackrel{\mathrm{def}}{=} \mathsf{States} \times \mathsf{PermDG} \\ \mathsf{Envs} &\stackrel{\mathrm{def}}{=} \mathsf{Actions} \rightarrow \mathsf{Worlds} \times \mathsf{Worlds} \end{aligned}$$

Note that the indirection through syntax here avoids a recursive domain equation.

Assertions



local
state

local
permission

$$(\sigma, pr), (\sigma', pr'), i \models p$$

shared
state

shared
permission

Stability

$$G_{pr,\eta} \overset{\mathbf{def}}{=} \{a \mid \exists \gamma, \vec{x}.\, \eta(\gamma, \vec{x}) = a \wedge pr(\gamma, \vec{x}) \in \{(\mathsf{guar}, k), 1\}\}$$

$$R_{pr,\eta} \overset{\mathbf{def}}{=} \{a \mid \exists \gamma, \vec{x}.\, \eta(\gamma, \vec{x}) = a \wedge pr(\gamma, \vec{x}) \in \{(\mathsf{guar}, k), 0\}\}$$

$$\begin{aligned} \mathsf{stable}(p, \eta) \quad \overset{\text{def}}{\iff} \quad & (\sigma, pr), (\sigma', pr'), i \models p \\ & \wedge ((\sigma, pr), (\sigma'', pr'')) \in R_{(pr \oplus pr'), \eta} \\ & \quad \implies (\sigma'', pr''), (\sigma', pr'), i \models p \end{aligned}$$

bigger example:

the lock-coupling list

Algorithm overview

A fine-grained algorithm for adding and removing elements from a list.

Traverse a list by hand-over-hand locking.

Threads cannot overtake other threads locking nodes.

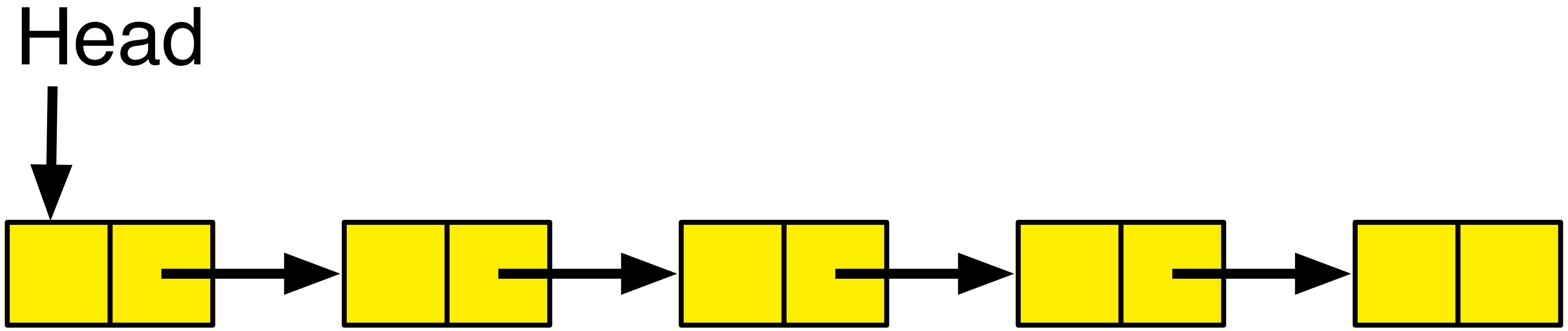
```
locate(e) {
  local p, c;
  p := hd;
  lock(p);
  c := p.next;
  while (c != nil) {
    lock(c);
    if (c.value = e)
      return (p,c);
    unlock(p);
    p := c;
    c := p.next;
  }
  return(p, c);
}
```

```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y != nil) {
    z := y.next;
    x.next := z;
    dispose(y);
  }
  unlock(x);
}
```

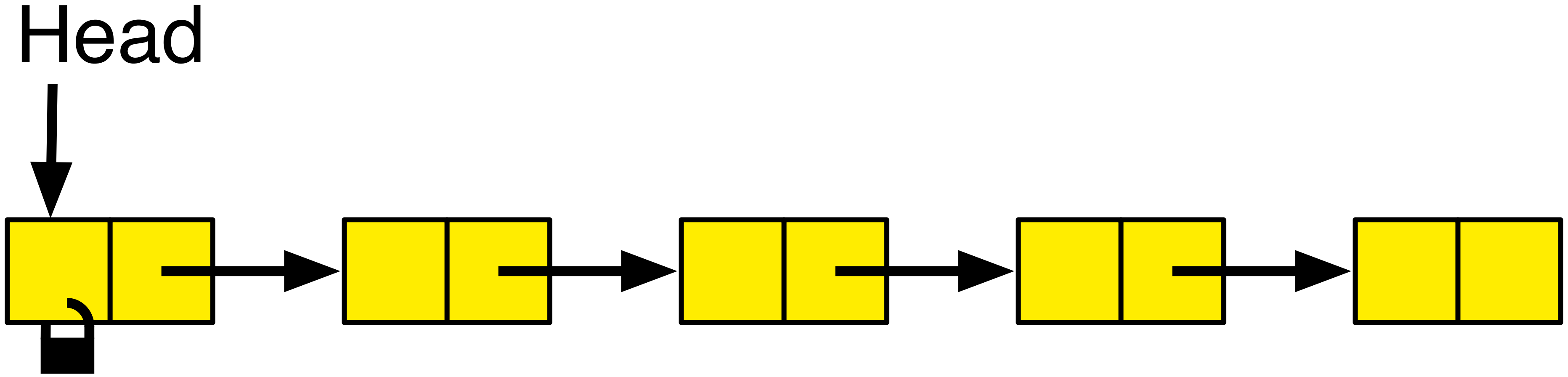
```
locate(e) {
  local p, c;
  p := hd;
  lock(p);
  c := p.next;
  while (c != nil) {
    lock(c);
    if (c.value = e)
      return (p,c);
    unlock(p);
    p := c;
    c := p.next;
  }
  return(p, c);
}
```

Lock the fixed head of the list.

Fine-grained locking



Fine-grained locking



Lock the head

```
locate(e) {
  local p, c;
  p := hd;
  lock(p);
  c := p.next;
  while (c != nil) {
    lock(c);
    if (c.value = e)
      return (p,c);
    unlock(p);
    p := c;
    c := p.next;
  }
  return(p, c);
}
```

Traverse down the list, hand-over-hand.

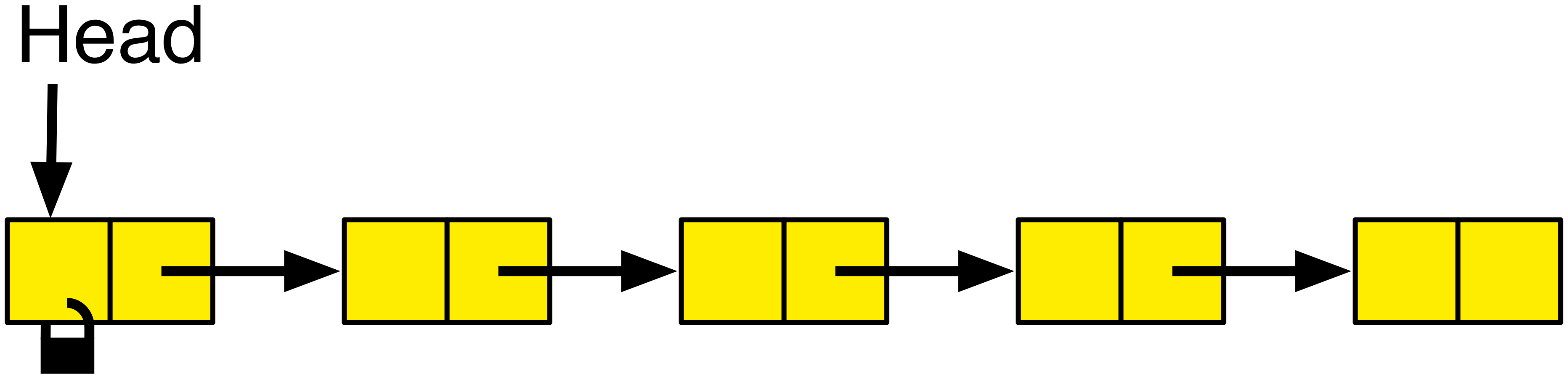
```
locate(e) {
  local p, c;
  p := hd;
  lock(p);
  c := p.next;
  while (c != nil) {
    lock(c);
    if (c.value = e)
      return (p,c);
    unlock(p);
    p := c;
    c := p.next;
  }
  return(p, c);
}
```

lock the next node in the list.

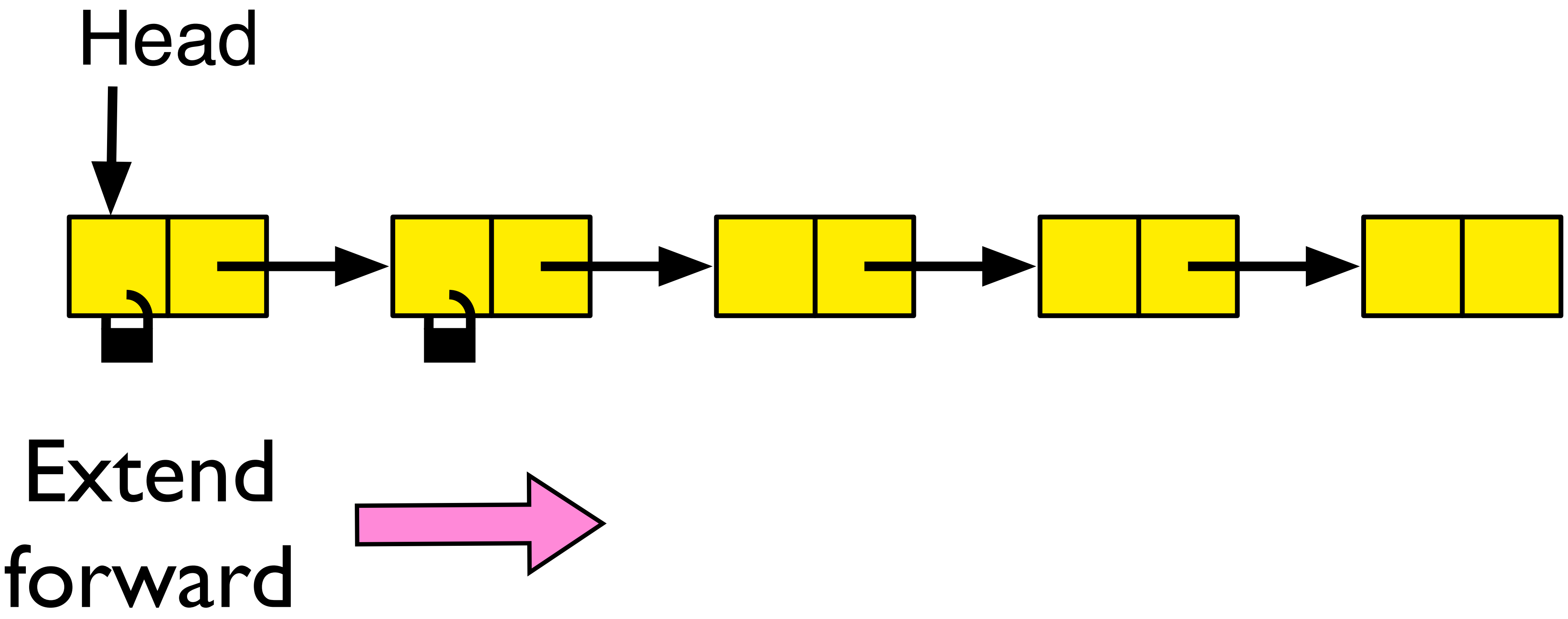
```
locate(e) {
  local p, c;
  p := hd;
  lock(p);
  c := p.next;
  while (c != nil) {
    lock(c);
    if (c.value = e)
      return (p,c);
    unlock(p);
    p := c;
    c := p.next;
  }
  return(p, c);
}
```

unlock the previous node.

