Towards formal verification of imperative concurrent data structures

{Alejandro, César} Sánchez

IMDEA Software - Madrid - Spain

Workshop ReSeCo - FaMAF - Dec. 17, 2009

What are we interested in?

What are we interested in?

- Imperative programs

$$P$$

What are we interested in?

- Imperative programs
- Concurrent data structures

$$P_1 \parallel \cdots \parallel P_n$$

What are we interested in?

- Imperative programs
- Concurrent data structures

$$P_1 \parallel \cdots \parallel P_n$$

↓

data structures
(heap)

What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)

$$P_1 \parallel \cdots \parallel P_n \models \varphi$$

data structures
(heap)

What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

$$\underbrace{P_1 \parallel \cdots \parallel P_n}_{\text{data structures (heap)}} \models \varphi$$

data structures
(heap)

What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

$$\underbrace{P_1 \parallel \cdots \parallel P_n}_{\text{data structures (heap)}} \models \varphi \;\}\; \text{LTL } (\Box, \Diamond, \bigcirc, \mathcal{U})$$

data structures
(heap)

What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

$$\underbrace{P_1 \parallel \cdots \parallel P_n}_{\downarrow} \models \varphi \} \text{ LTL } (\Box, \Diamond, \bigcirc, \mathcal{U})$$

Regional Logic { data structures (heap)

What are we interested in?

- Imperative programs
- Concurrent data structures
- Temporal property (safety, liveness)
- Formal verification

$$P_1 \parallel \cdots \parallel P_n \models \varphi$$

$\}$ LTL ($\square, \diamond, \bigcirc, \mathcal{U}$)

$P_1 \parallel \cdots \parallel P_n$ → Regional Logic { data structures (heap)

$\models$ → Verification Diagram

Reasoning about the heap

Reasoning about the heap

- Separation Logic
    - Hoare logic extension to reason about shared mutable data structure

Reasoning about the heap

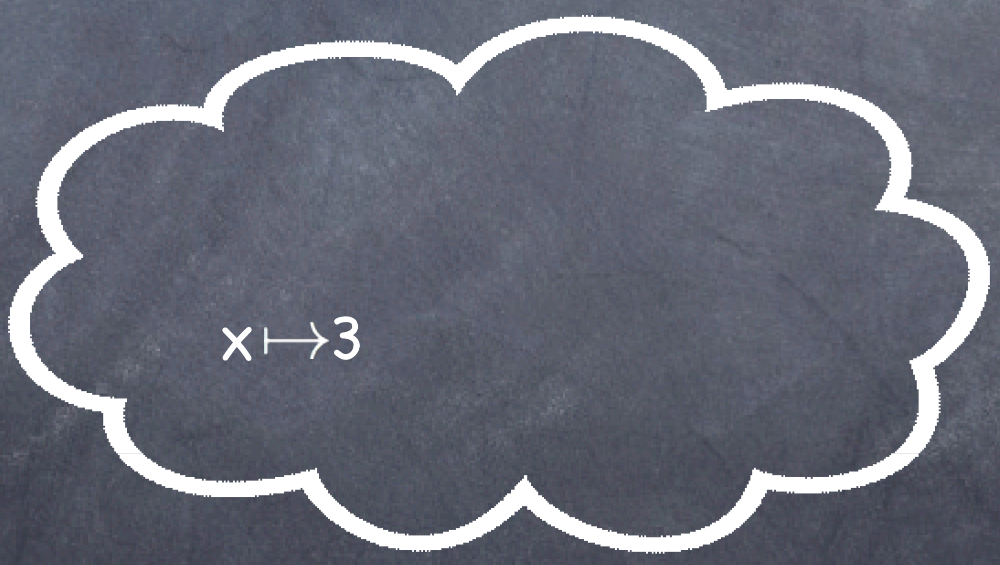
- Separation Logic
  - Hoare logic extension to reason about shared mutable data structure
  - emp

Reasoning about the heap

- Separation Logic

  Hoare logic extension to reason about shared mutable data structure

  emp , $\mapsto$



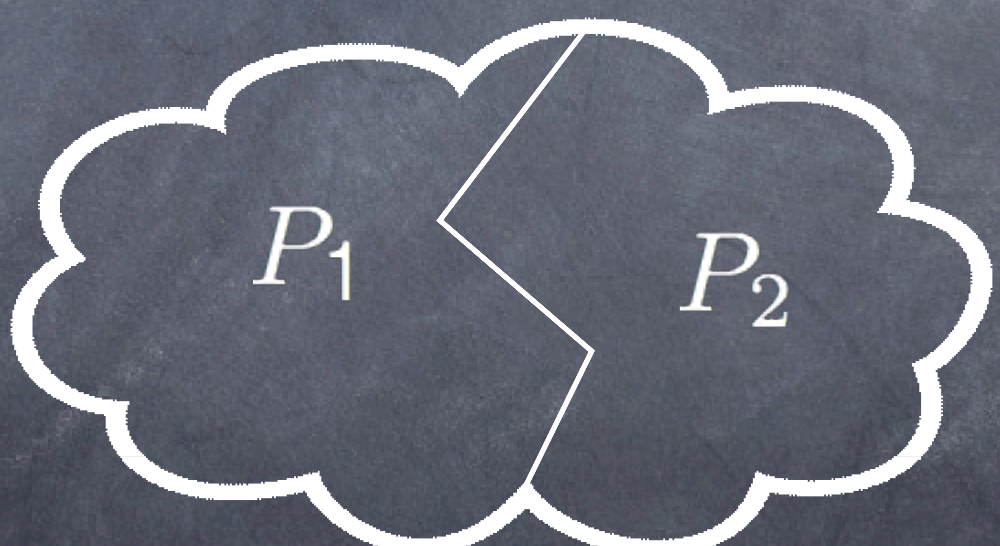
Reasoning about the heap

- Separation Logic
  - Hoare logic extension to reason about shared mutable data structure
  - emp , $\mapsto$, $*$

$$[P_0 * P_1]\, s\, h \Leftrightarrow \exists h_0, h_1 \bullet h_0 \perp h_1 \wedge h_0.h_1 = h \wedge [P_0]\, s\, h_0 \wedge [P_1]\, s\, h_1$$

Reasoning about the heap

- Separation Logic
  - Hoare logic extension to reason about shared mutable data structure
  - emp , $\mapsto$, $*$



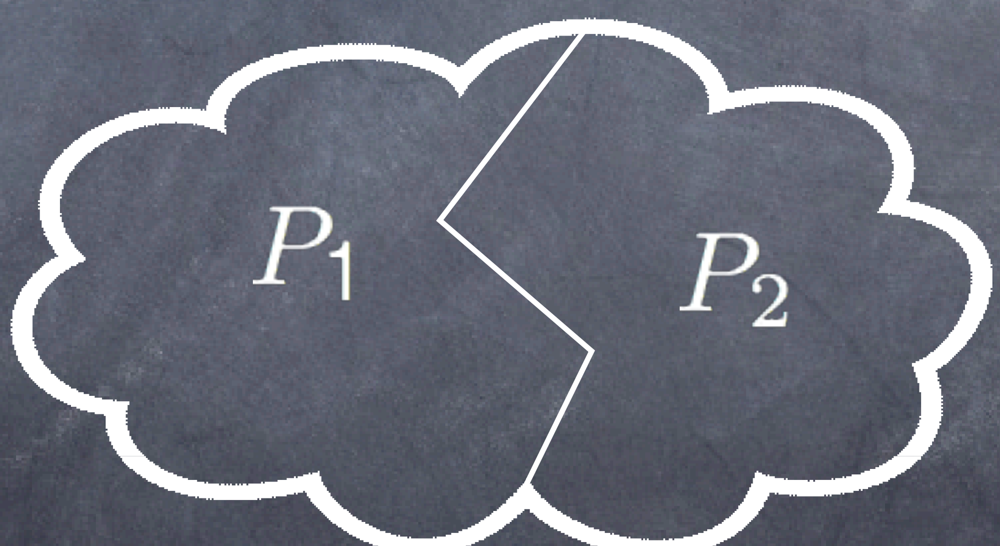
$$[P_0 * P_1]\, s\, h \;\Leftrightarrow\; \exists h_0, h_1 \bullet h_0 \perp h_1 \wedge h_0.h_1 = h \wedge [P_0]\, s\, h_0 \wedge [P_1]\, s\, h_1$$

Reasoning about the heap

- Separation Logic

  Hoare logic extension to reason about shared mutable data structure

  emp , $\mapsto$, $*$, $-\!*$



$$[P_0 * P_1]\, s\, h \;\Leftrightarrow\; \exists h_0, h_1 \bullet h_0 \perp h_1 \wedge h_0.h_1 = h \wedge [P_0]\, s\, h_0 \wedge [P_1]\, s\, h_1$$

Reasoning about the heap

- Regional Logic

Reasoning about the heap

- Regional Logic
  - Classical first order logic

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables
  - Region manipulation language: emp, $\langle\ \rangle$, $\cup$, $\cap$, -

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables
  - Region manipulation language: emp, $\langle\ \rangle$, $\cup$, $\cap$, -
  - Region assertion language: $R_1 \subseteq R_2$

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables
  - Region manipulation language: emp, $\langle\ \rangle$, $\cup$, $\cap$, -
  - Region assertion language: $R_1 \subseteq R_2, R_1 \# R_2$

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables
  - Region manipulation language: emp, $\langle\ \rangle$, $\cup$, $\cap$, -
  - Region assertion language: $R_1 \subseteq R_2, R_1 \# R_2, R_1.f \subseteq R_2$

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables
  - Region manipulation language: emp, $\langle\,\rangle$, $\cup$, $\cap$, -
  - Region assertion language: $R_1 \subseteq R_2$, $R_1 \# R_2$, $R_1.f \subseteq R_2$, $R_1.f \# R_2$

Reasoning about the heap

- Regional Logic
  - Classical first order logic
  - Based on Hoare logic
  - Ghost fields/variables
  - Region manipulation language: emp, $\langle\ \rangle$, $\cup$, $\cap$, -
  - Region assertion language: $R_1 \subseteq R_2, R_1 \# R_2, R_1.f \subseteq R_2, R_1.f \# R_2$
    $\forall x : K \in R \mid P$

Verification Diagrams

Verification Diagrams

$$P \models \varphi$$

Verification Diagrams

$$\begin{array}{c} P \models \varphi \\ \downarrow \\ \Psi \end{array}$$

Verification Diagrams

- Representation of a system by FTS

$$P \models \varphi$$

$$\downarrow$$

$$\Psi$$

Verification Diagrams

- Representation of a system by FTS
- Sound & complete

$$P \models \varphi$$
$$\downarrow$$
$$\Psi$$

Verification Diagrams

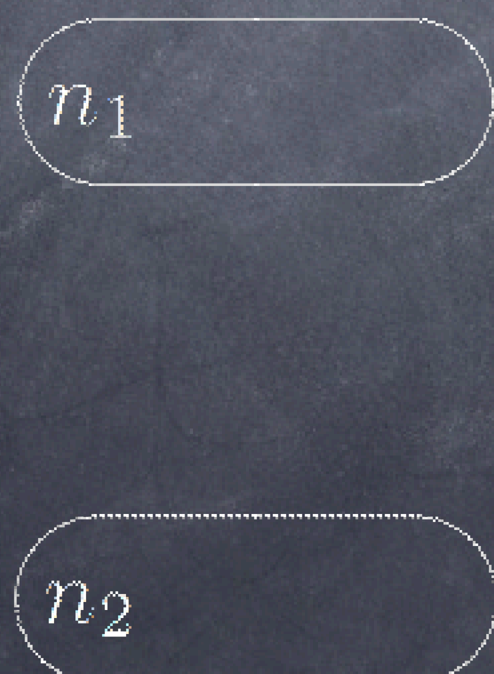
- Representation of a system by FTS
- Sound & complete

$$\Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$

Verification Diagrams

- Representation of a system by FTS
- Sound & complete

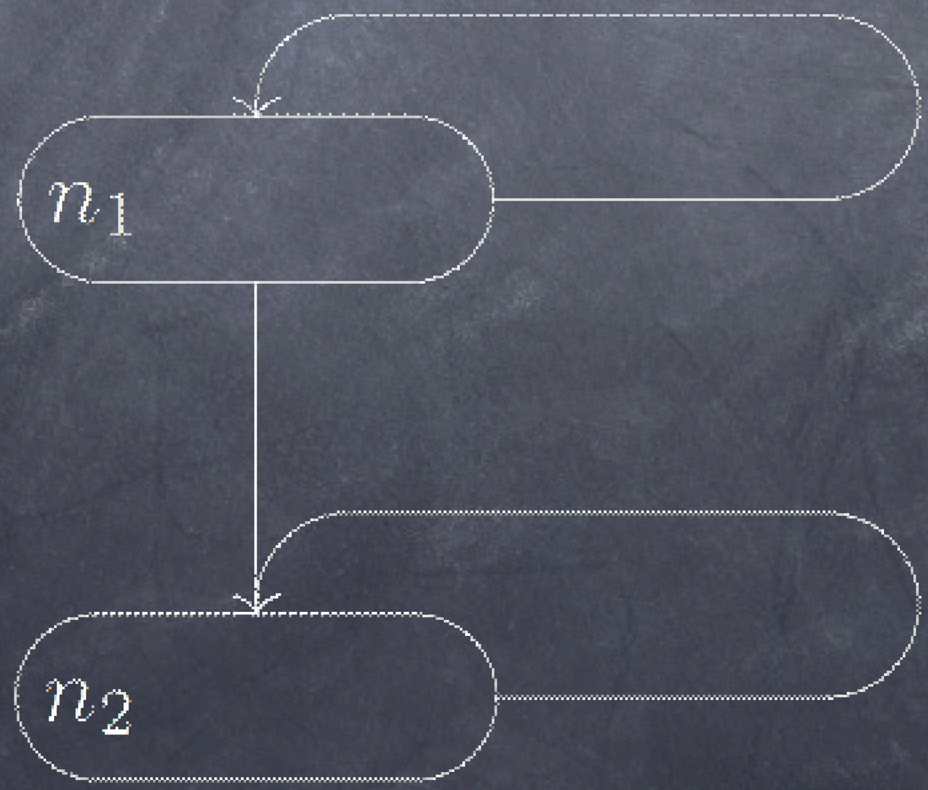
$$\Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Verification Diagrams