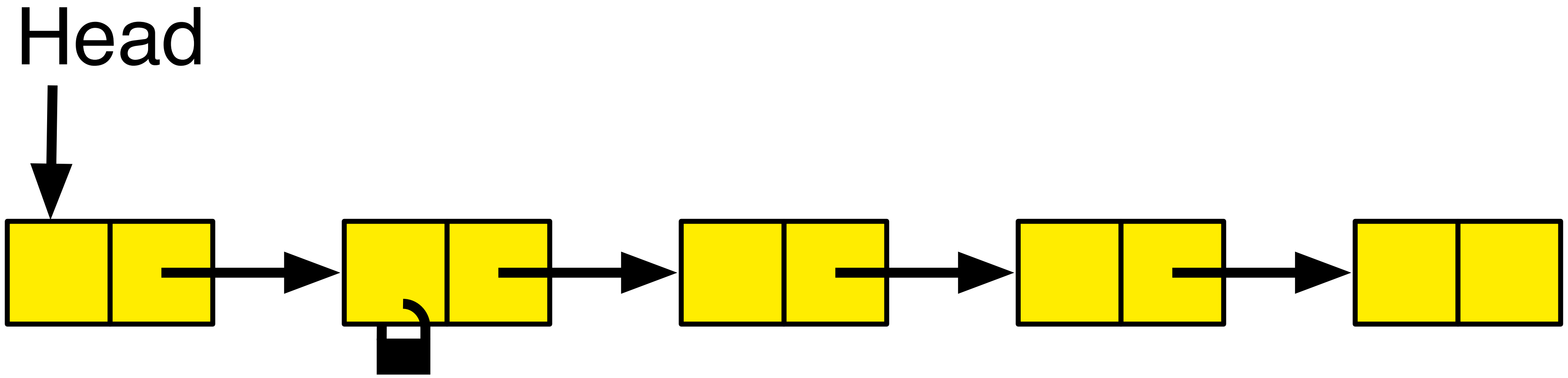
Fine-grained locking



Fine-grained locking

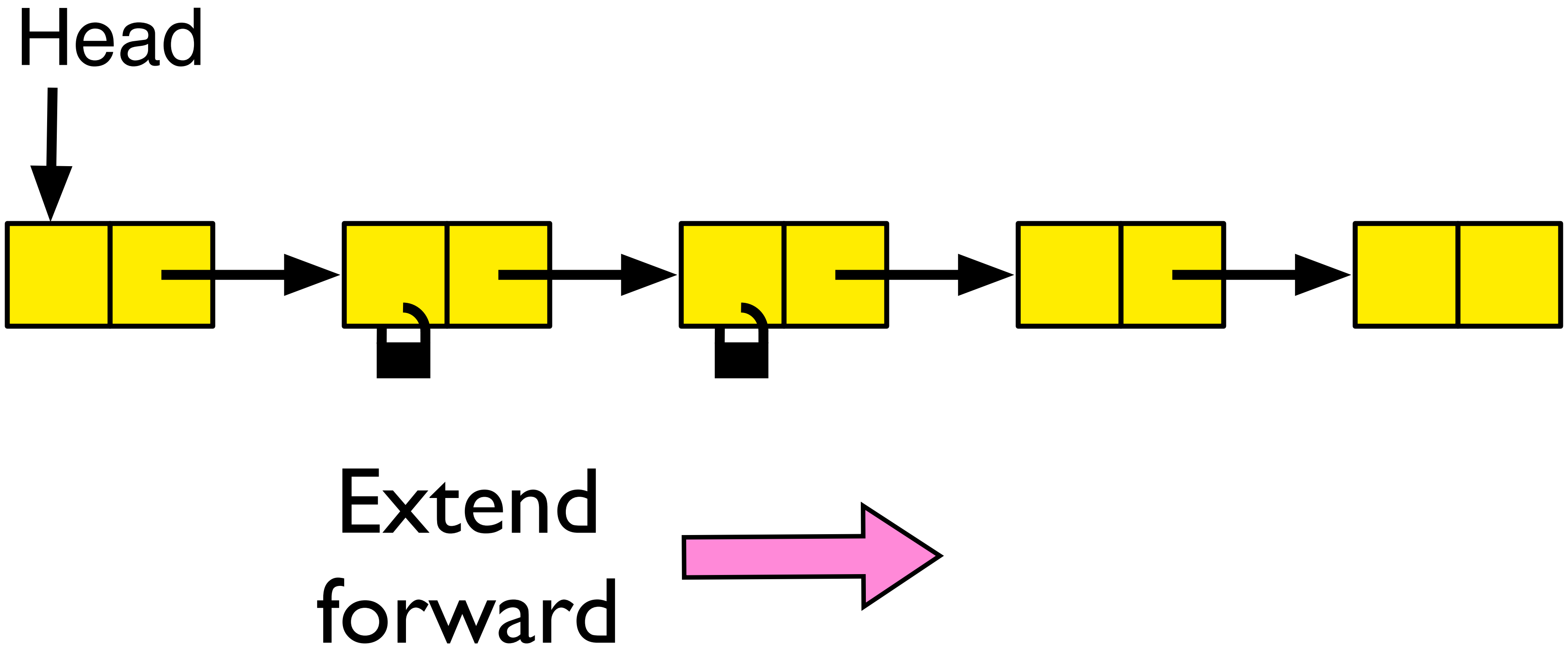


Fine-grained locking



Release previous lock

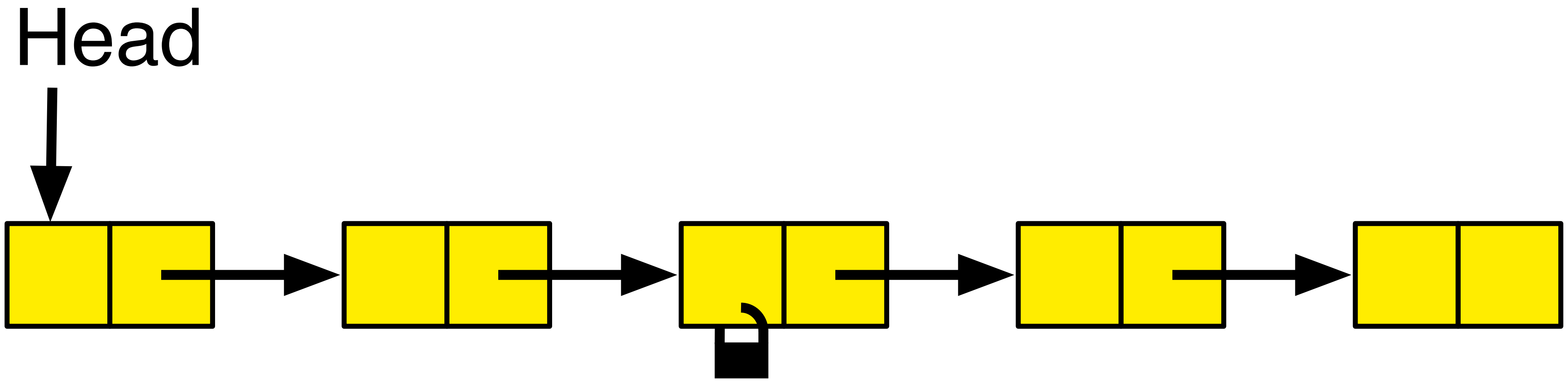
Fine-grained locking



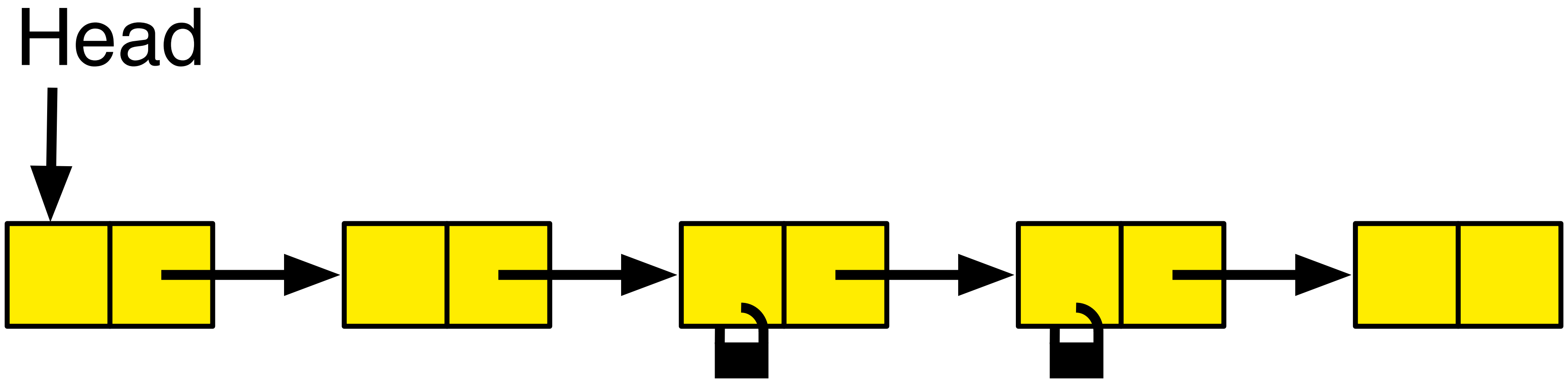
Head

Extend forward

Fine-grained locking



Fine-grained locking



```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y != nil) {
    z := y.next;
    x.next := z;
    dispose(y);
  }
  unlock(x);
}
```

If the located region is not nil, remove the node

```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y != nil) {
    z := y.next;
    x.next := z;
    dispose(y);
  }
  unlock(x);
}
```

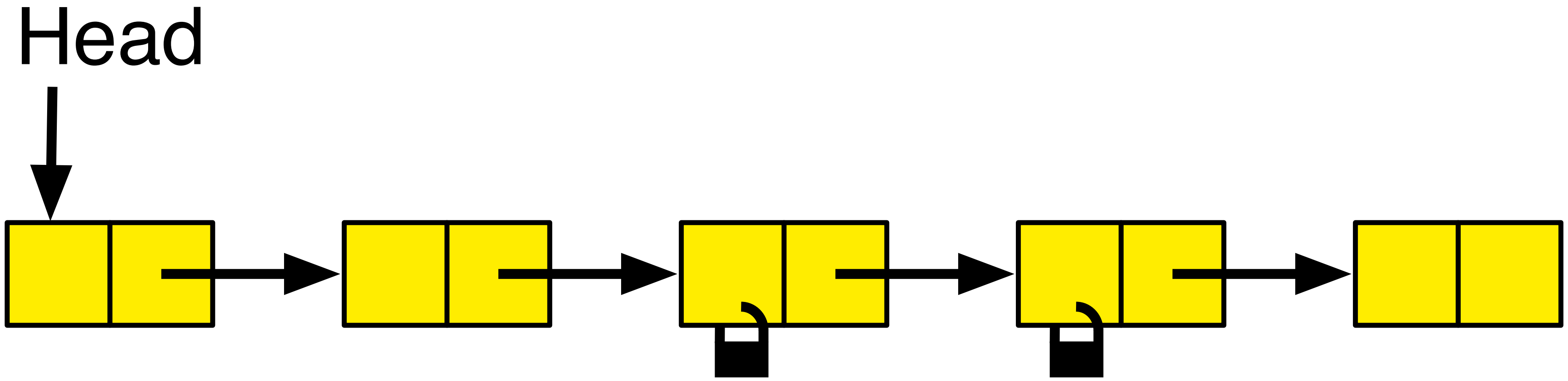
swing pointer forward to the next node

```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y != nil) {
    z := y.next;
    x.next := z;
    dispose(y);
  }
  unlock(x);
}
```

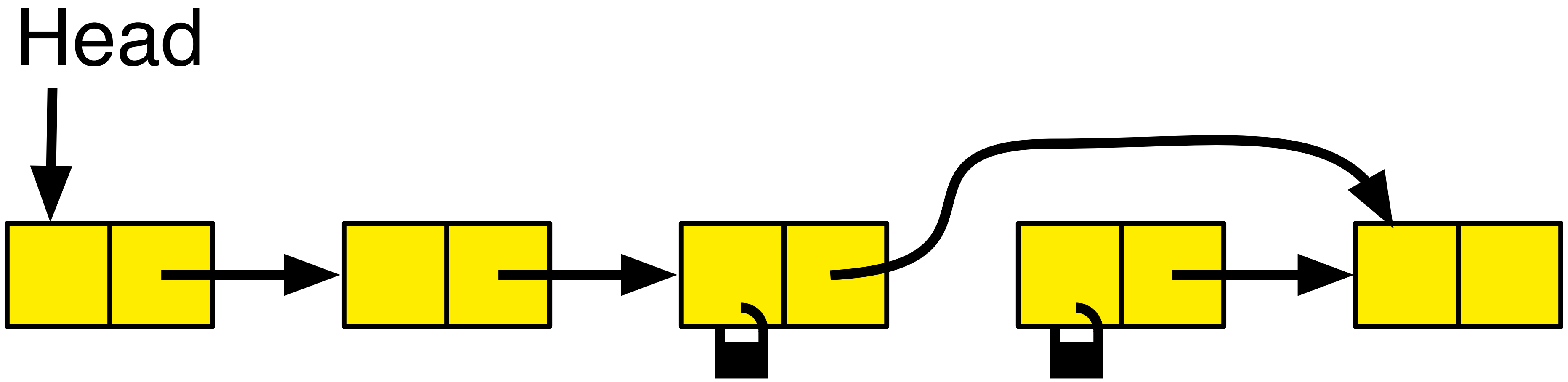
```
remove(e) {
  local x, y, z;
  (x, y) := locate(e);
  if (y != nil) {
    z := y.next;
    x.next := z;
    dispose(y);
  }
  unlock(x);
}
```

safely dispose of the removed node.

Fine-grained locking

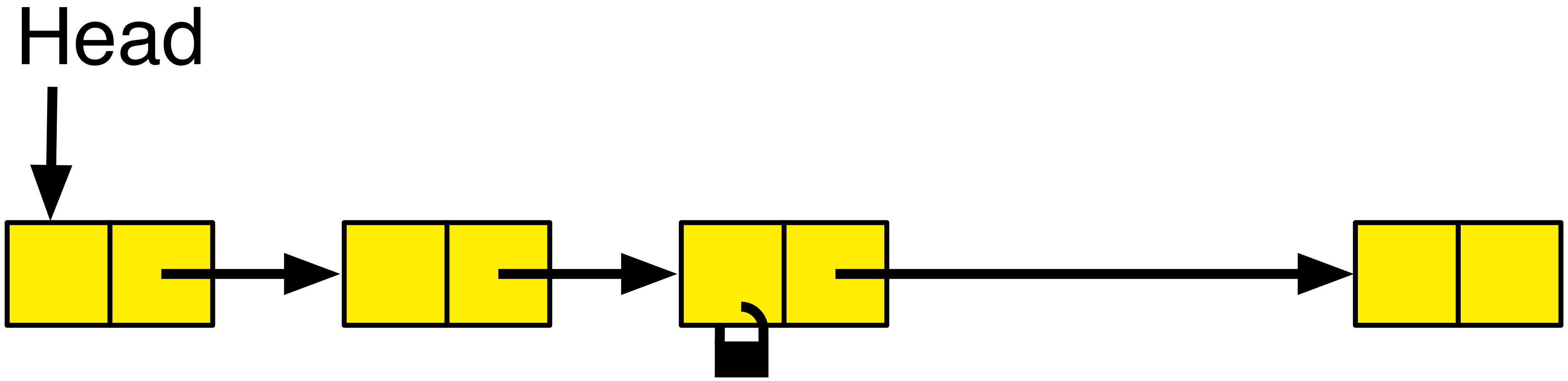


Fine-grained locking



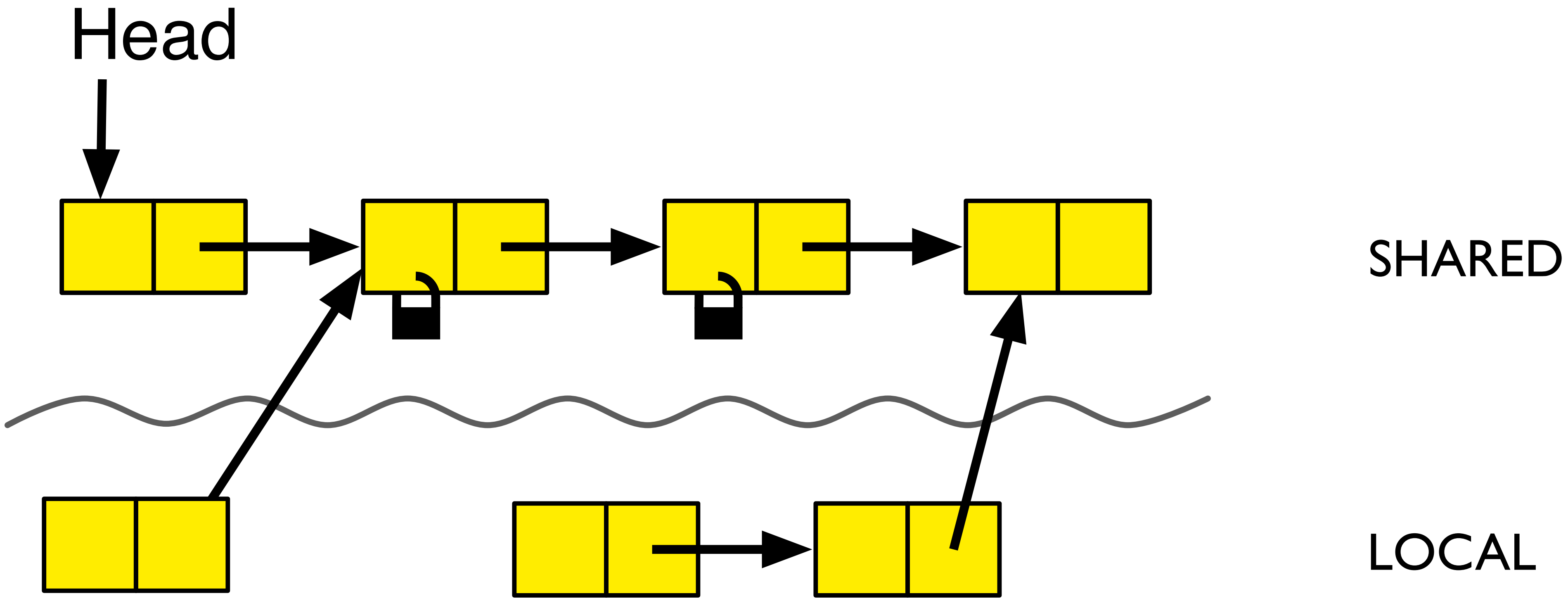
Remove locked node
by a pointer swing

Fine-grained locking



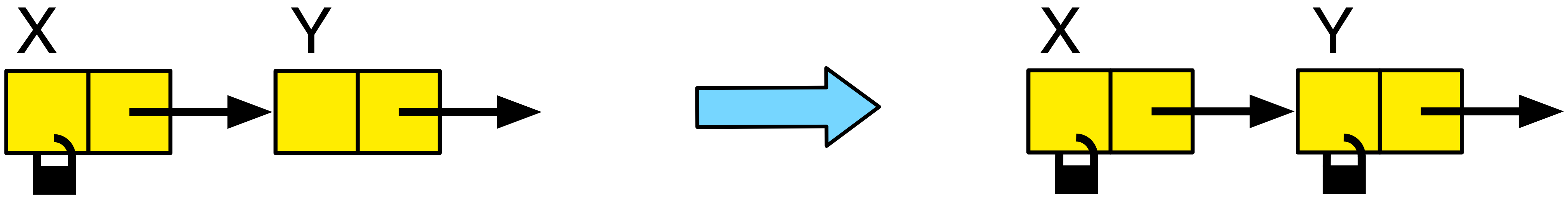
Free the
redundant node

RGSep proof



Locked nodes stay in the shared state.

RGSep actions

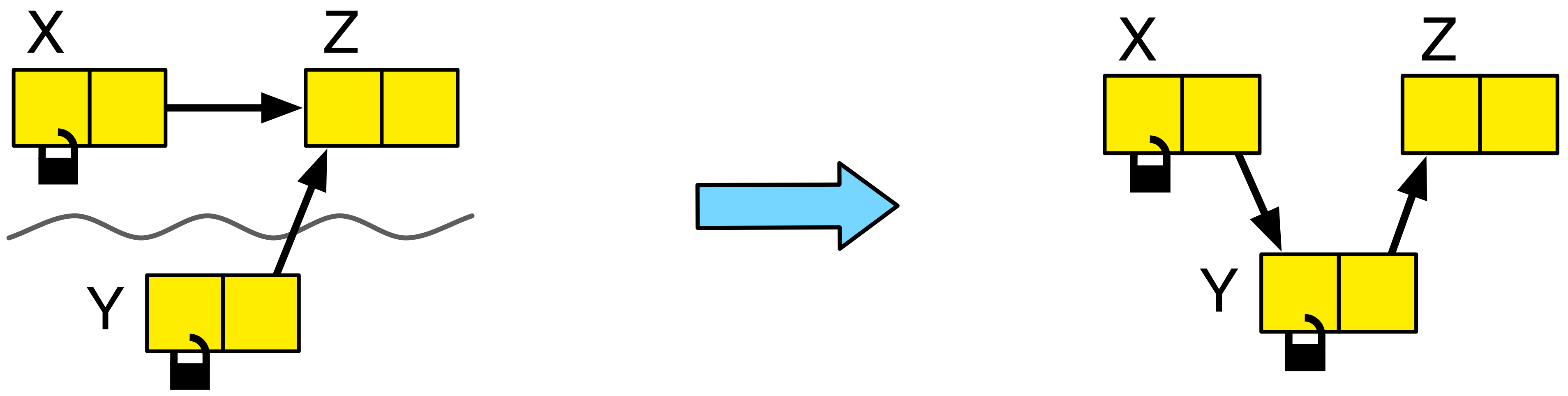


Consequently, blocks have to be manipulated *explicitly* by actions.

RGSep actions

Nodes added and removed by explicit pointer swings in the shared state.

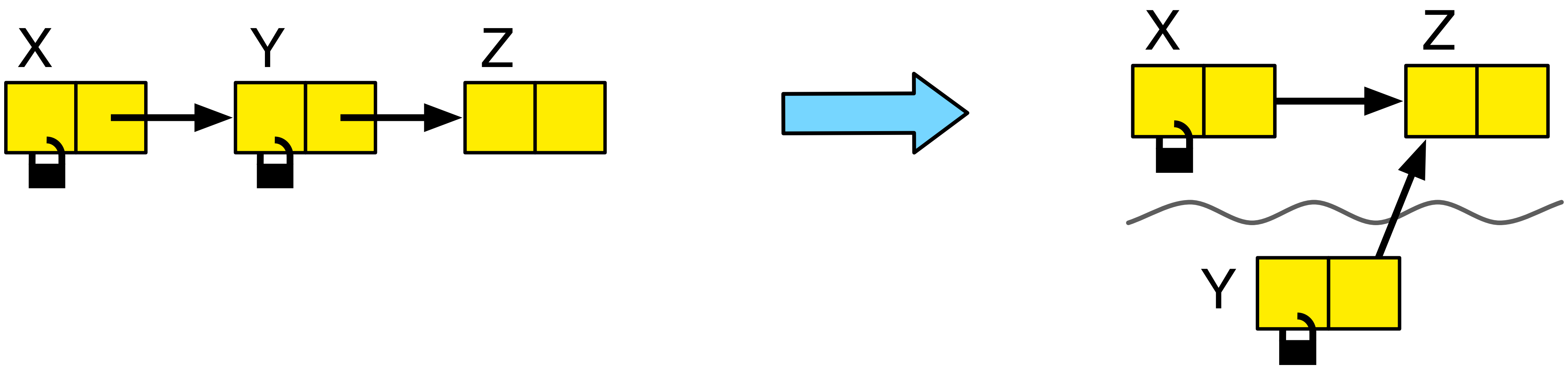
ADD:



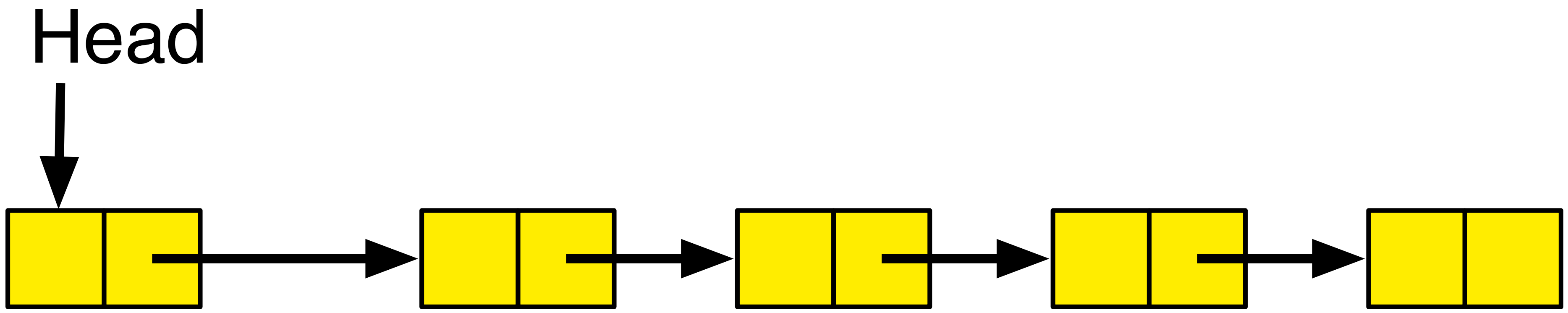
RGSep actions

Nodes added and removed by explicit pointer swings.

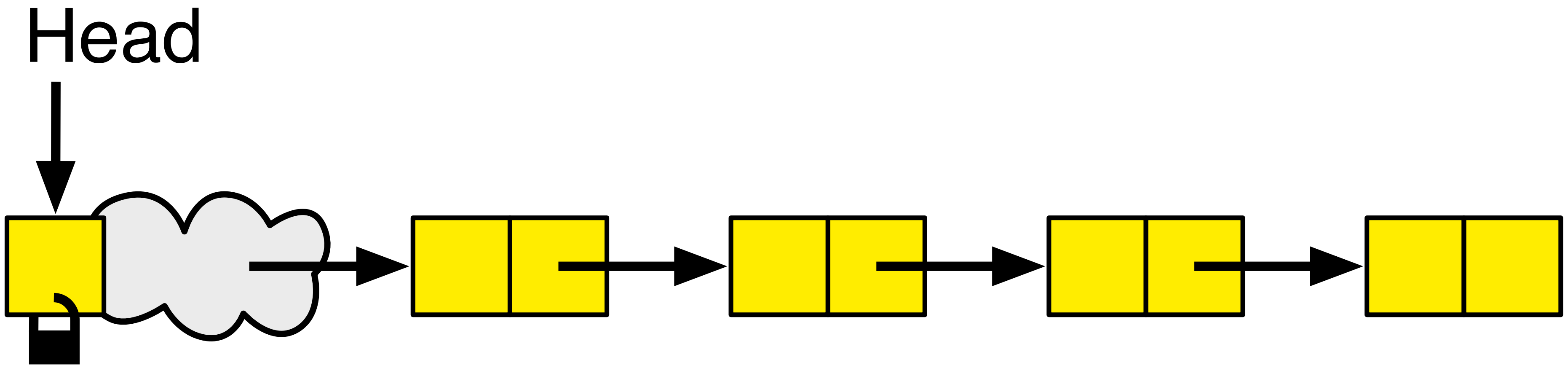
REMOVE:



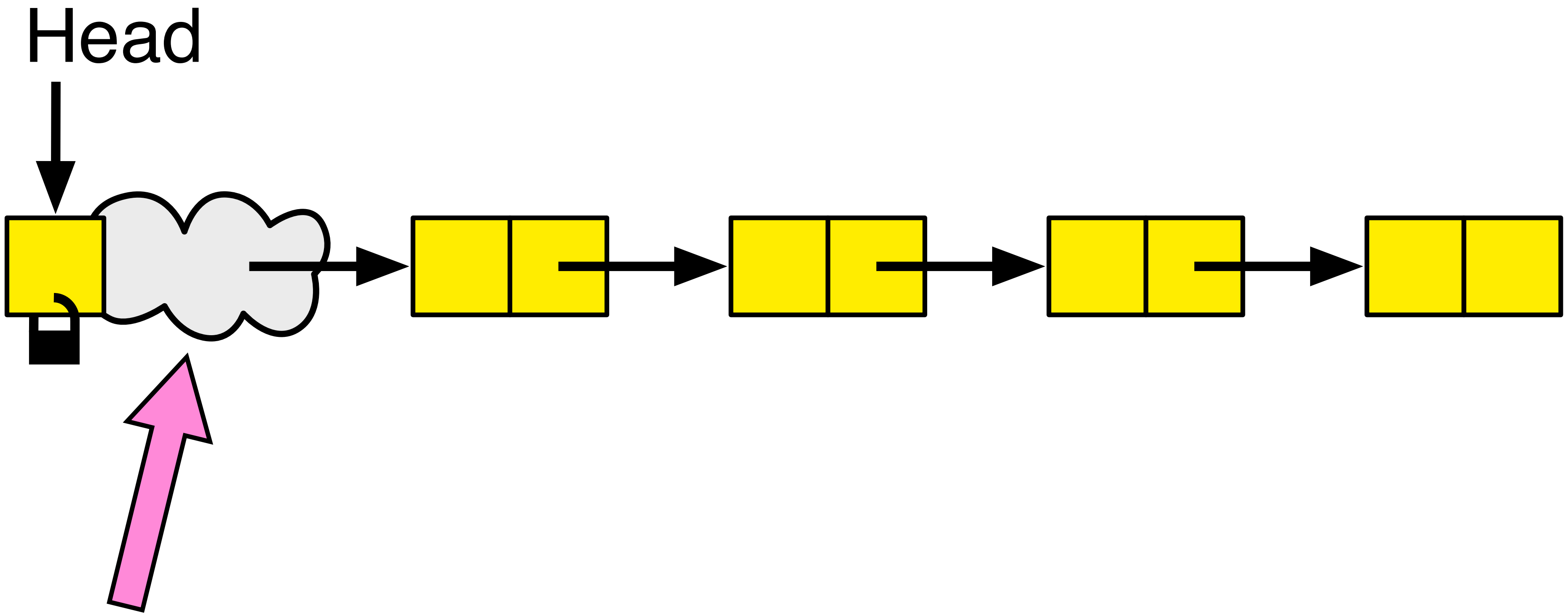
Conceptual view



Conceptual view

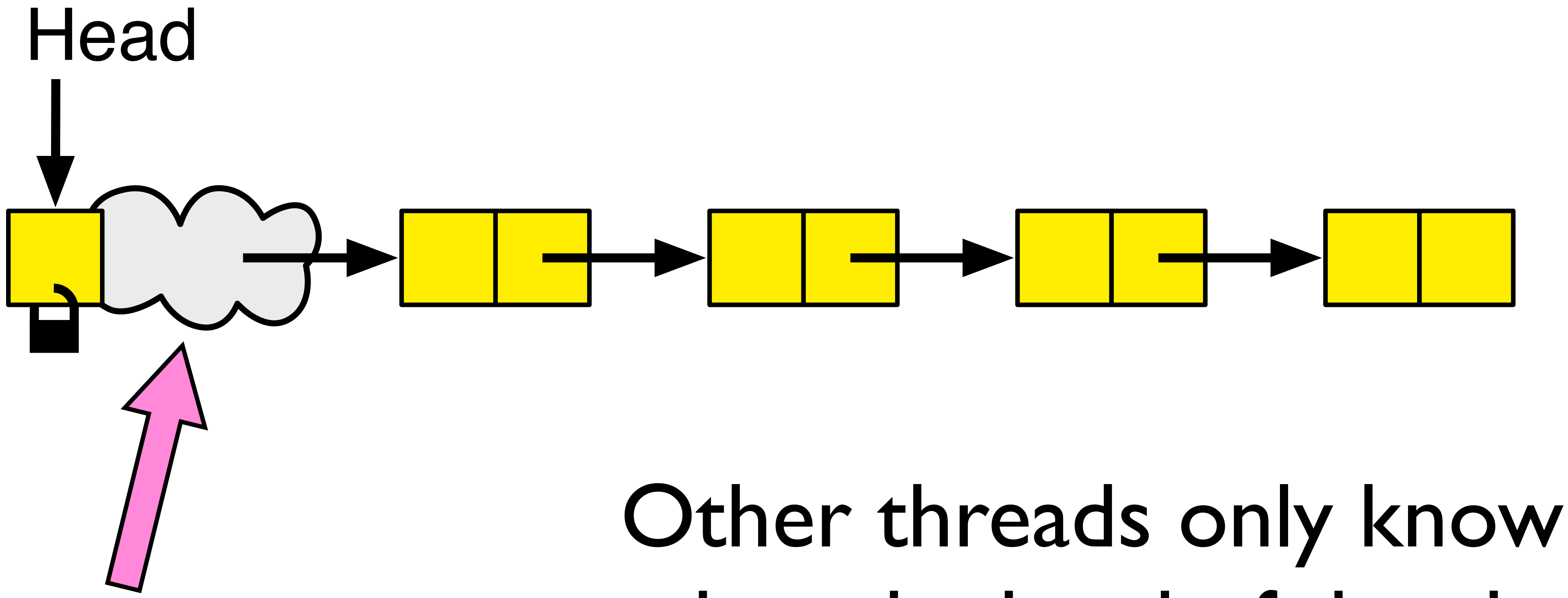


Conceptual view



cloud denotes a
hidden portion
of the list

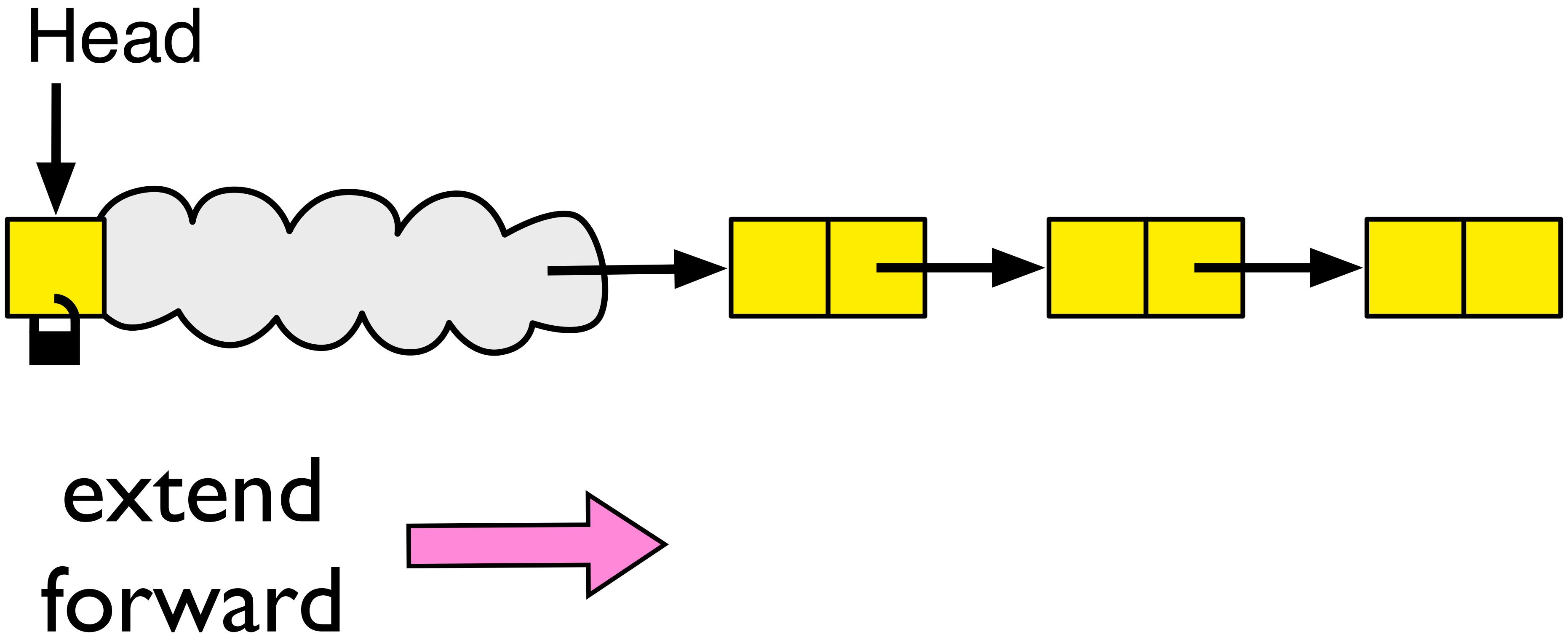
Conceptual view



cloud denotes a hidden portion of the list

Other threads only know that when the head of the cloud is unlocked, the list structure will be restored