- Representation of a system by FTS
- Sound & complete

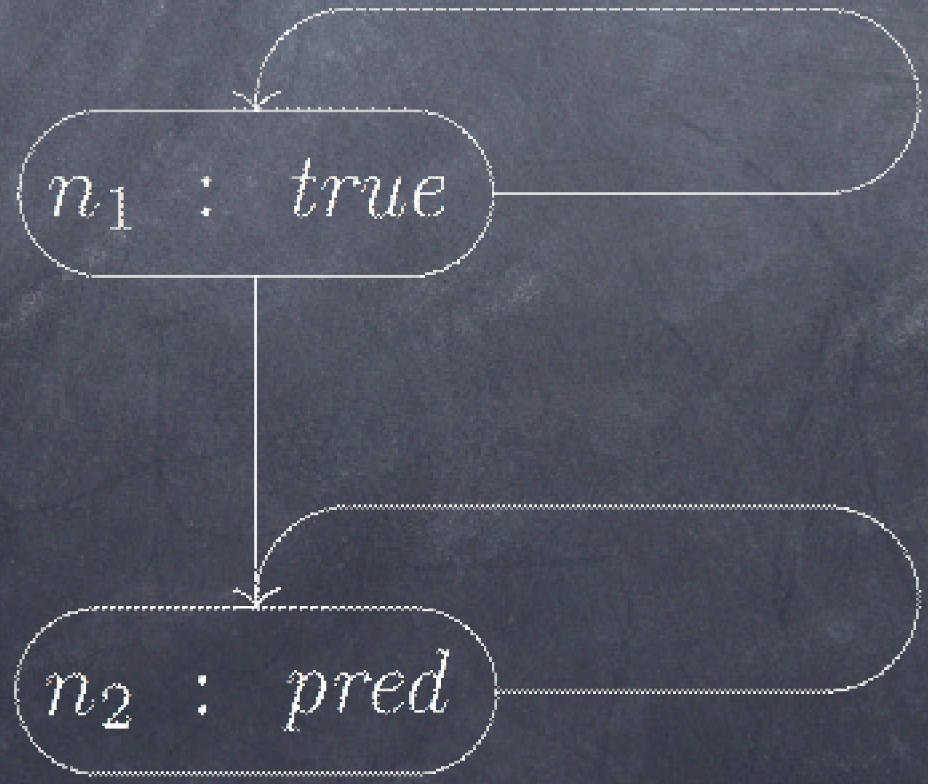
$$\Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Verification Diagrams

- Representation of a system by FTS
- Sound & complete

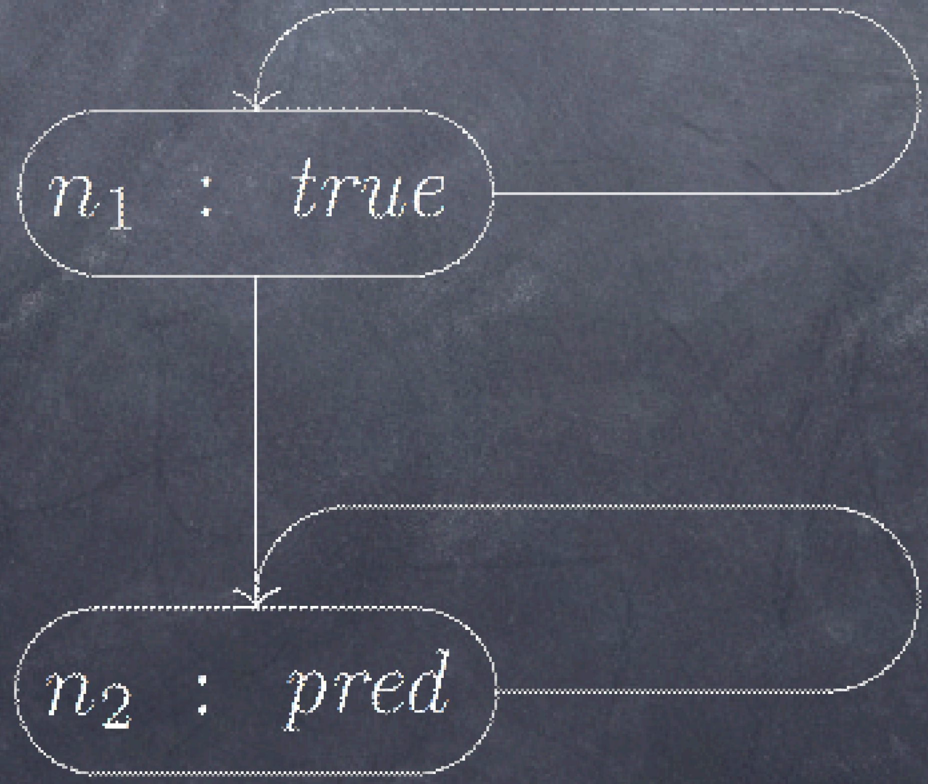
$$\Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Verification Diagrams

- Representation of a system by FTS
- Sound & complete

$$\Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$

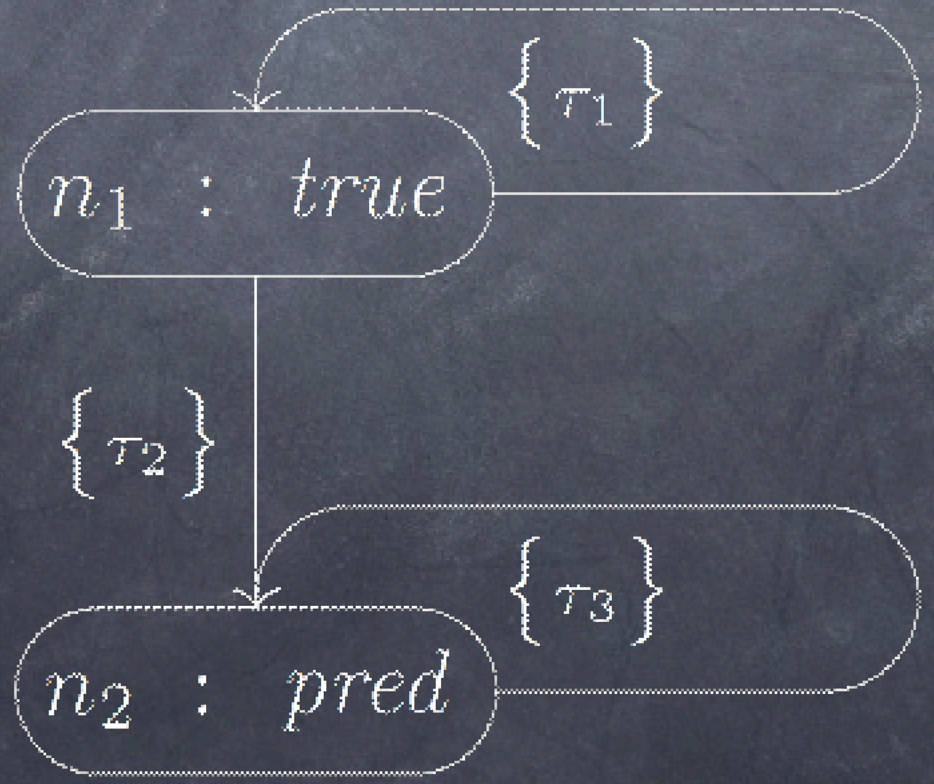


$$\Diamond \Box pred$$

Verification Diagrams

- Representation of a system by FTS
- Sound & complete

$$\Psi = \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Main Idea

Main Idea

Concurrent Data Structure

Main Idea

Concurrent Data Structure

φ

Main Idea

Concurrent Data Structure

Most General Client [N]

(extended with GV)

φ

Main Idea

Concurrent Data Structure

Most General Client [N] Ψ φ

(extended with GV)

Main Idea

Concurrent Data Structure

Most General Client [N] Ψ φ

(extended with GV)

- Verification conditions like: initialization, consecution, acceptance, fairness, satisfaction...

Main Idea

Concurrent Data Structure

Most General Client [N] $\longleftrightarrow$ Ψ $\longleftrightarrow$ φ

(extended with GV)

- Verification conditions like: initialization, consecution, acceptance, fairness, satisfaction...

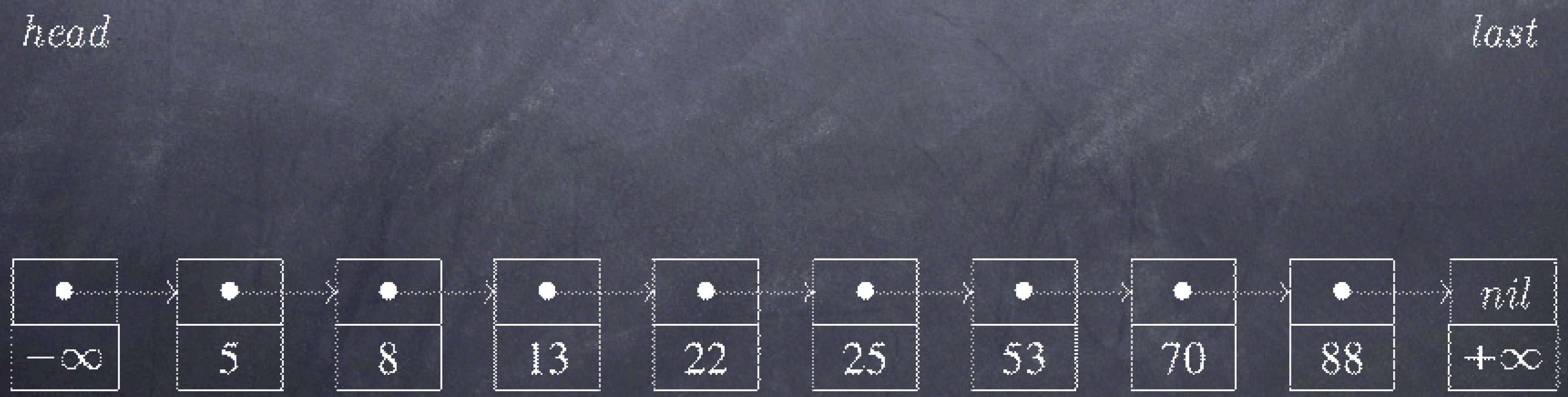
Skiplists

Skiplists

- Sorted list of elements

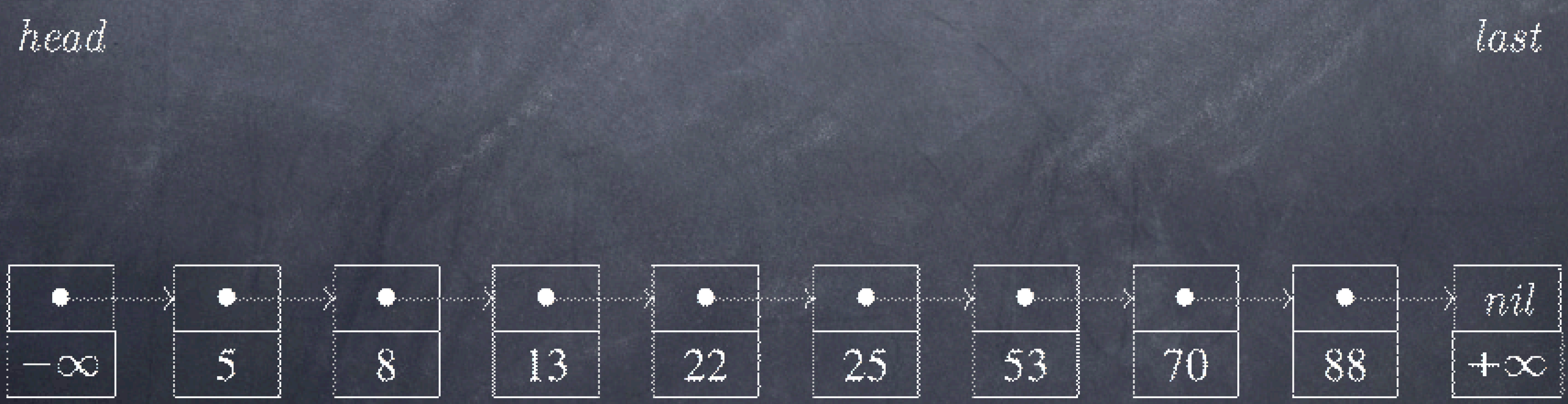
Skiplists

- Sorted list of elements



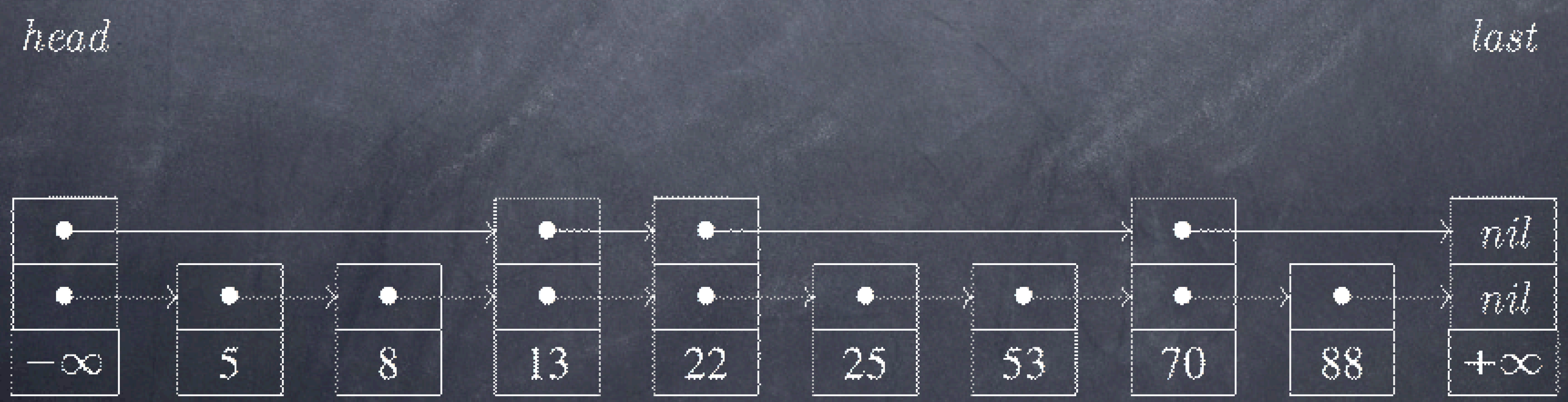
Skiplists

- Sorted list of elements
- Hierarchy of linked lists



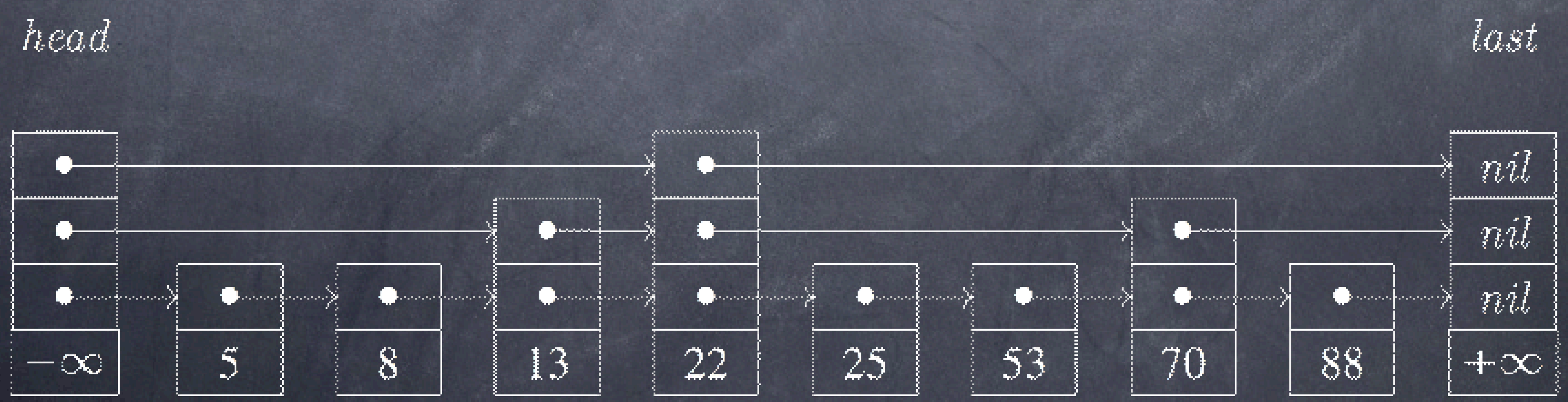
Skiplists

- Sorted list of elements
- Hierarchy of linked lists



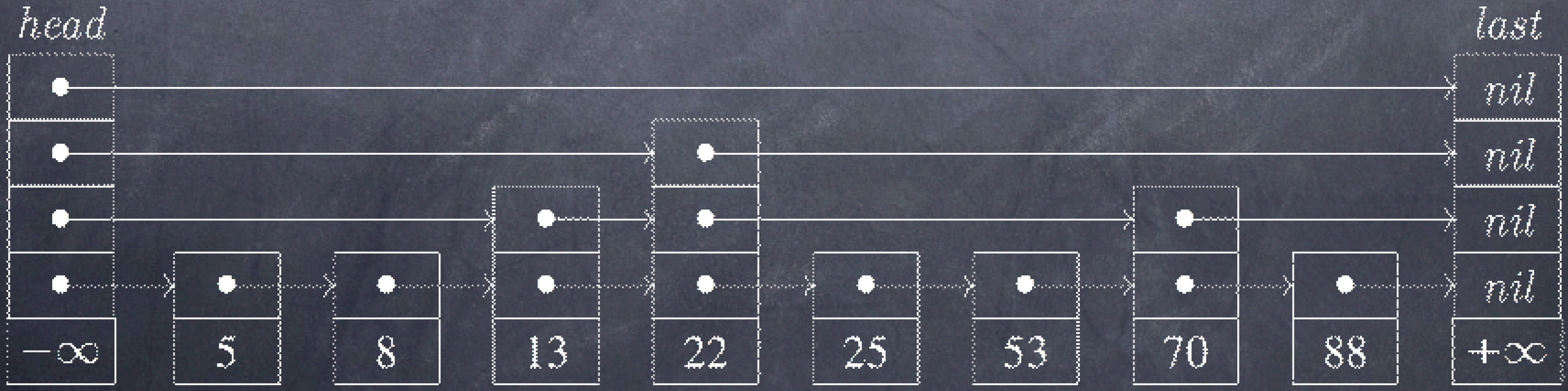
Skiplists

- Sorted list of elements
- Hierarchy of linked lists



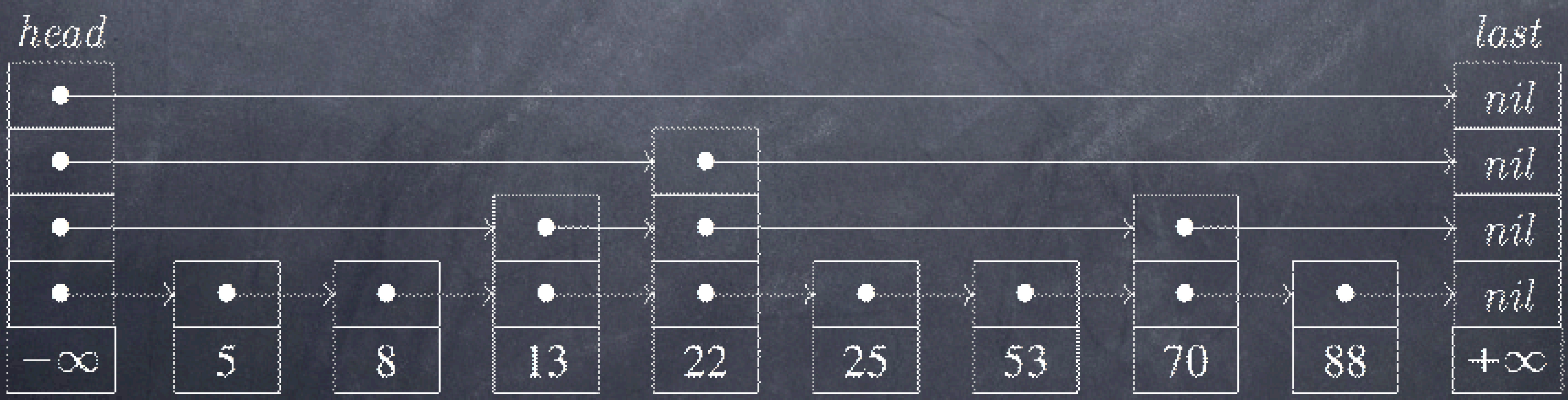
Skiplists

- Sorted list of elements
- Hierarchy of linked lists



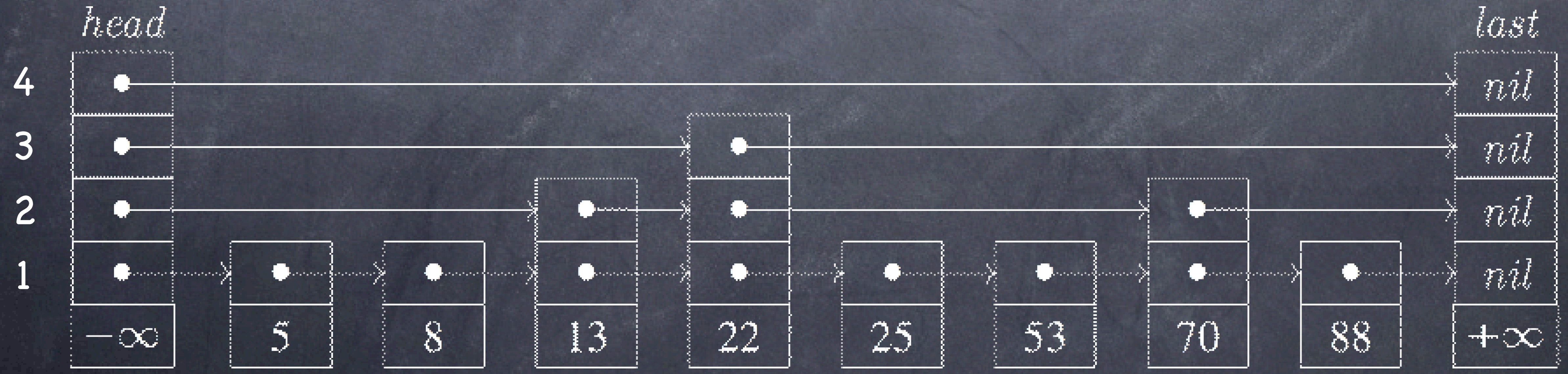
Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees



Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees



Fine-grained lock-coupling concurrent skiplists

Fine-grained lock-coupling concurrent skiplists

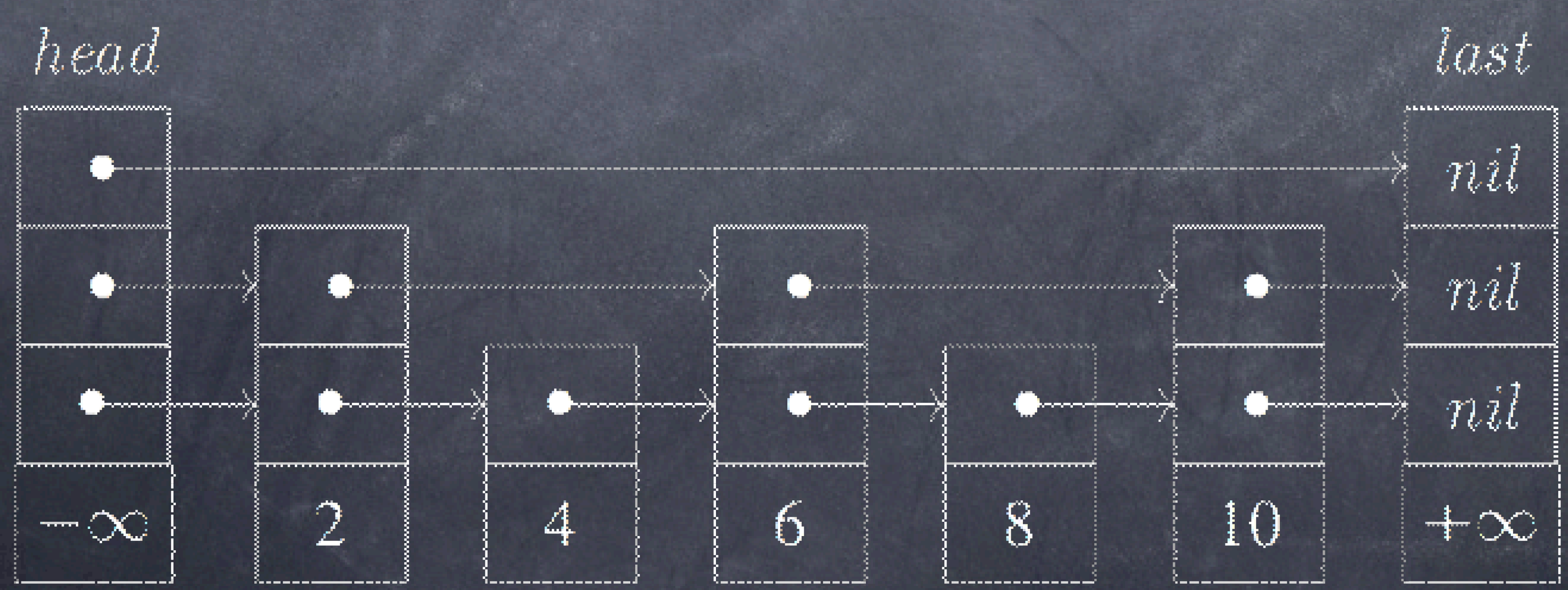
- Reduce granularity of locks

Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

Fine-grained lock-coupling concurrent skiplists

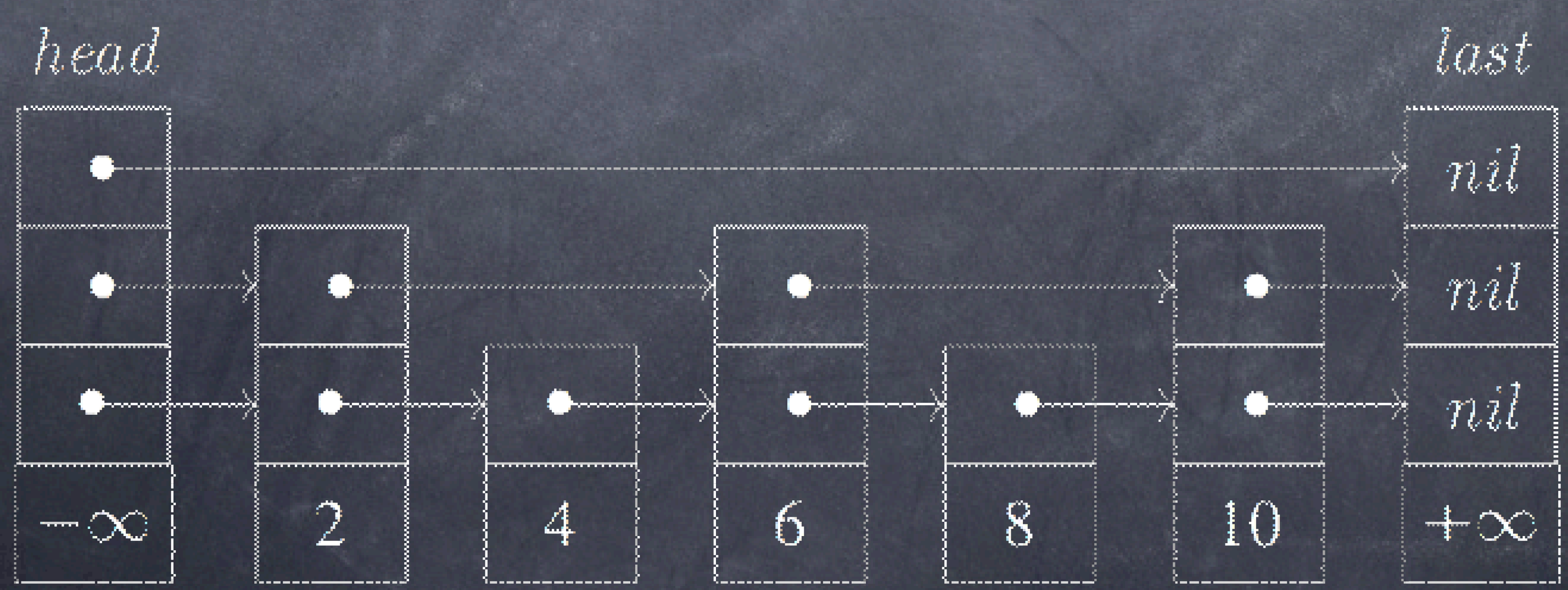
- Reduce granularity of locks
- Locks acquired and released in climbing fashion