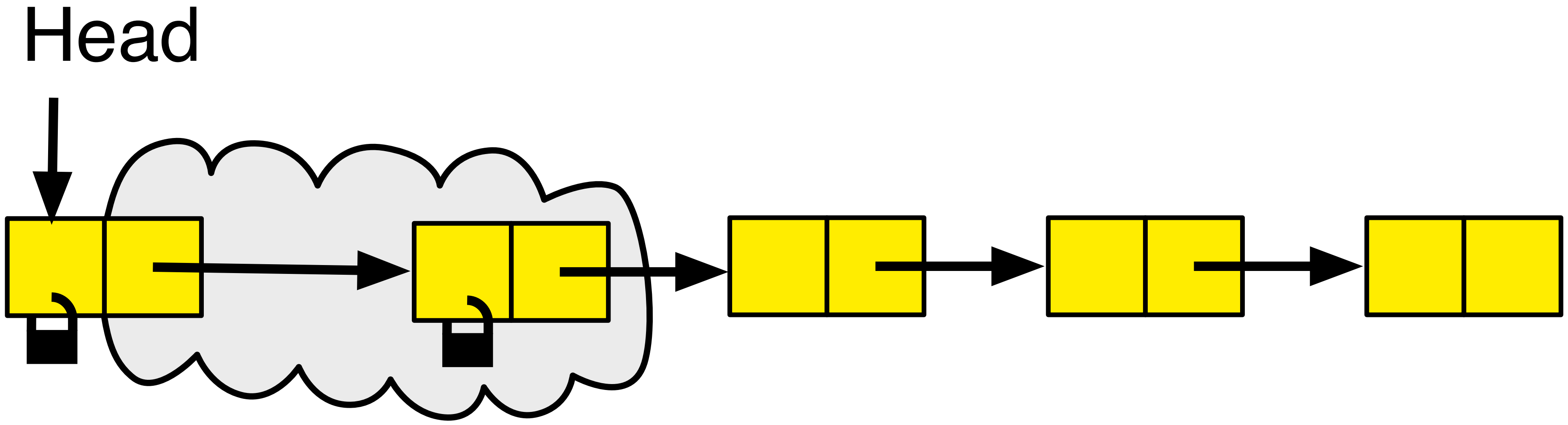
Conceptual view



Head

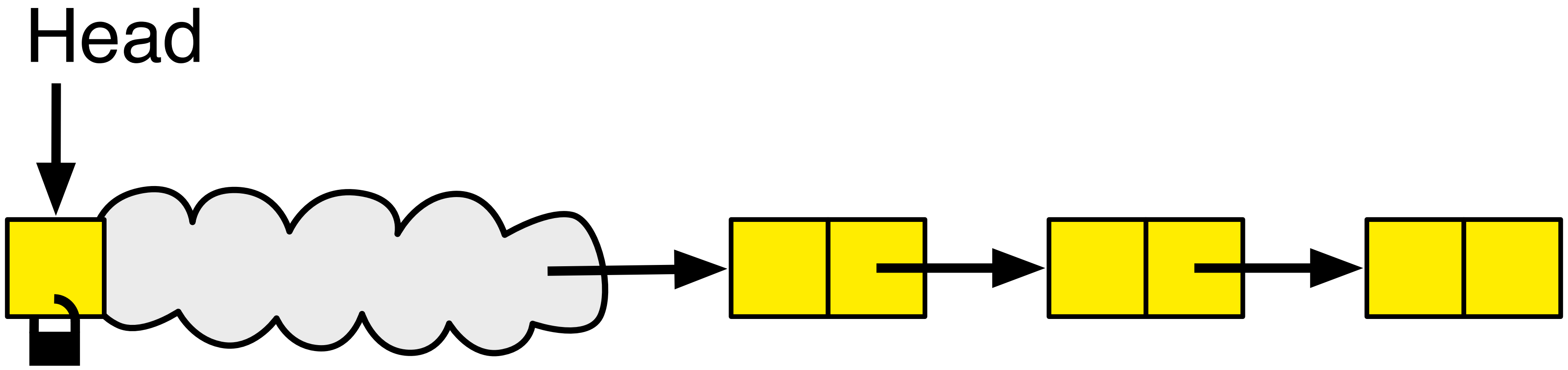
extend forward

Conceptual view

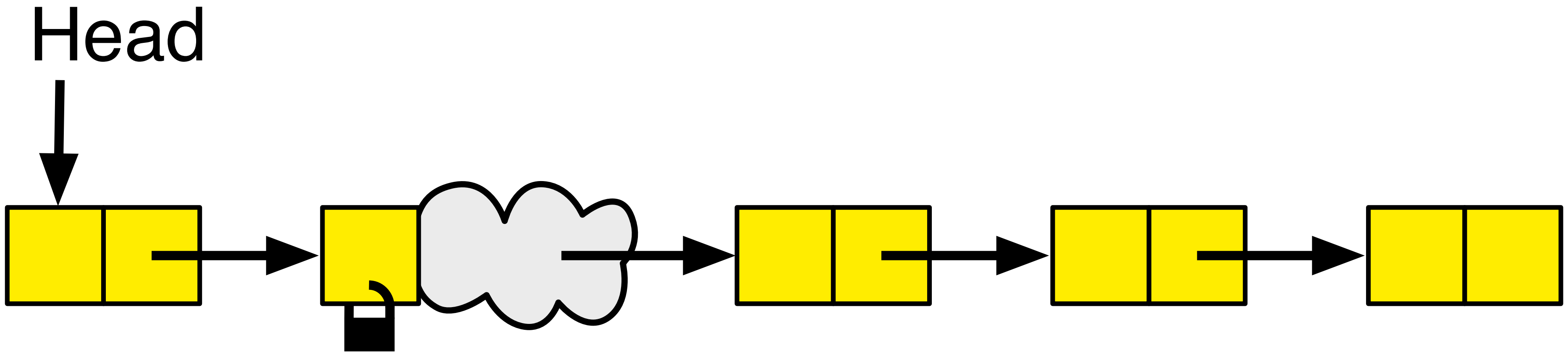


locked nodes are hidden
from the public state

Conceptual view

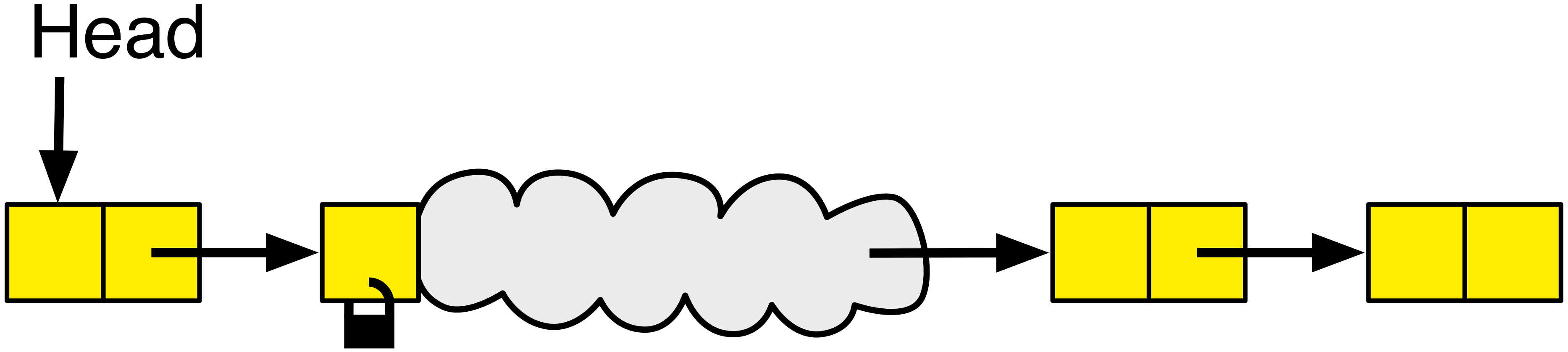


Conceptual view

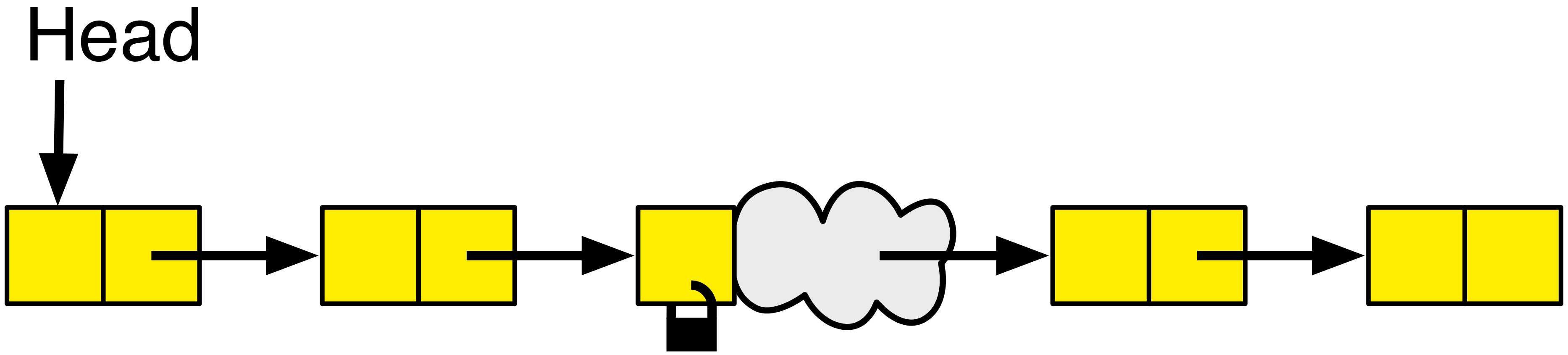


release some list from
the hidden state

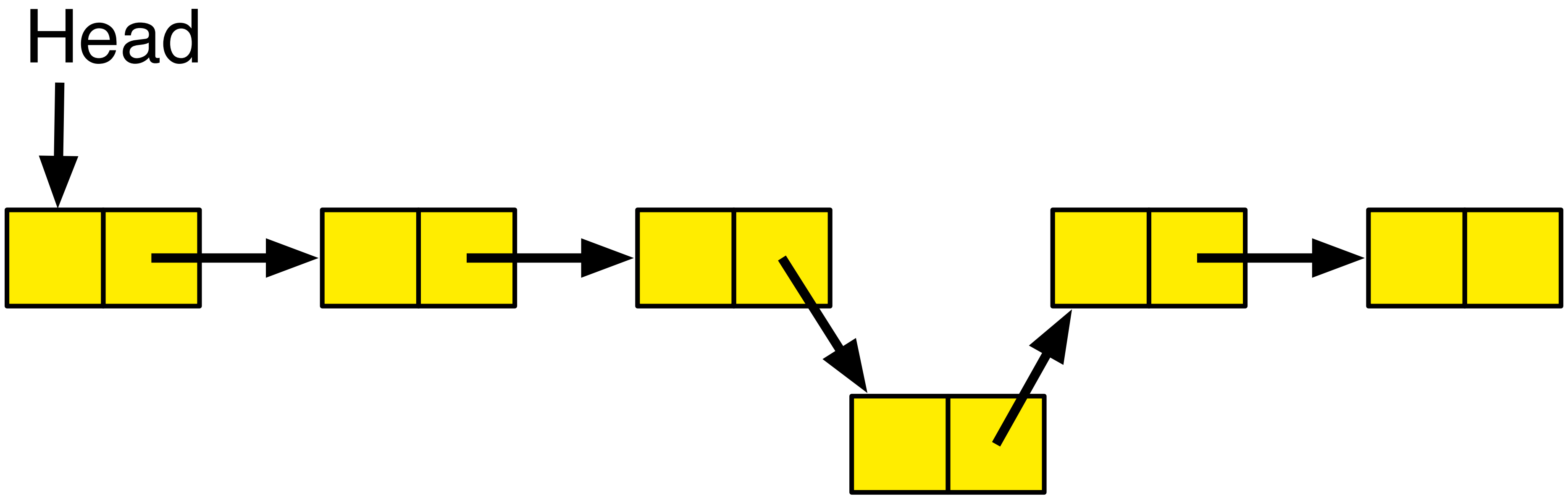
Conceptual view



Conceptual view

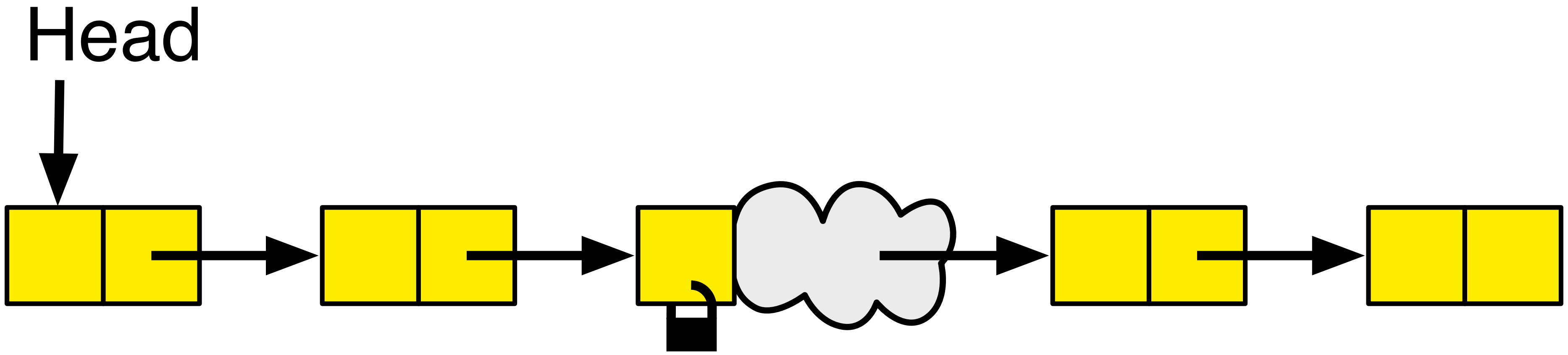


Conceptual view

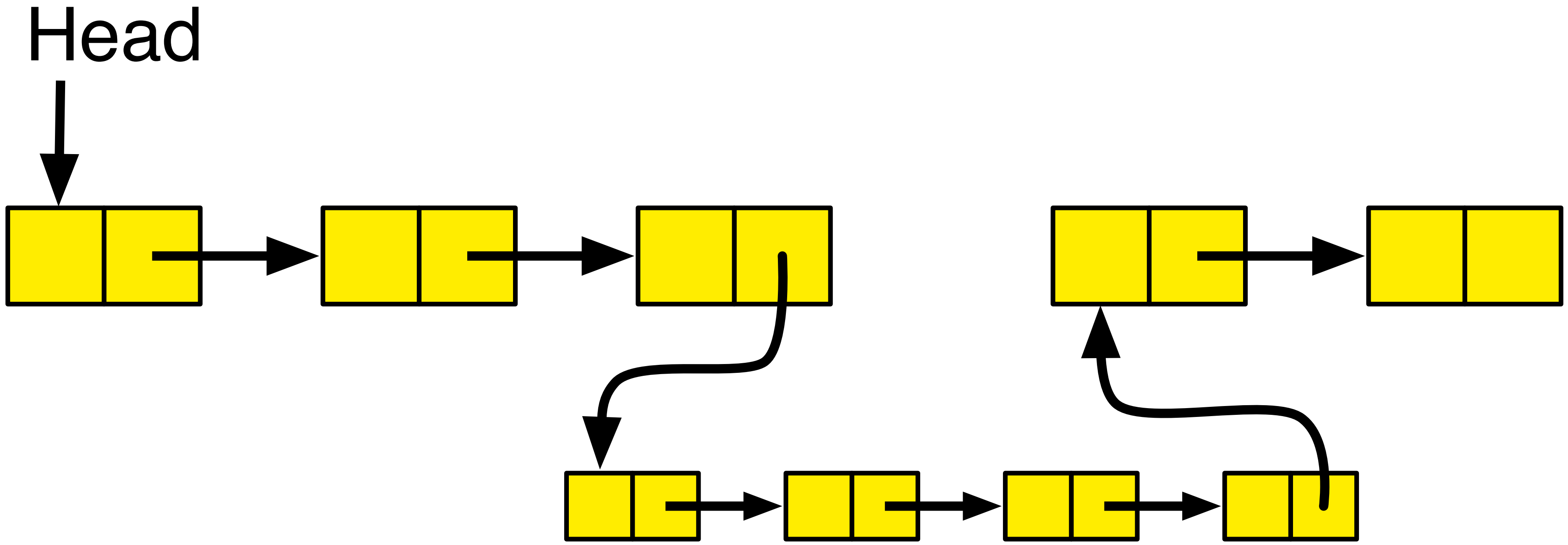


can remove the lock and release
a single node from hidden area

Conceptual view

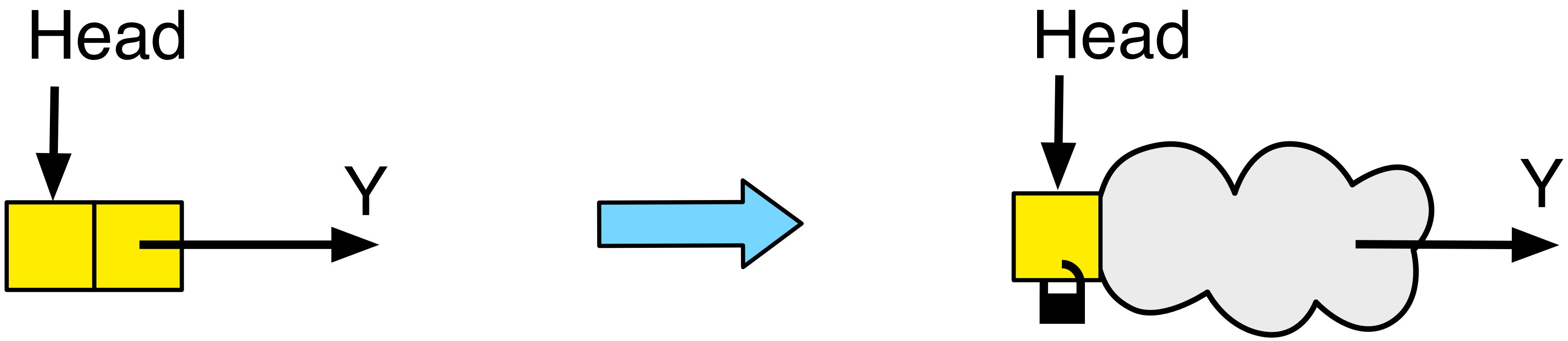


Conceptual view



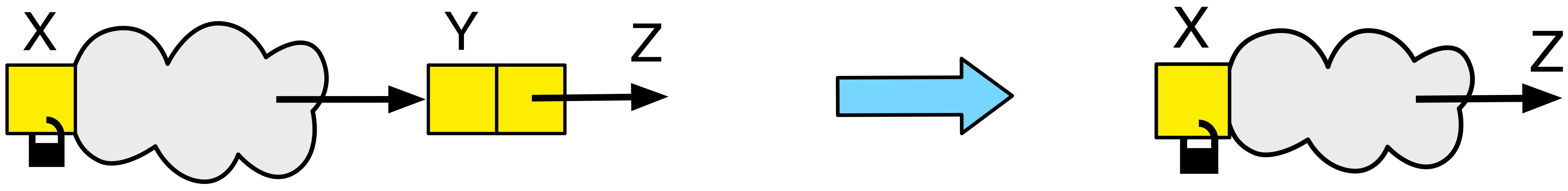
can replace hidden area
with arbitrary unlocked list

Intuitive actions



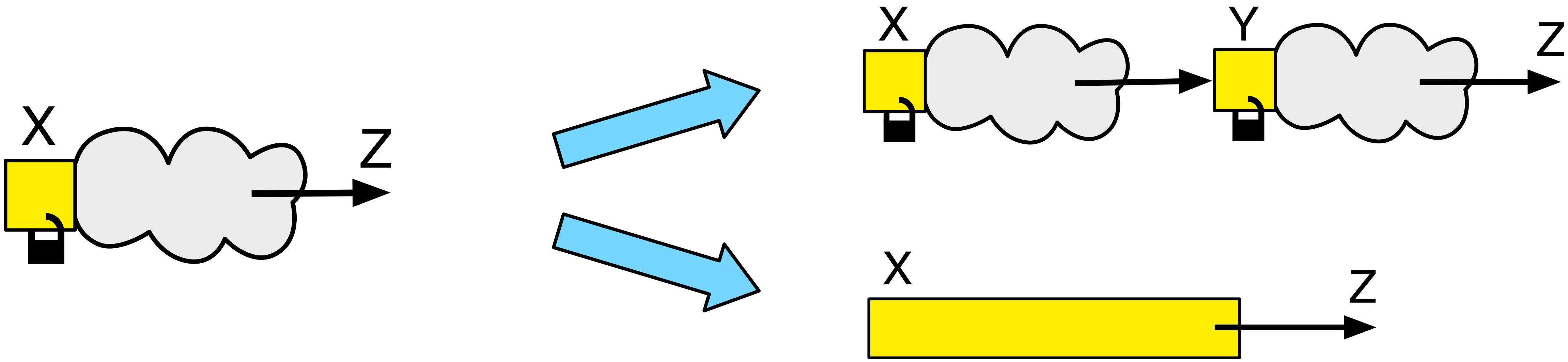
Lock the head and
add a hidden section

Intuitive actions



Extend the hidden section
(hide more of the list)

Intuitive actions



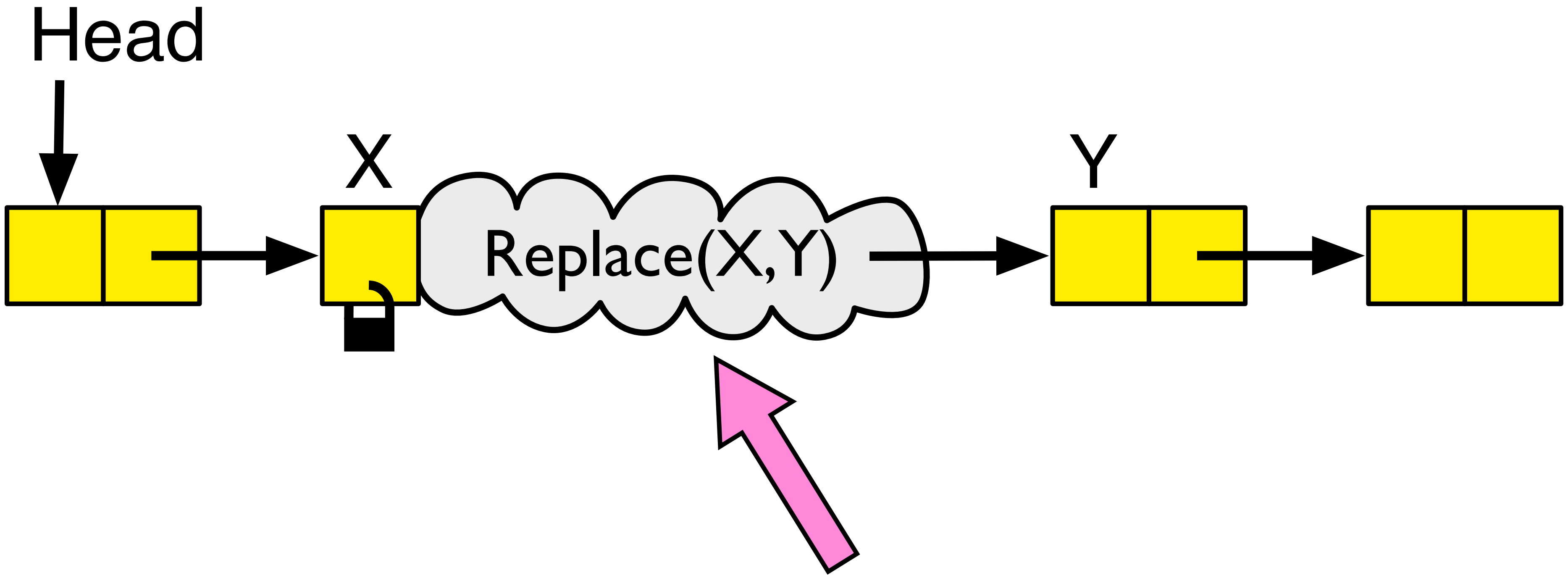
Split the hidden section,

or

return an unlocked list segment

Deny-guarantee solution:

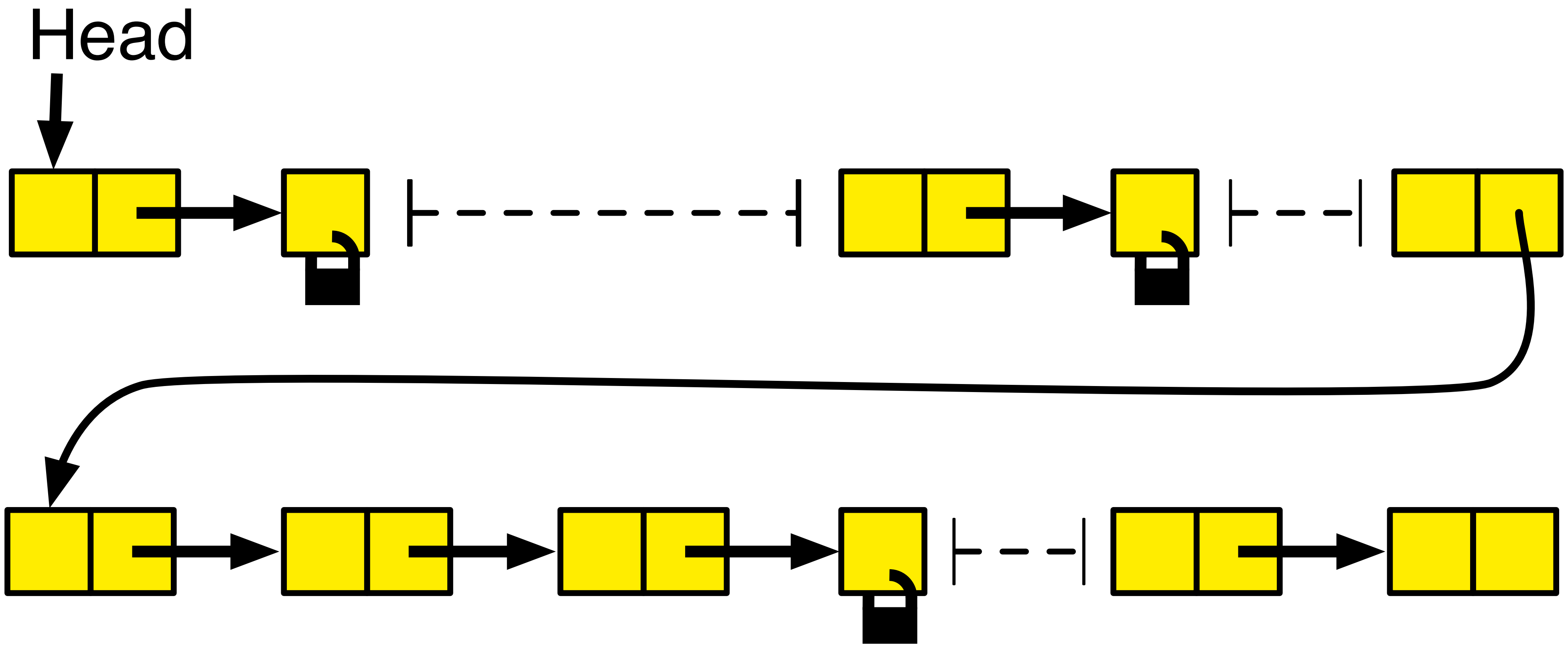
Model gaps in the list as the permission to insert something into the gap.