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

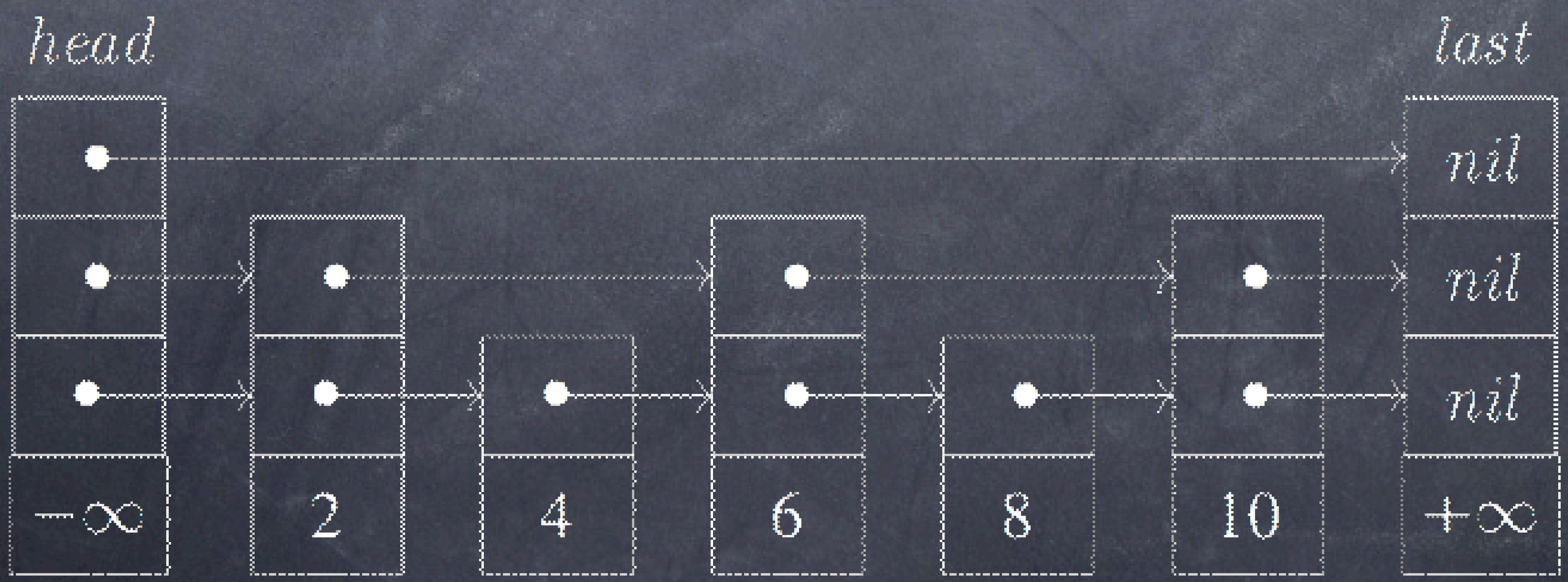
$insert^{[k]}(9)$



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

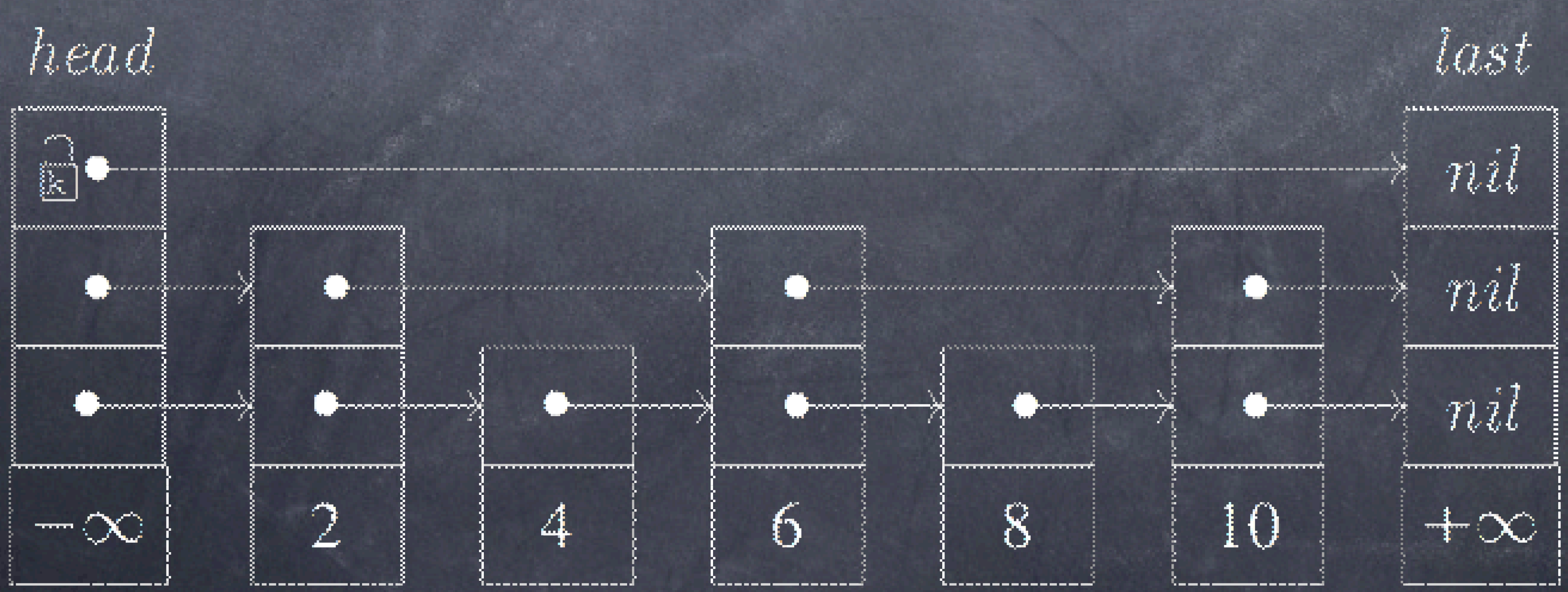
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

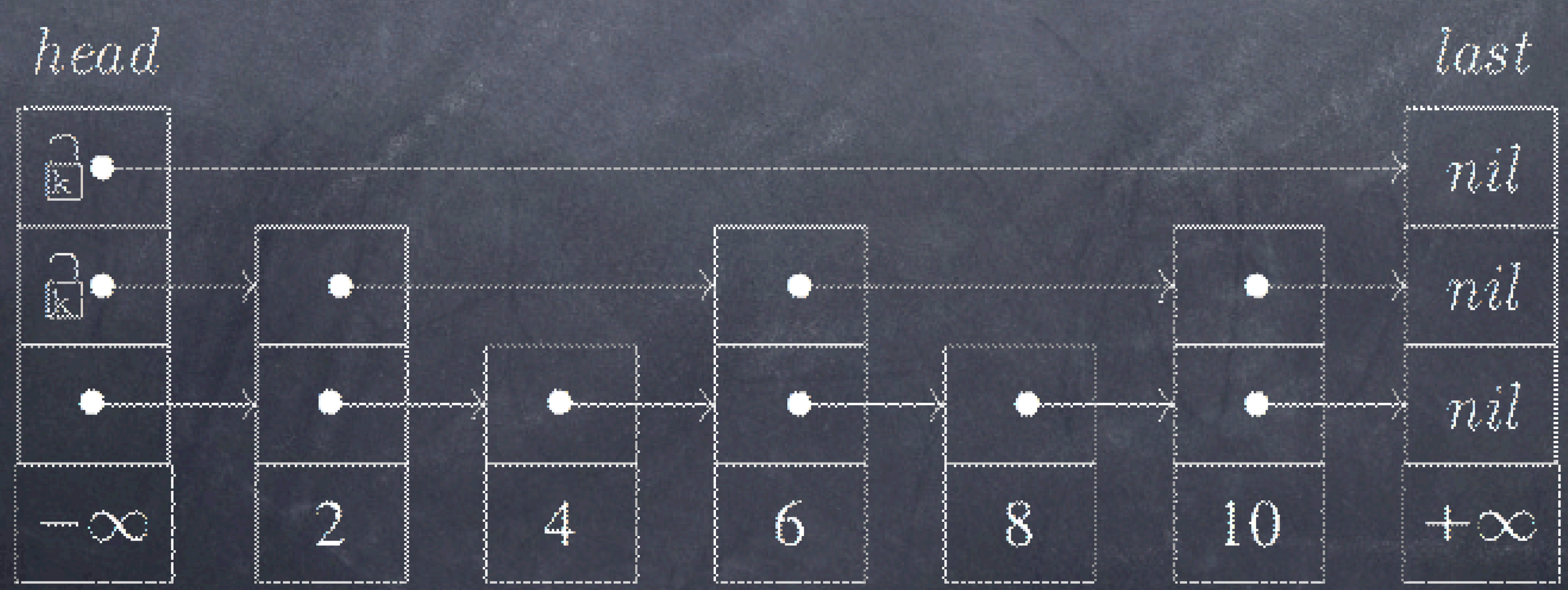
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

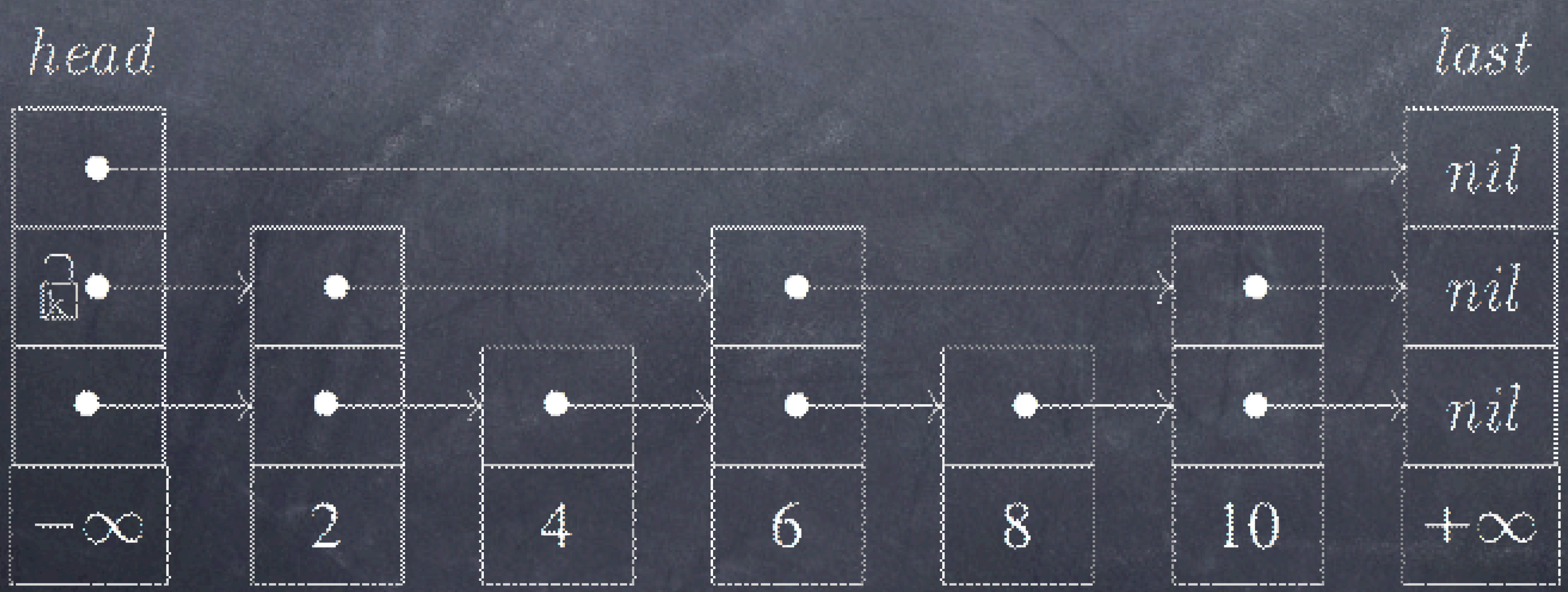
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

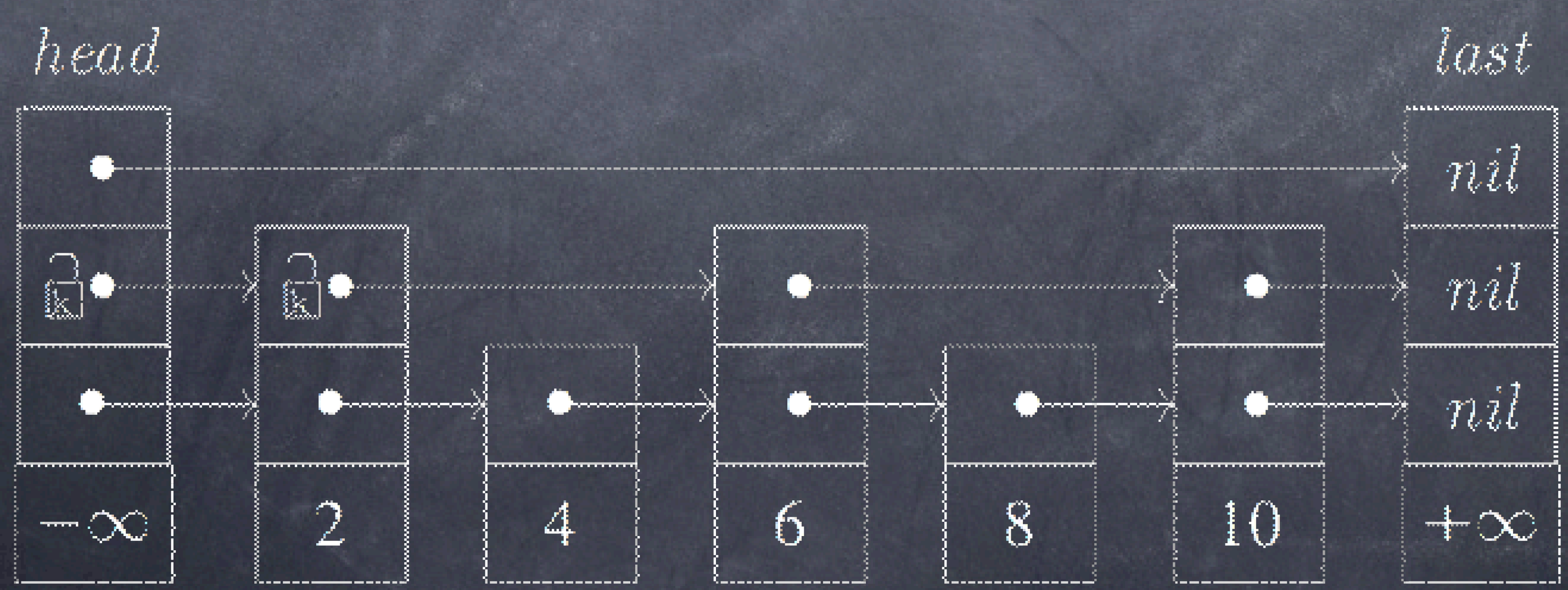
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

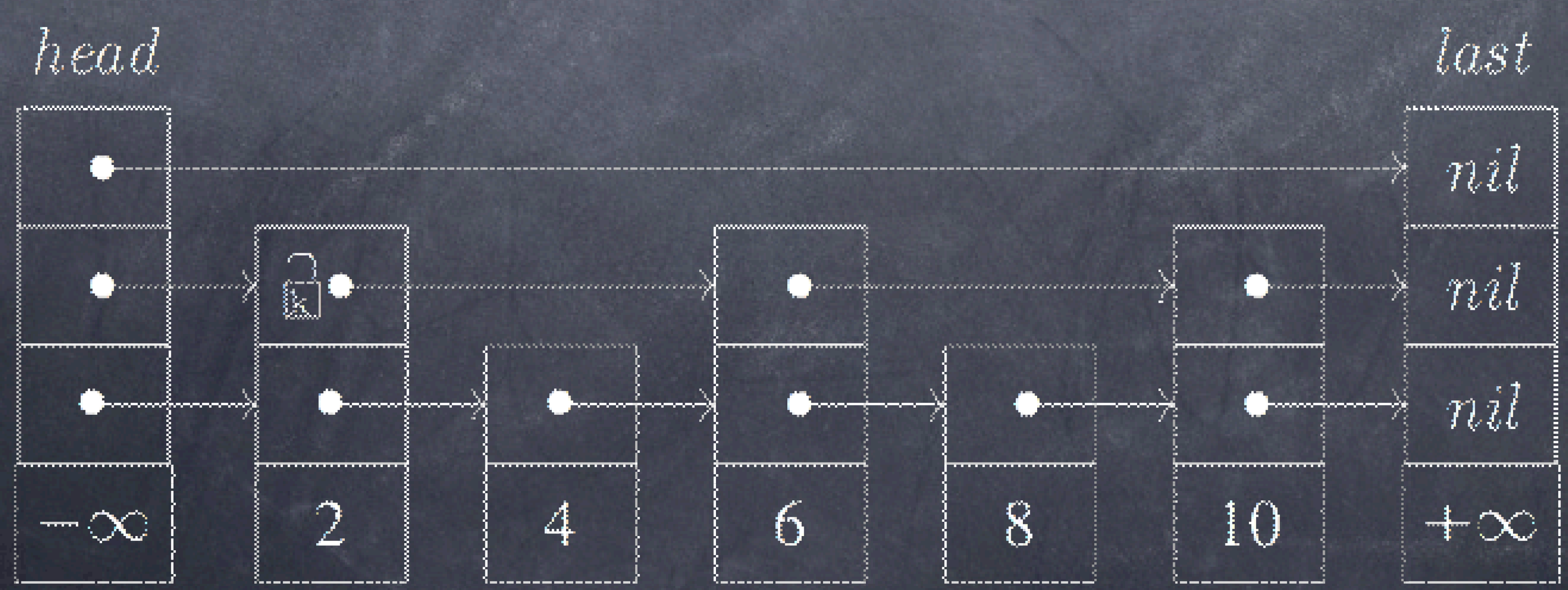
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

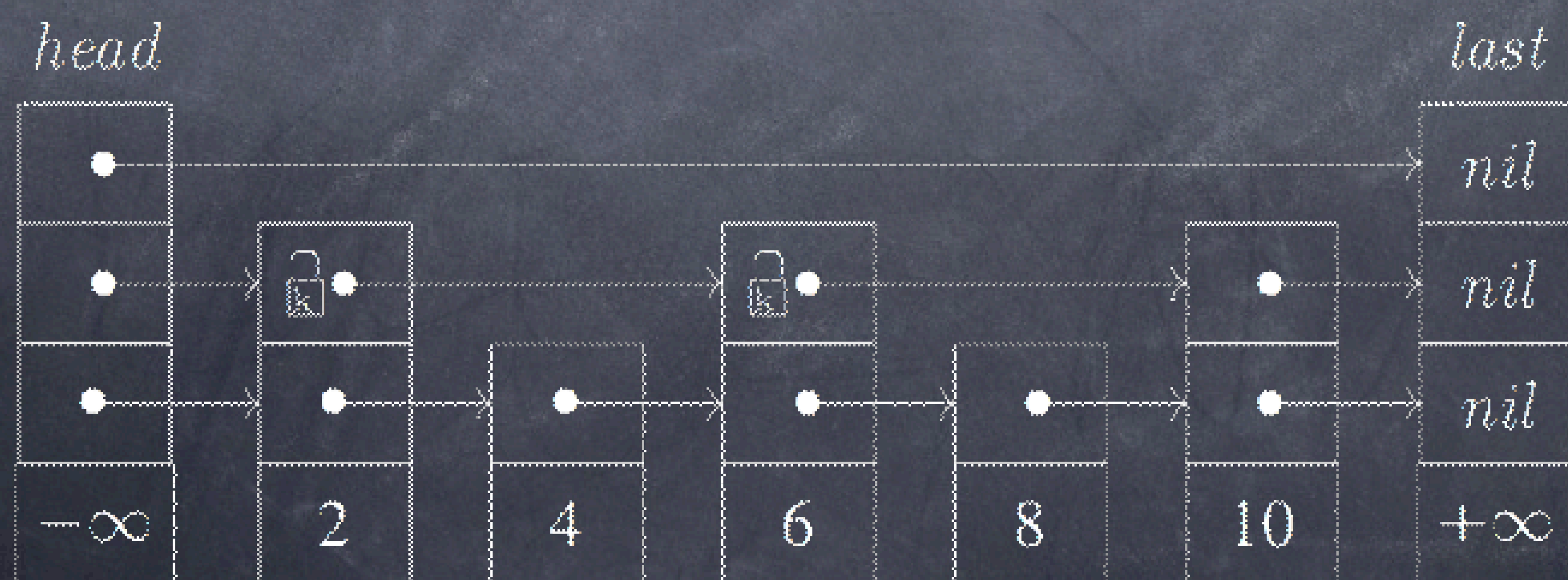
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

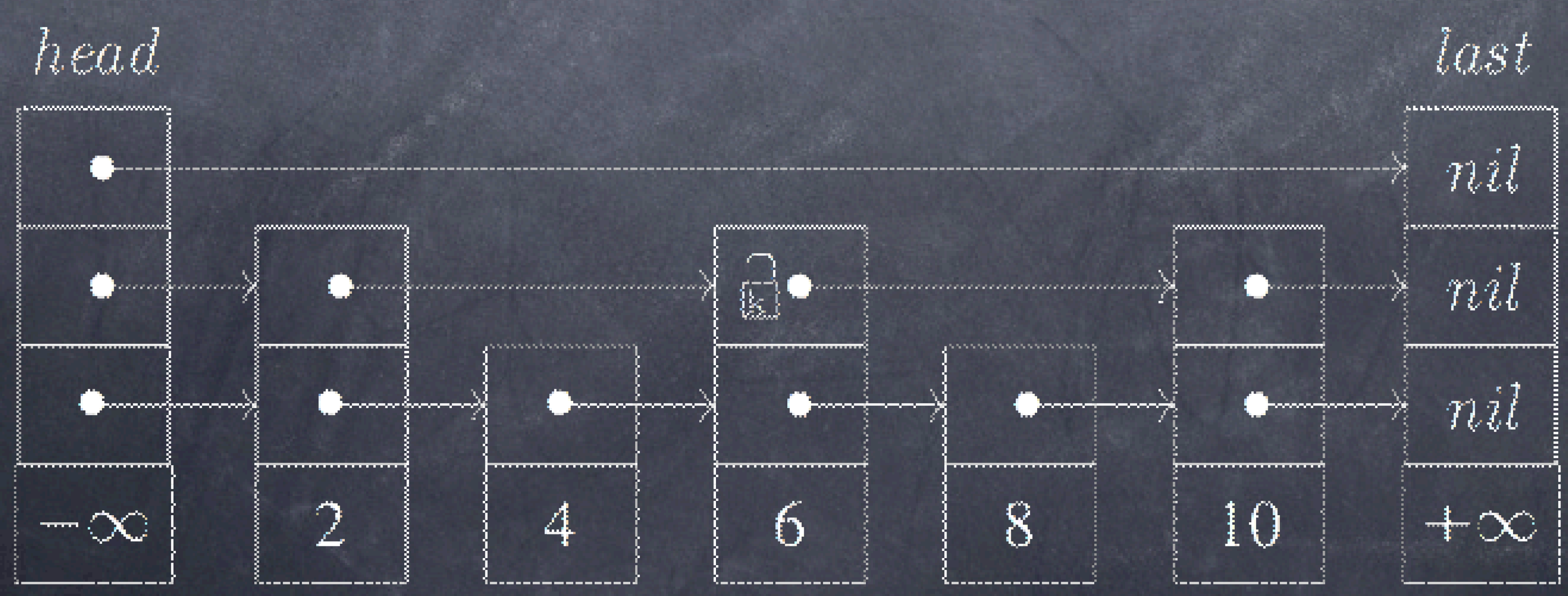
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

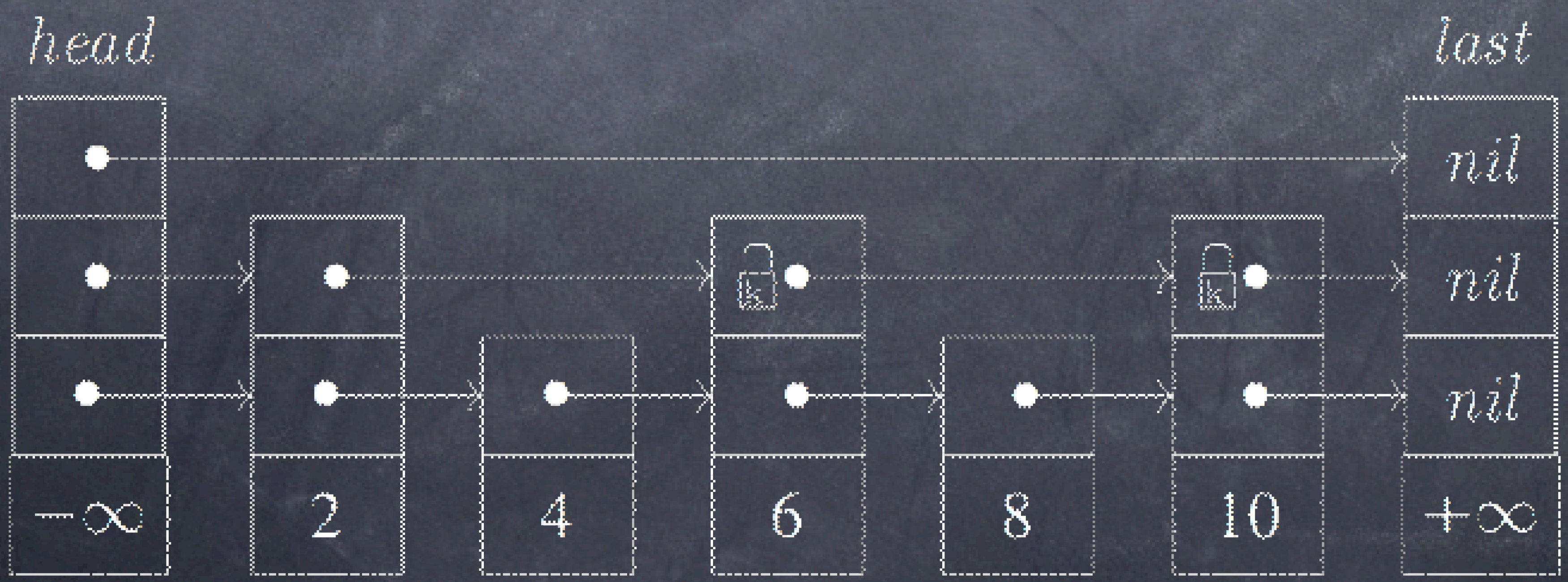
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