Permission to insert a list from (X+1) to Y, or extend the permission

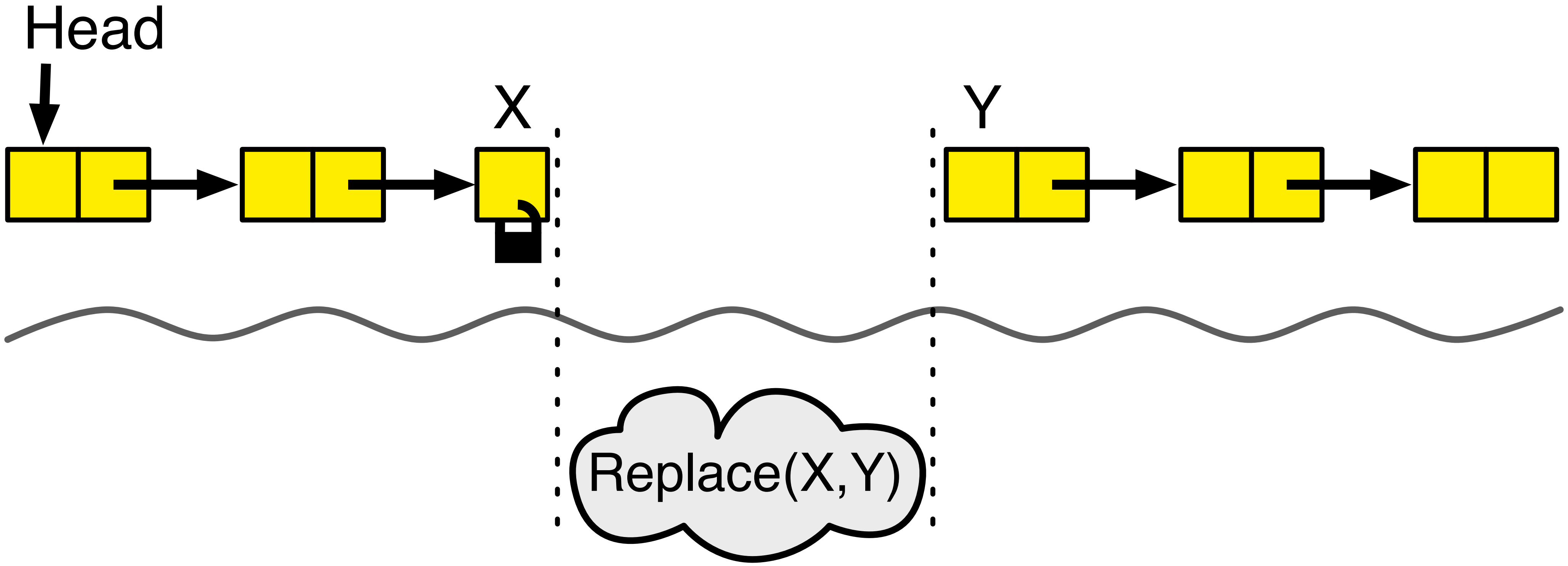
Shared state

Shared state consists of a list with gaps:

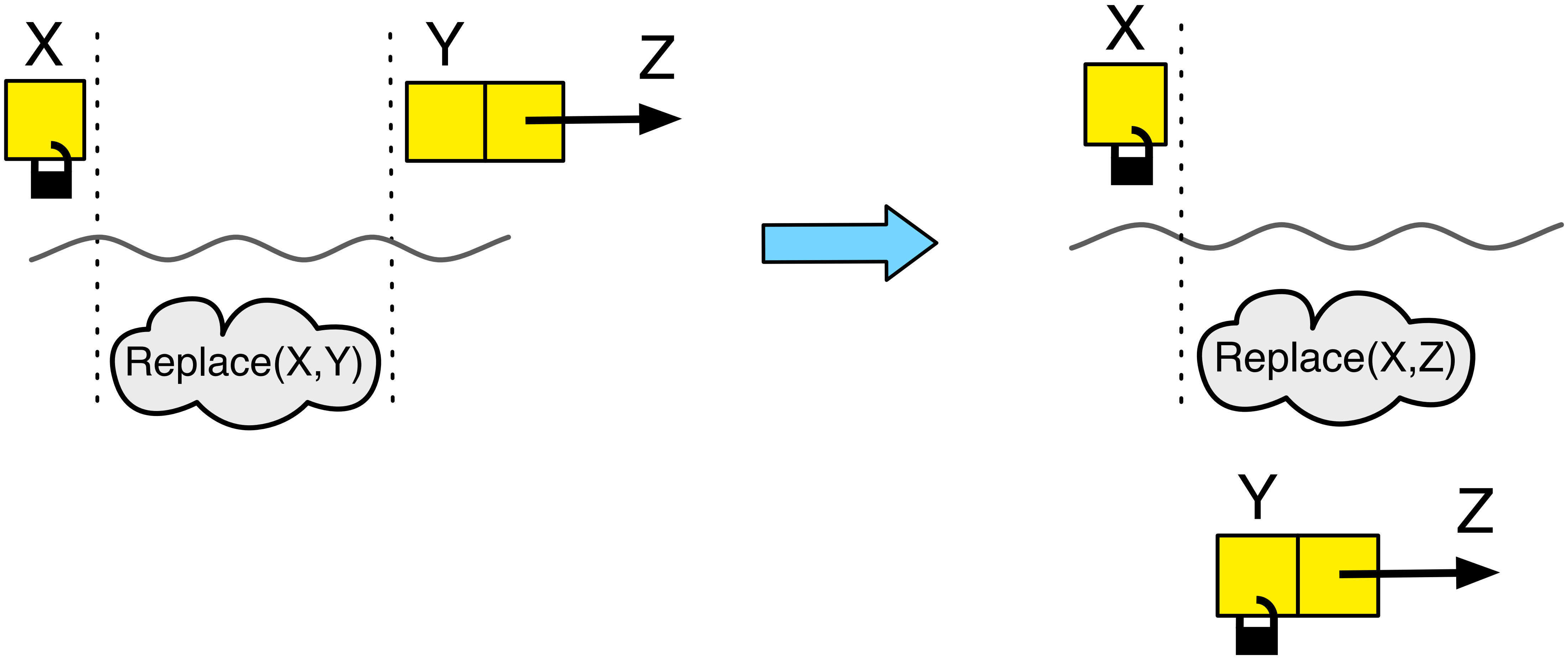


Local state

Permissions on gaps are held in local state

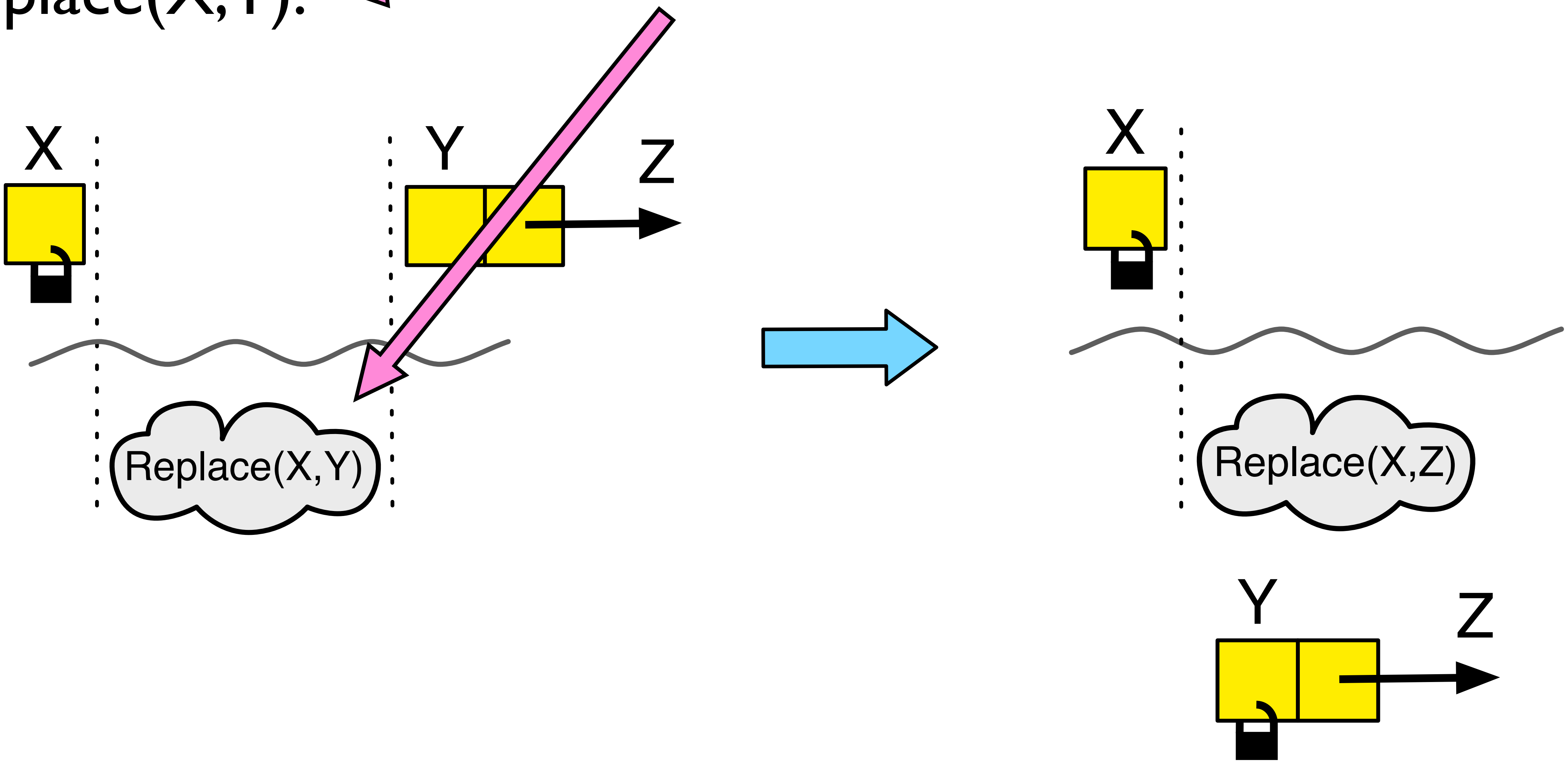


Replace(X,Y):



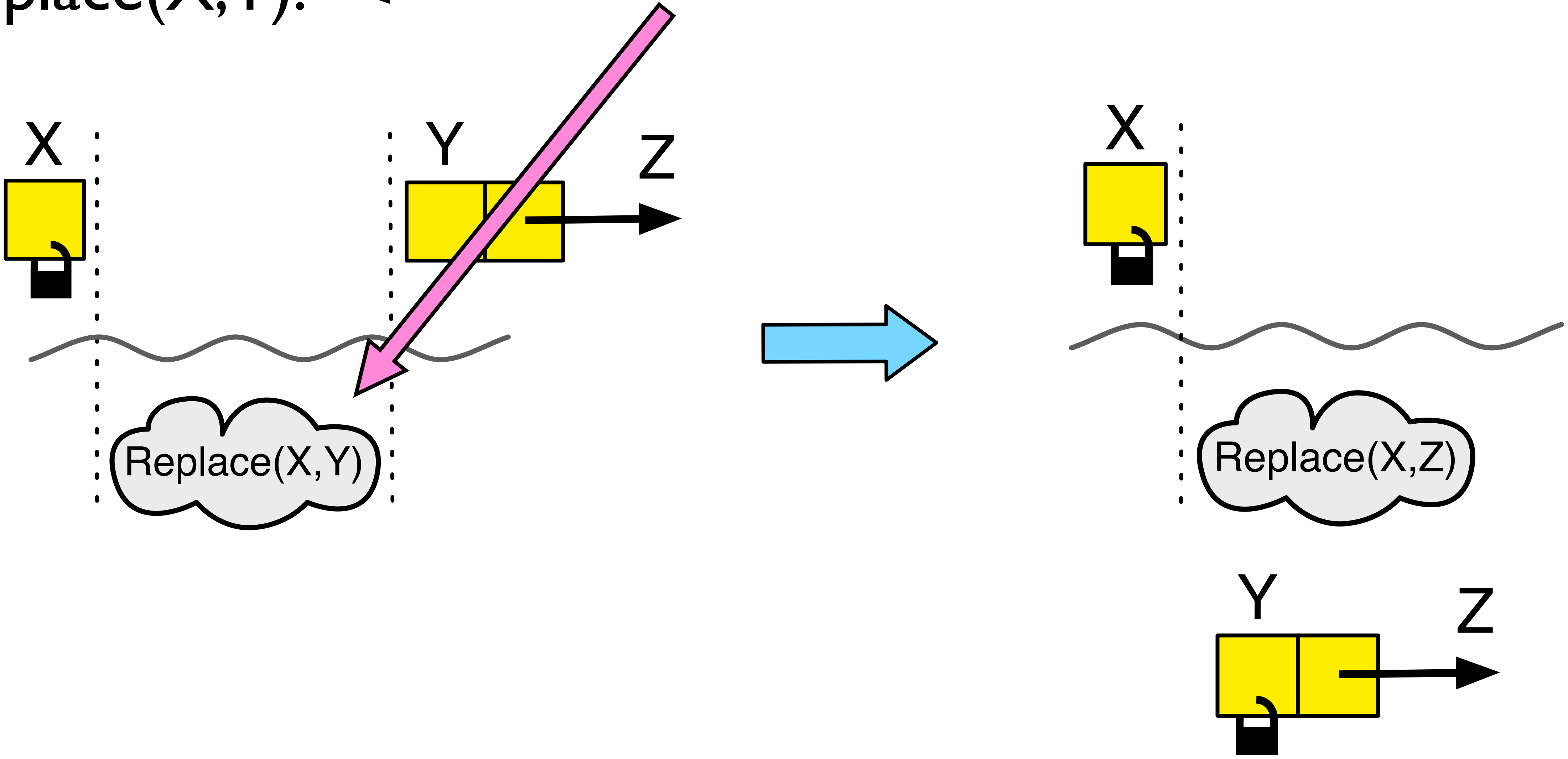
Replace(X,Y):