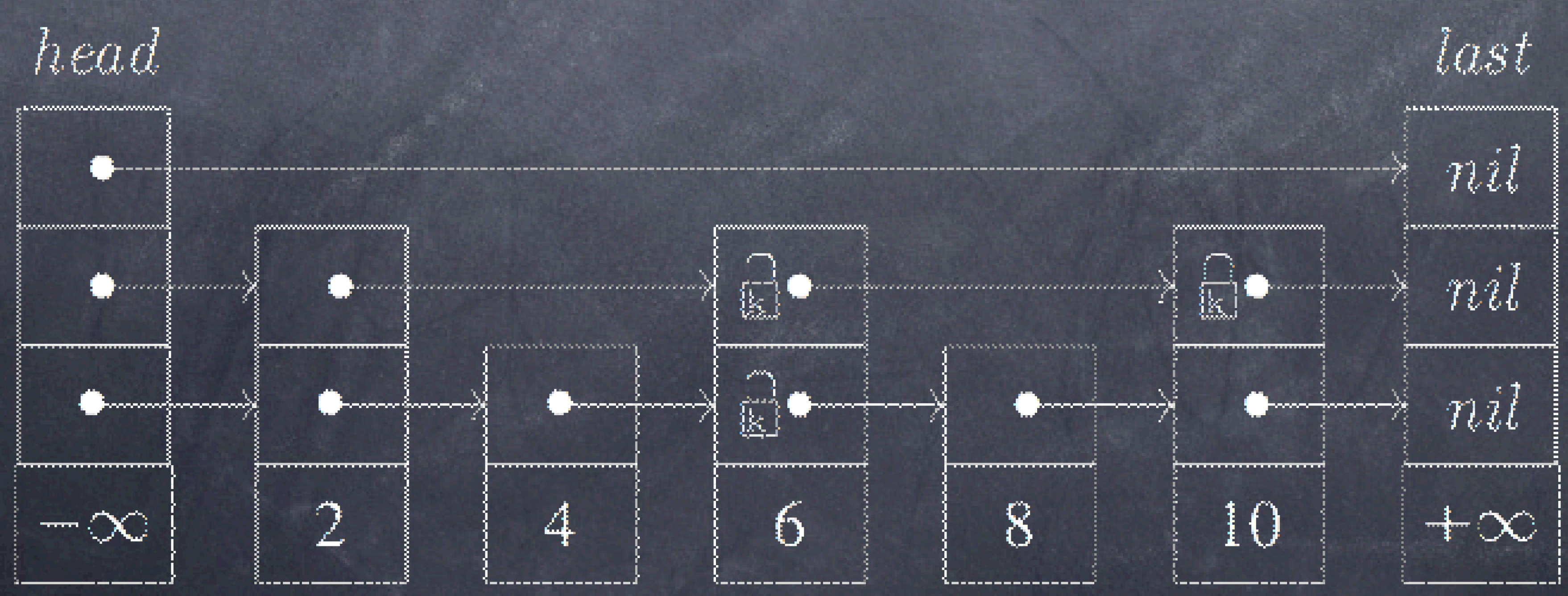
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

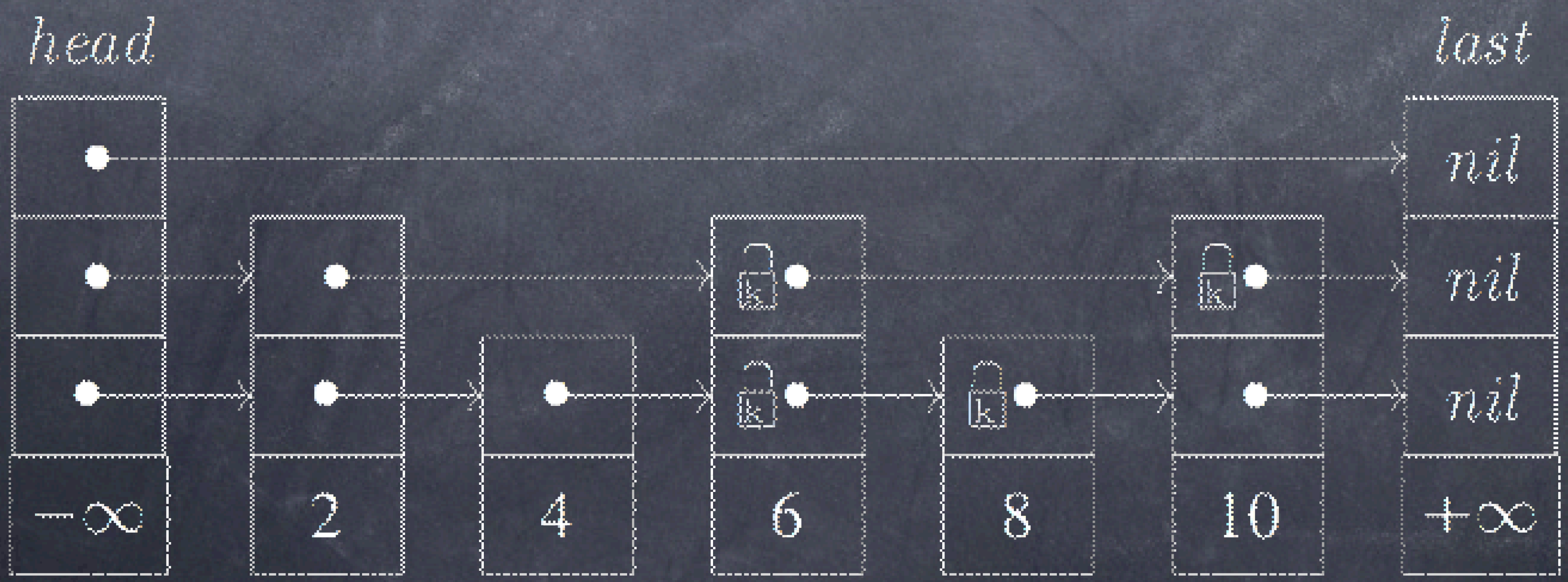
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

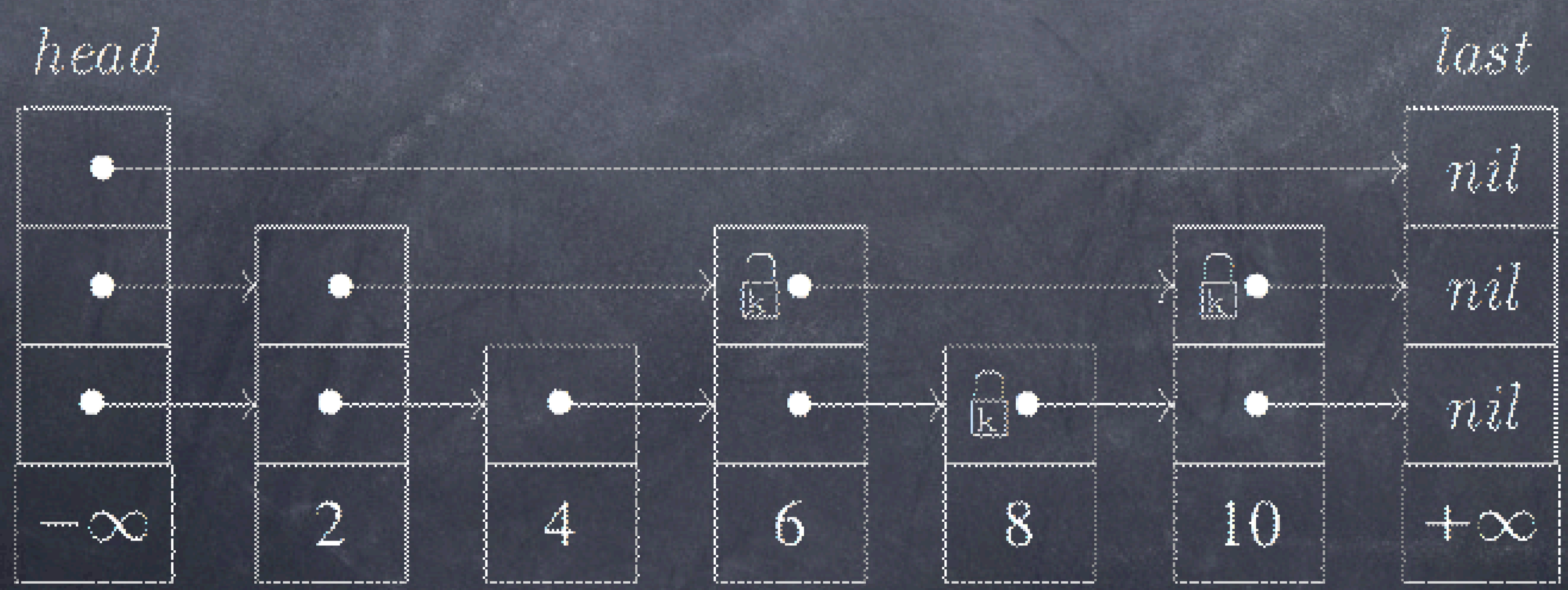
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

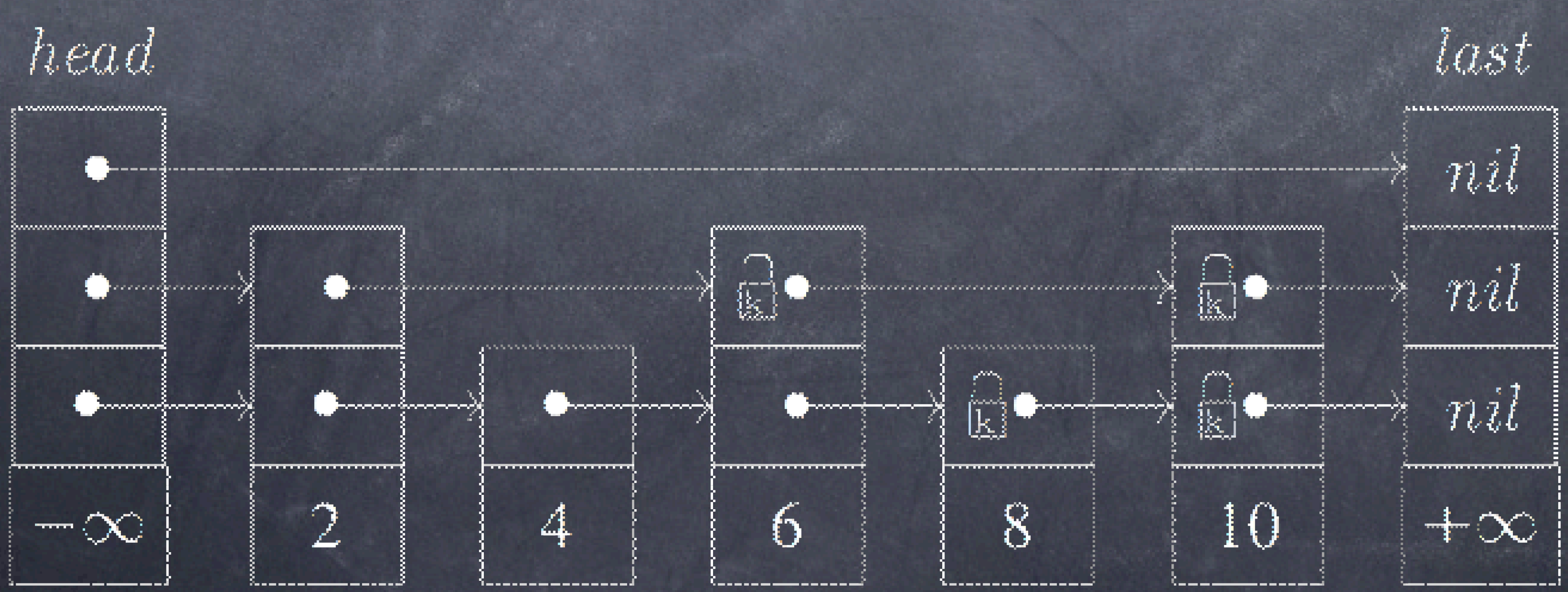
$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling concurrent skiplists

- Reduce granularity of locks
- Locks acquired and released in climbing fashion

$insert^{[k]}(9)$ level = 2



Fine-grained lock-coupling CSL

Fine-grained lock-coupling CSL

$insert(sl, v)$

Algorithm 4 Insertion on a lock-coupling concurrent skiplist

```
procedure INSERT(SkipList sl, Key k, Value newval)
    Vector < Node* > update[1..sl.maxLevel]    //@ mrgn m_r := emp
    lvl := randomLevel()
    Node * pred := sl.head
    pred.locks[lvl].lock()                      //@ m_r := m_r ∪ (pred.forward[lvl])
    Node * curr := pred.forward[lvl]
    curr.locks[lvl].lock()                      //@ m_r := m_r ∪ (curr.forward[lvl])
    for i := lvl downto 1 do
        if i < lvl then
            pred.locks[i].lock()                //@ m_r := m_r ∪ (pred.forward[i])
            curr := pred.forward[i]
            curr.locks[i].lock()                //@ m_r := m_r ∪ (curr.forward[i])
        end if
        while curr.key < k do
            pred.locks[i].unlock()              //@ m_r := m_r − (pred.forward[i])
            pred := curr
            curr := pred.forward[i]
            curr.locks[i].lock()                //@ m_r := m_r ∪ (curr.forward[i])
        end while
        update[i] := pred
    end for
    if curr.key = k then
        curr.val = newval
        for i := 1 to lvl do
            update[i].forward[i].locks[i].unlock()
                              //@ m_r := m_r − (update[i].forward[i].forward[i])
            update[i].locks[i].unlock()         //@ m_r := m_r − (update[i].forward[i])
        end for
    else
        x := CreateNode(lvl, k, newval)
        for i := 1 to lvl do
            x.forward[i] := update[i].forward[i]
            update[i].forward[i] := x           //@ sl.r := sl.r ∪ (x)
            x.forward[i].locks[i].unlock() //@ m_r := m_r − (x.forward[i].forward[i])
            update[i].locks[i].unlock()         //@ m_r := m_r − (update[i].forward[i])
        end for
    end if
end procedure
```

Fine-grained lock-coupling CSL

$insert(sl, v)$

Algorithm 4 Insertion on a lock-coupling concurrent skiplist

```
procedure INSERT(SkipList sl, Key k, Value newval)
    Vector < Node* > update[1..sl.maxLevel]     //@ mrgn m_r := emp
    lvl := randomLevel()
    Node * pred := sl.head
    pred.locks[lvl].lock()                      //@ m_r := m_r ∪ (pred, forward[lvl])
    Node * curr := pred.forward[lvl]
    curr.locks[lvl].lock()                      //@ m_r := m_r ∪ (curr, forward[lvl])
    for i := lvl downto 1 do
        if i < lvl then
            pred.locks[i].lock()                //@ m_r := m_r ∪ (pred, forward[i])
            curr := pred.forward[i]
            curr.locks[i].lock()                //@ m_r := m_r ∪ (curr, forward[i])
        end if
        while curr.key < k do
            pred.locks[i].unlock()              //@ m_r := m_r − (pred, forward[i])
            pred := curr
            curr := pred.forward[i]
            curr.locks[i].lock()                //@ m_r := m_r ∪ (curr, forward[i])
        end while
        update[i] := pred
    end for
```

```
while curr.key < k do
    pred.locks[i].unlock()          //@ m_r := m_r − (pred, forward[i])
    pred := curr
    curr := pred.forward[i]
    curr.locks[i].lock()            //@ m_r := m_r ∪ (curr, forward[i])
end while
```

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v)$

while $curr.key < k$ **do**
$\quad pred.locks[i].unlock()$ $\qquad //@\ m_r := m_r - (pred, forward[i])$
$\quad pred := curr$
$\quad curr := pred.forward[i]$
$\quad curr.locks[i].lock()$ $\qquad //@\ m_r := m_r \cup (curr, forward[i])$
end while

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v) \quad decide(sl)$

while $curr.key < k$ **do**

$\quad pred.locks[i].unlock() \qquad //@\ m_r := m_r - (pred, forward[i])$

$\quad pred := curr$

$\quad curr := pred.forward[i]$

$\quad curr.locks[i].lock() \qquad //@\ m_r := m_r \cup (curr, forward[i])$

end while

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v) \quad decide(sl)$

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v) \quad decide(sl)$

T_i

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v) \quad decide(sl)$

$$T_i \models \Box \varphi_{insert}(i)$$

$$\varphi_{insert}(i) \hat{=} at\_insert^{[i]}_{8..36} \rightarrow at\_insert^{[i]}_{8..36} \,\mathcal{U}\, at\_insert^{[i]}_{37}$$

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v) \quad decide(sl)$

$$\|_{j \in T_{ID} - \{i\}} T_j \parallel T_i \models \Box \varphi_{insert}(i)$$

$$\varphi_{insert}(i) \hat{=} at\_insert^{[i]}_{8..36} \rightarrow at\_insert^{[i]}_{8..36} \, \mathcal{U} \, at\_insert^{[i]}_{37}$$

Fine-grained lock-coupling CSL

$insert(sl, v) \quad search(sl, v) \quad remove(sl, v) \quad decide(sl)$

$$\|_{j \in T_{ID} - \{i\}} T_j \parallel T_i \models \Box \varphi_{insert}(i)$$

$$\updownarrow$$

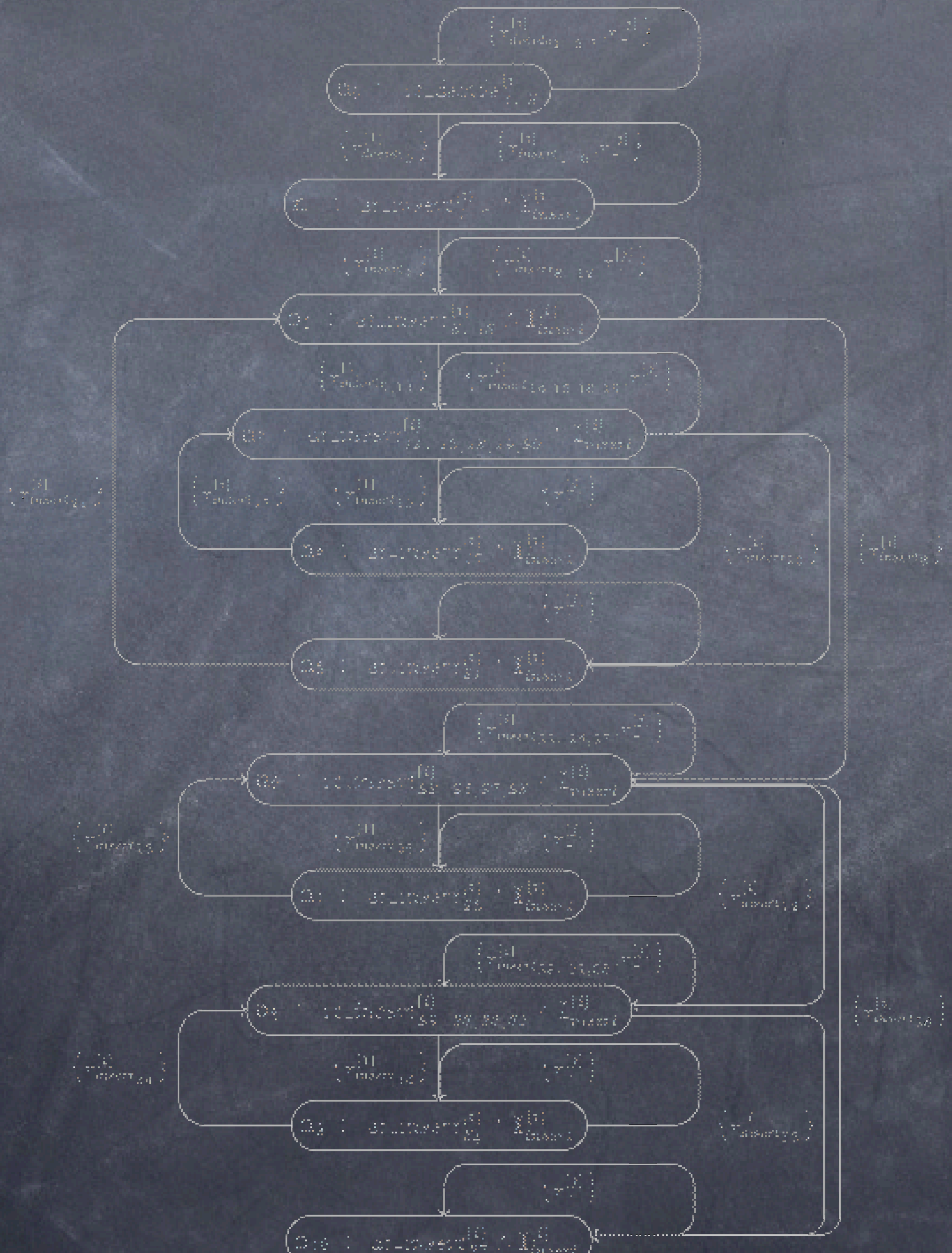
$$\Psi$$

$$\updownarrow$$

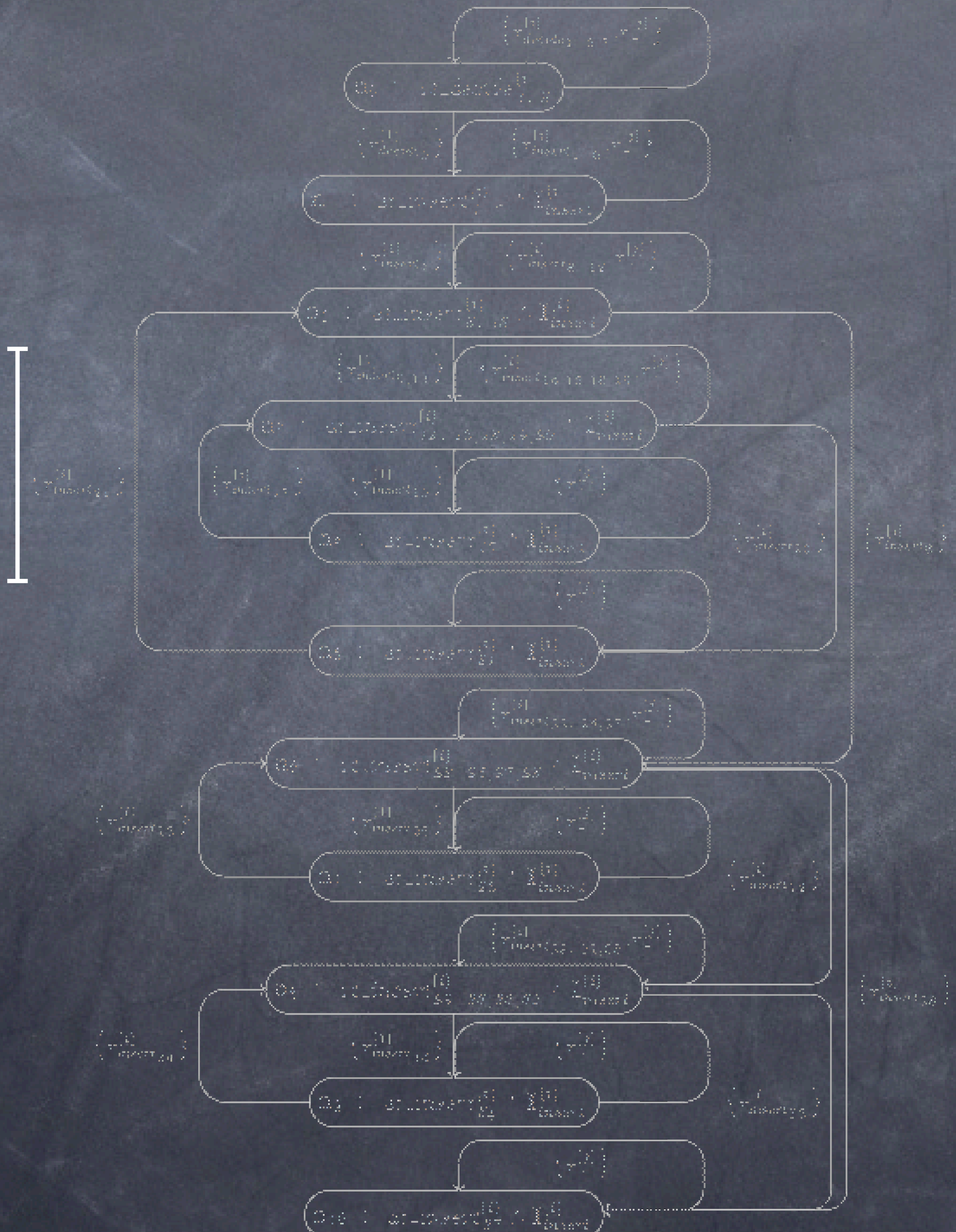
$$\varphi_{insert}(i) \hat{=} at\_insert^{[i]}_{8..36} \rightarrow at\_insert^{[i]}_{8..36} \, \mathcal{U} \, at\_insert^{[i]}_{37}$$