note recursion

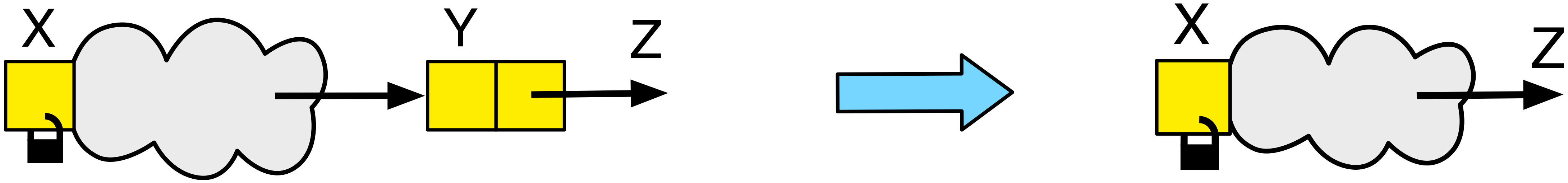


Replace(X,Y):

note recursion

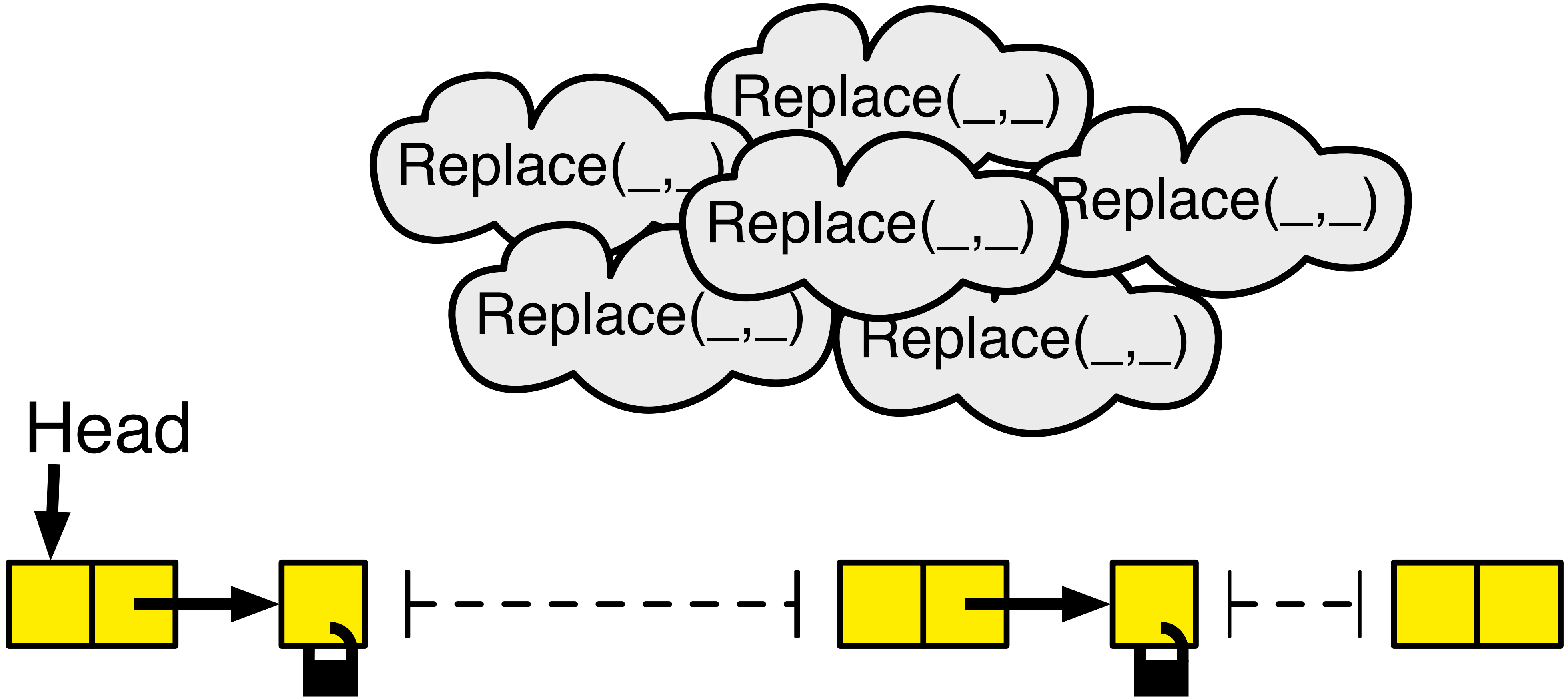


Corresponds to this intuitive action:

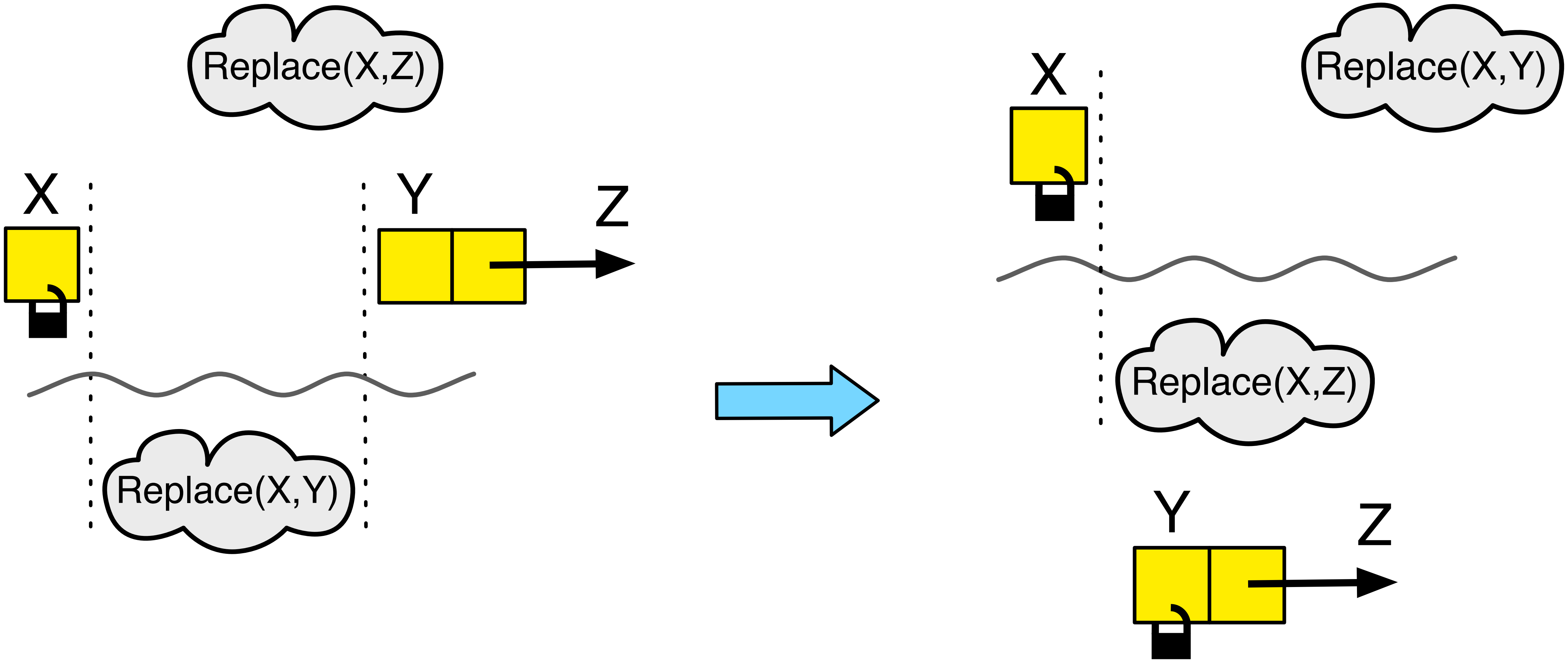


Unused actions

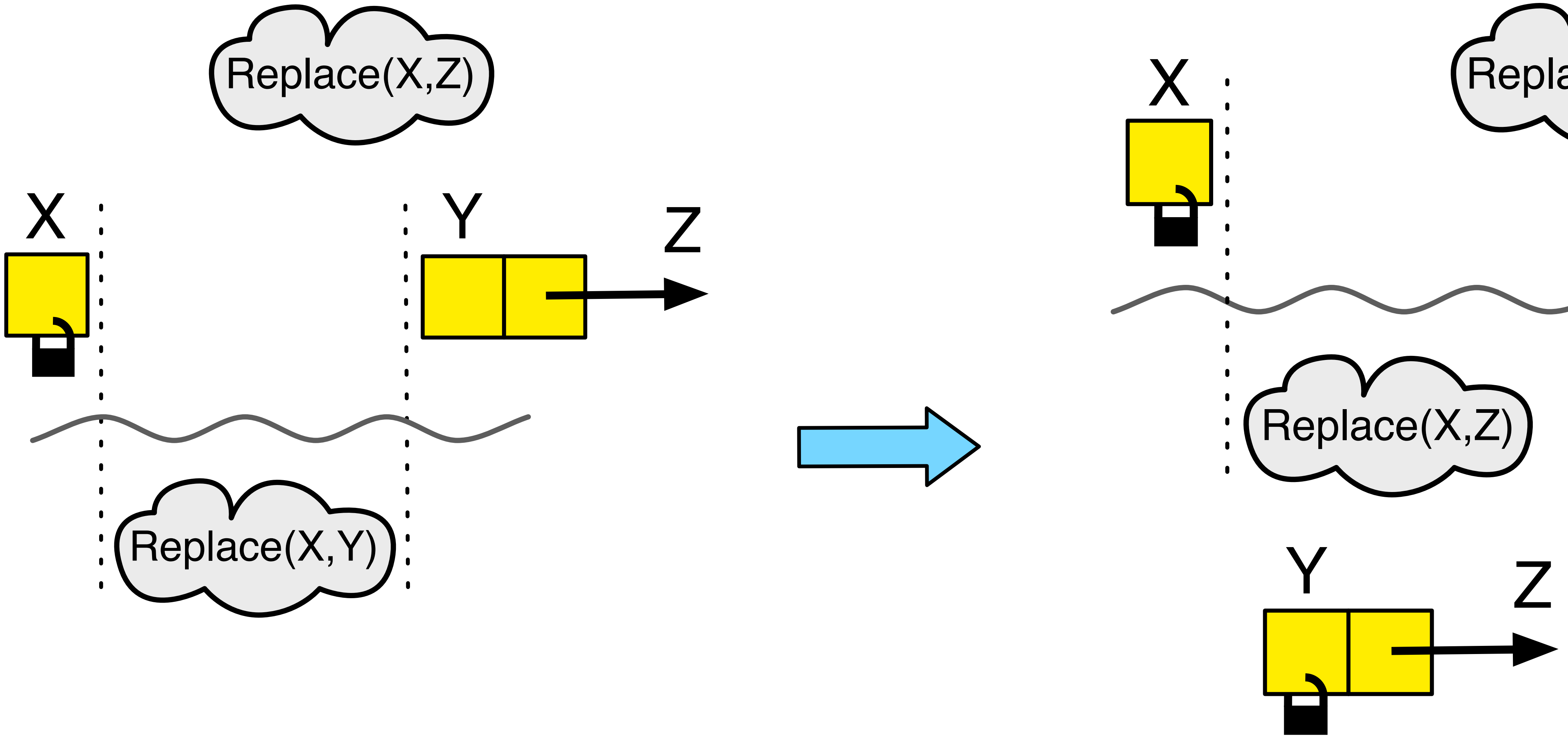
Unused permissions are held in the public state.



Replace(X,Y):

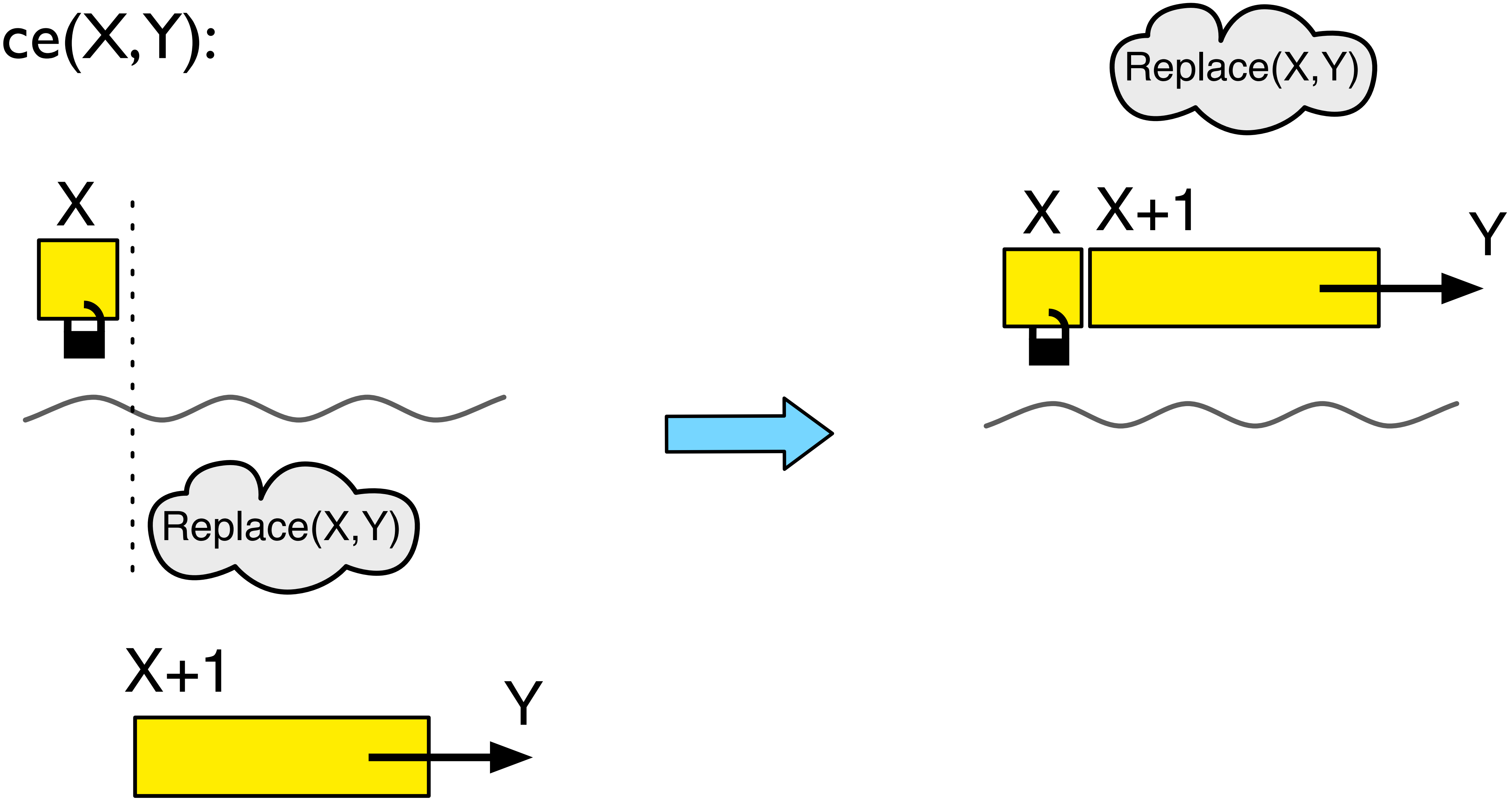


Replace(X,Y):

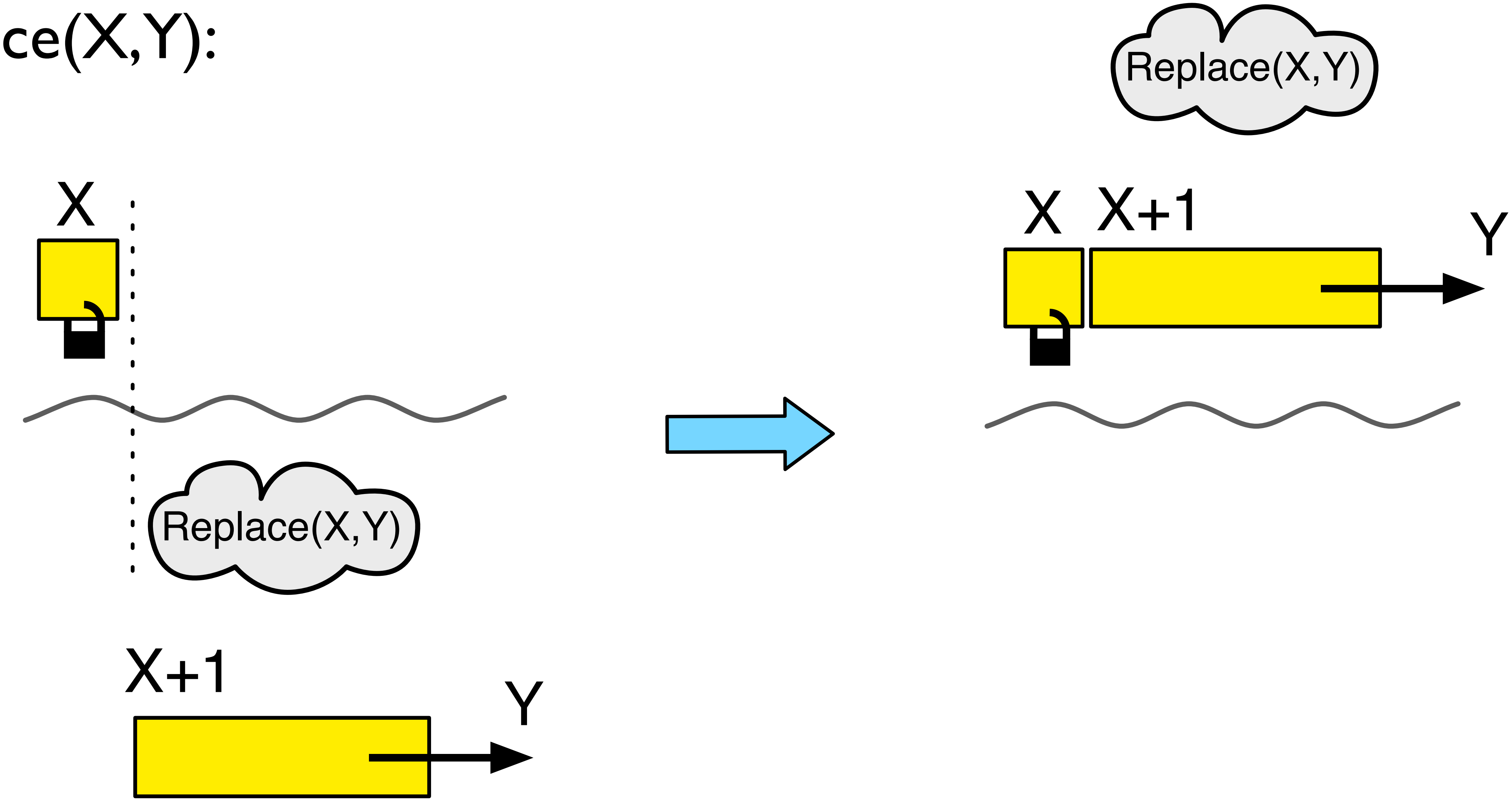


$$\textsc{Rep}(x,y):\ \ \mathsf{L}(x) * [\textsc{Rep}(x,z)]_1 * \mathsf{Un}(y,v,z) \leadsto \mathsf{L}(x) * [\textsc{Rep}(x,y)]_1$$

Replace(X,Y):

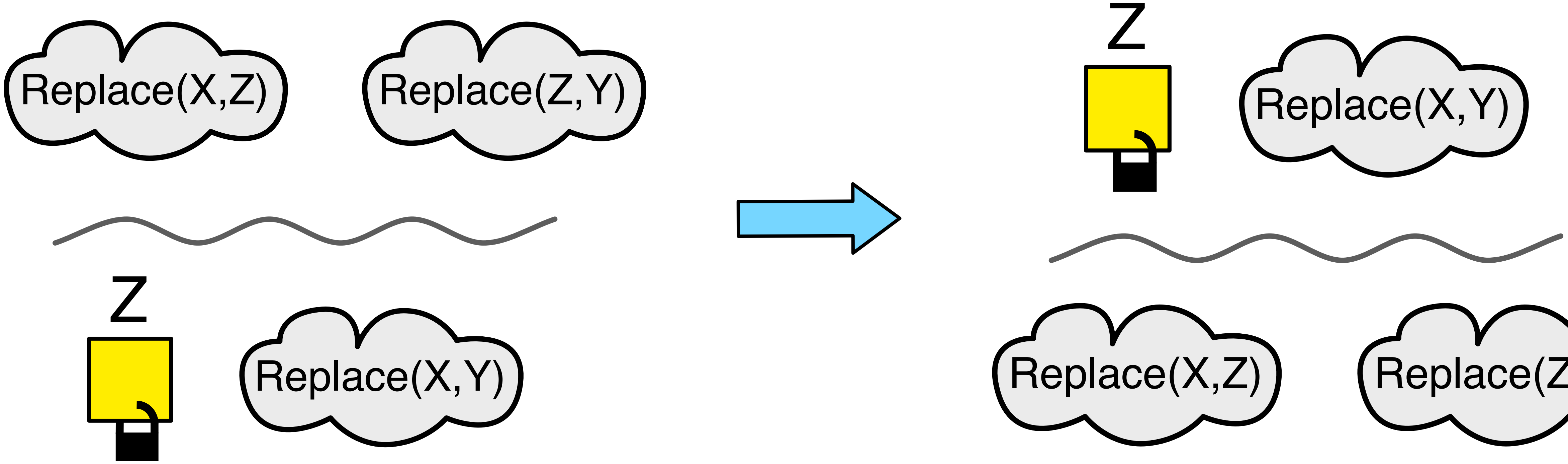


Replace(X,Y):

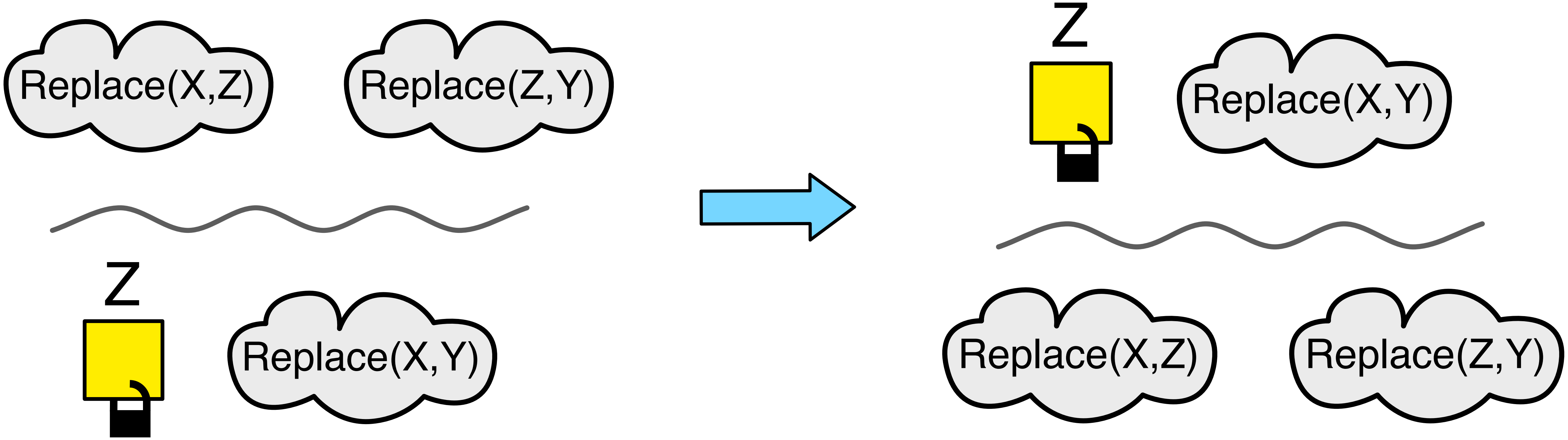


$$\textsc{Rep}(x,y) : \quad \mathsf{L}(x) \quad \rightsquigarrow \quad \mathsf{L}(x) * \mathsf{lseg}(x+1,y) * [\textsc{Rep}(x,y)]_1 *$$

Replace(X,Y):


Replace(X,Z)
Replace(Z,Y)
Z
Replace(X,Y)
Z
Replace(X,Y)
Replace(X,Z)
Replace(Z,Y)

Replace(X,Y):



$$\textsc{Rep}(x,y): \quad [\textsc{Rep}(x,z)]_1 * [\textsc{Rep}(z,y)]_1 \quad \rightsquigarrow \quad \mathsf{L}(z) * [\textsc{Rep}(x,y)]_1$$

Algorithm insights

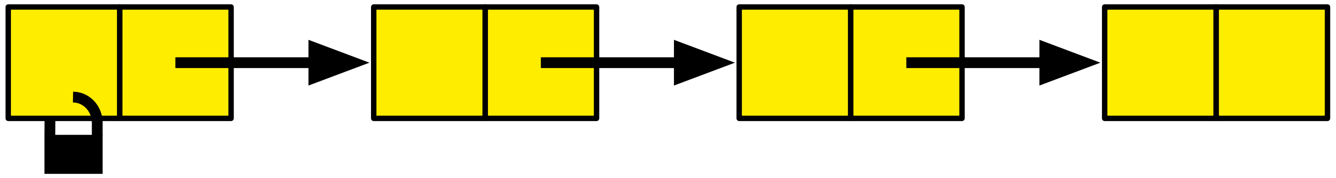
Proof of soundness removes the need for auxiliary variables.

Actions capture semantically the changes in the public interference.

Actions are more general than the algorithm: any list can be restored to an unlocked segment.

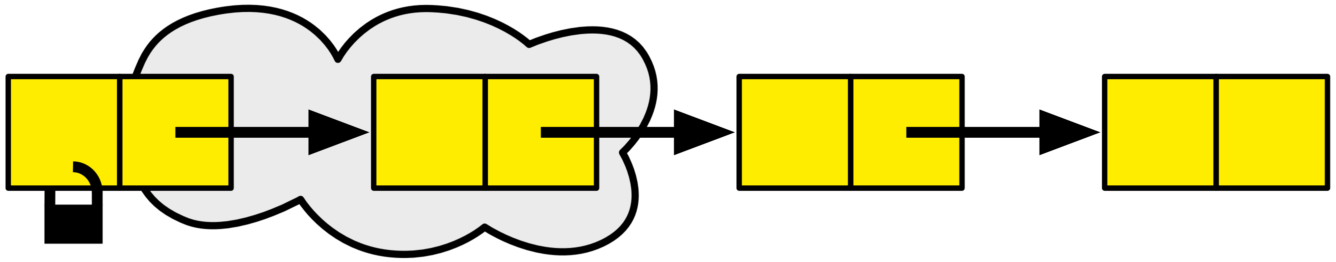
Algorithm insights

Lock extension just requires a test, not locking



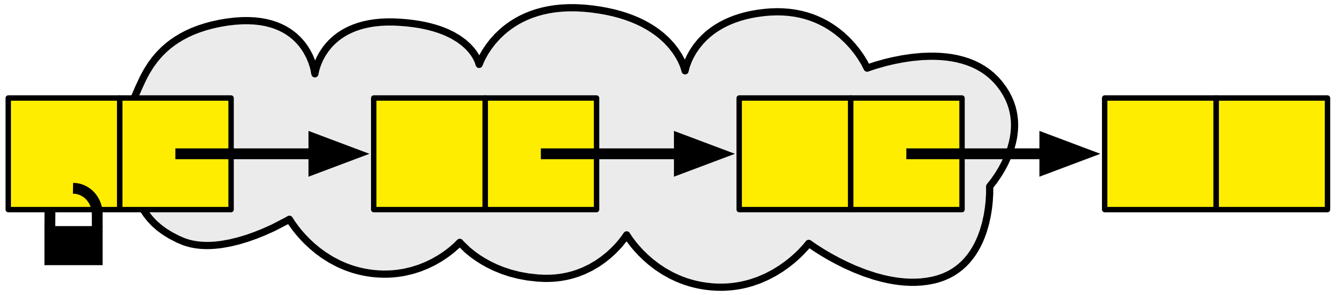
Algorithm insights

Lock extension just requires a test, not locking



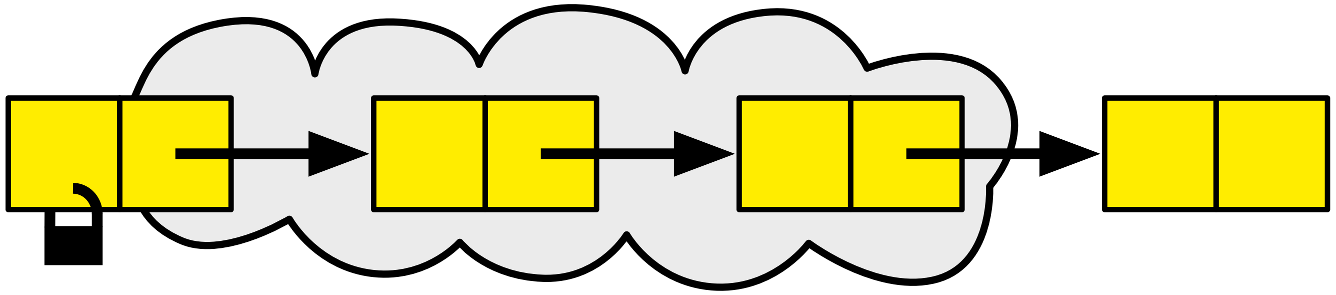
Algorithm insights

Lock extension just requires a test, not locking



Algorithm insights

Lock extension just requires a test, not locking



Consequently, we don’t need a CAS to extend.
(we do need one at the list head though)

Conclusions

Higher-order deny-guarantee removes the need for auxiliary variables in many cases.

Captures more clearly the structure of the algorithm in the proof.

Allows temporal reasoning about complex properties.

Our semantics avoids problems with recursion.

Limitations

Compositionality is difficult: environments don’t compose unless they are disjoint.

- Locality may be the answer, but we don’t know how to make this work yet.

Constructing the right set of actions is often complex.

Would like a completeness result of some kind.