Fine-grained lock-coupling CSL

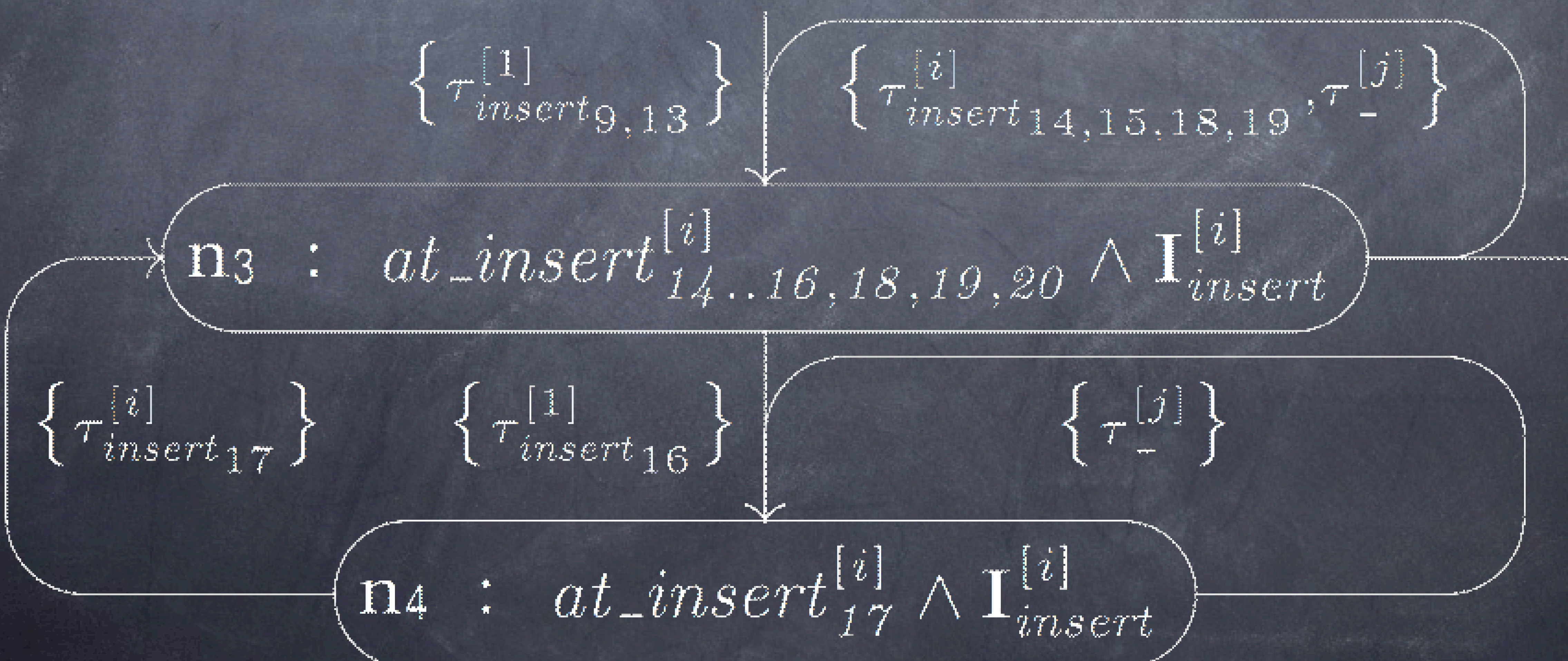
Fine-grained lock-coupling CSL



Fine-grained lock-coupling CSL

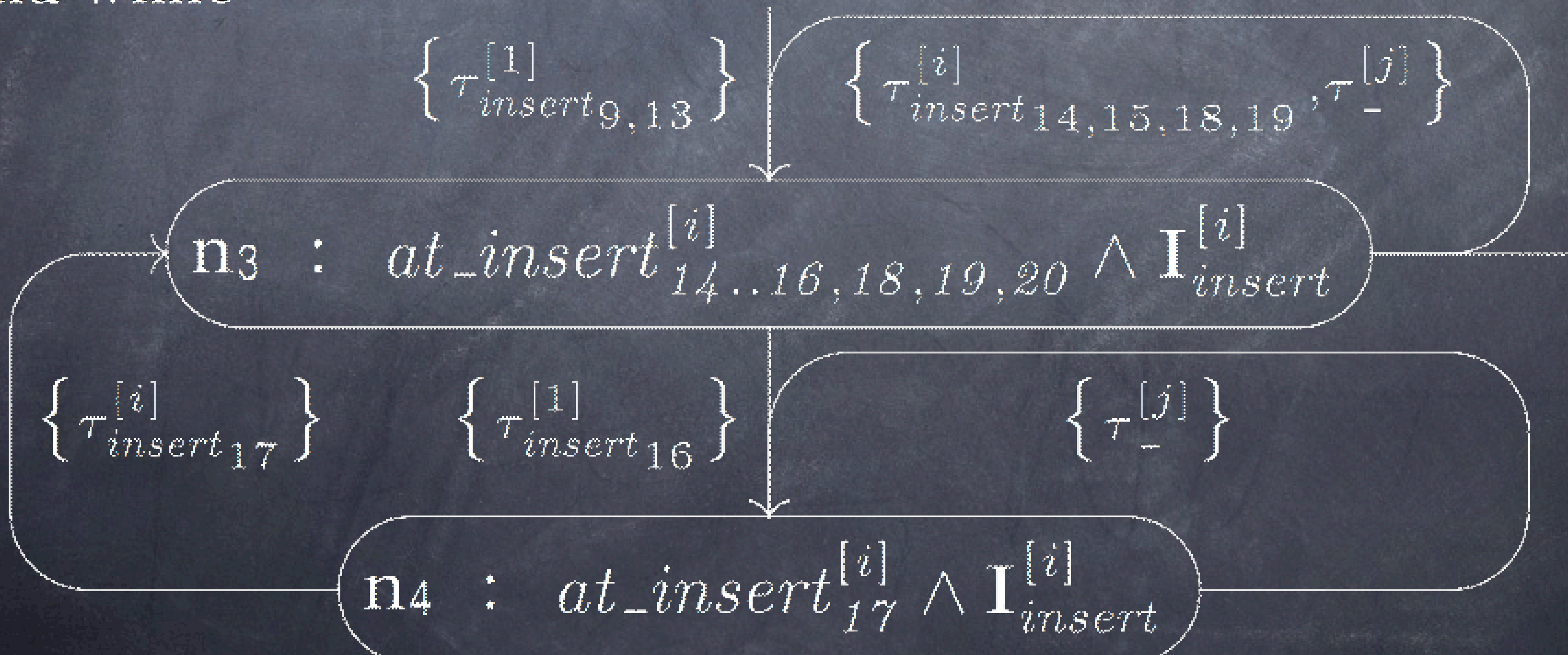


Fine-grained lock-coupling CSL



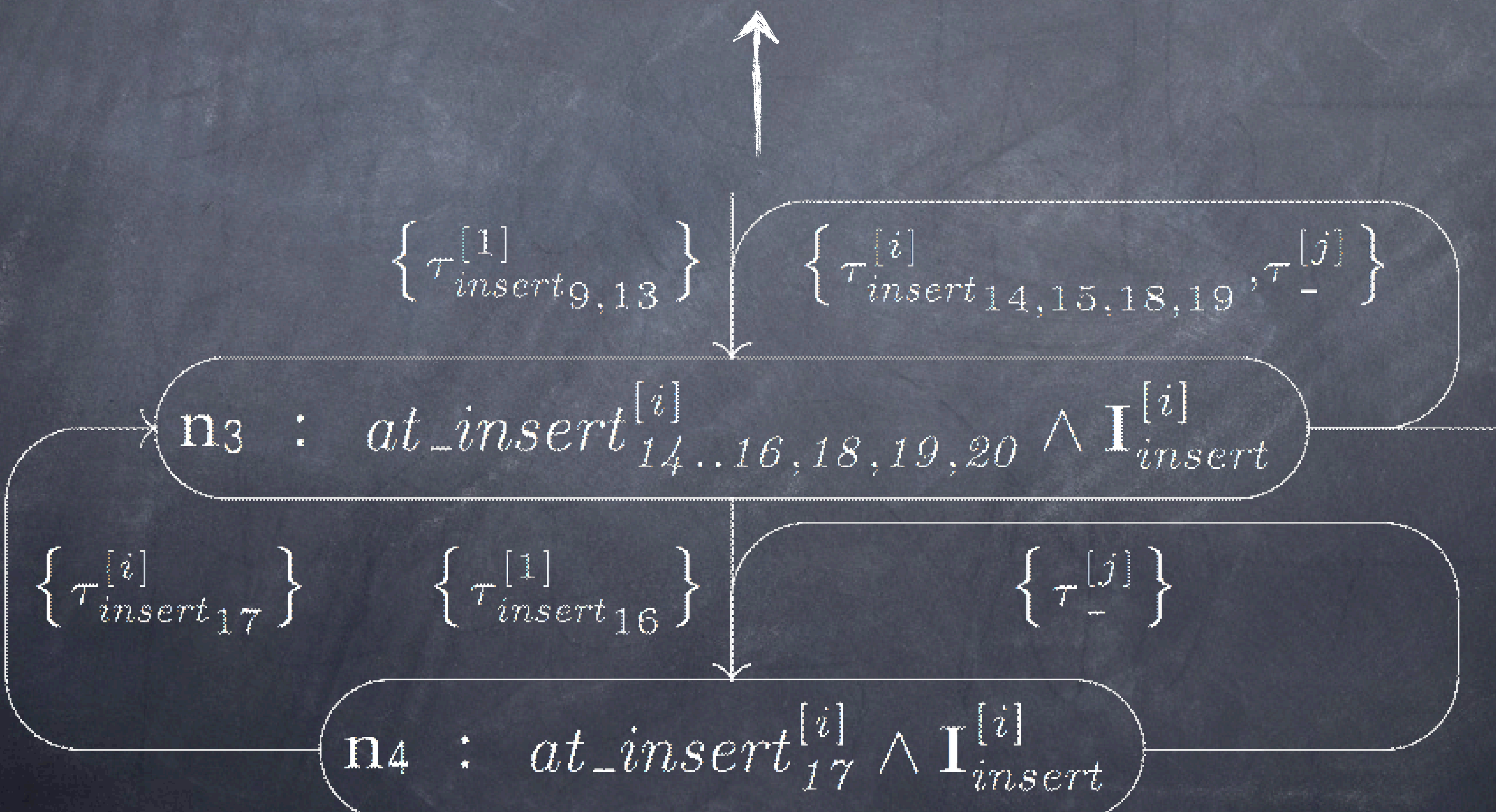
Fine-grained lock-coupling CSL

```
14: while curr.key < k do
15:     pred.locks[i].unlock()          //@ m_r := m_r − (pred, forward[i])
16:     pred := curr
17:     curr := pred.forward[i]
18:     curr.locks[i].lock()            //@ m_r := m_r ∪ (curr, forward[i])
19: end while
```



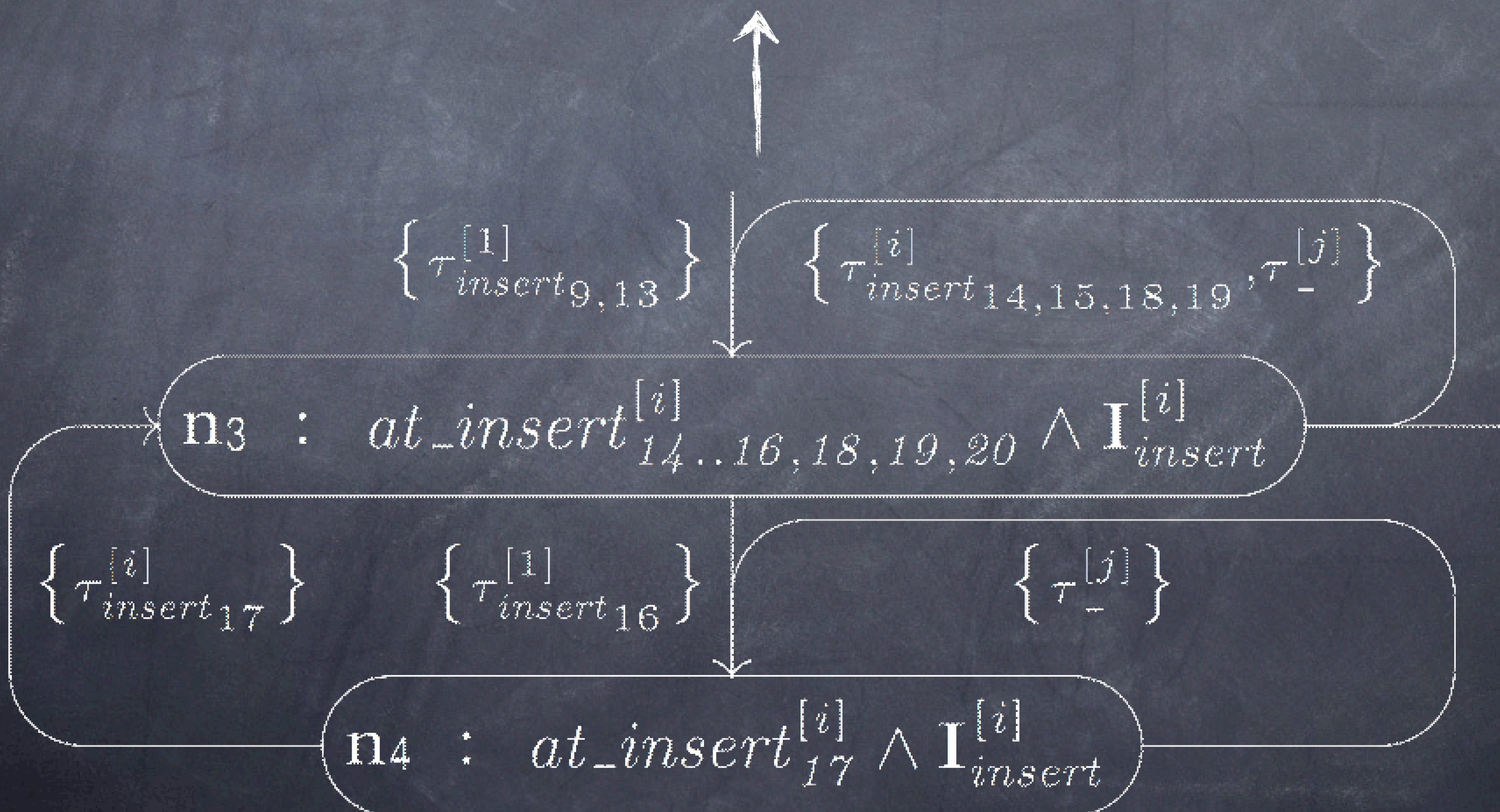
Fine-grained lock-coupling CSL

Verification conditions



Fine-grained lock-coupling CSL

Verification conditions ✓



Fine-grained lock-coupling CSL

Verification conditions ✓

$$\Psi$$

$$\|_{j \in T_{ID} - \{i\}} T_j \parallel T_i \models \varphi_{insert}(i)$$

Fine-grained lock-coupling CSL

Verification conditions ✓

$$\Psi$$

$$\downarrow$$

$$\|_{j \in T_{ID} - \{i\}} \; T_j \parallel T_i \models \; \varphi_{insert}(i)$$

Conclusions

Conclusions

- A method to formally verify temporal properties over concurrent data structures

Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties

Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
- A different approach to Separation Logic

Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
- A different approach to Separation Logic
- Good results over many mutable data structures

Conclusions

- A method to formally verify temporal properties over concurrent data structures
- Not just limited to safety properties
- A different approach to Separation Logic
- Good results over many mutable data structures
- Experience shows possibility of working with parameterized VD

Future work

Future work

- Extend the work over other concurrent data structures

Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams

Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions

Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
- Analyze decidability of involved logics

Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
- Analyze decidability of involved logics
- Development of assisted decision procedures

Future work

- Extend the work over other concurrent data structures
- Enrich verifications diagrams
- Automatic generation of verification conditions
- Analyze decidability of involved logics
- Development of assisted decision procedures
- This is just the beginning

Questions ?