香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

THE DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

From Shape Analysis to Heap Indexing

Chengpeng Wang

Dec 17th, 2020

Outline

- Shape analysis
- Problem
- Application
- Possible solution

Shape Analysis

- How memory locations are connected?
- Different logics
  - Three value logic
  - Separation logic

```
Node* curr = a;
while (curr->next != NULL) {
  curr = curr->next;
}
curr->next = b;
```

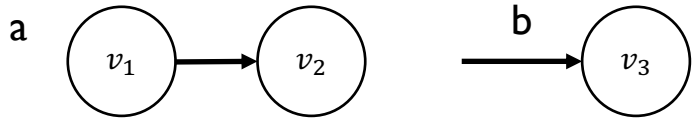
$$\mathrm{sll}(a, nil) * \mathrm{sll}(b, nil)$$

$$\mathrm{sll}(a, curr) * \mathrm{sll}(curr, b) * \mathrm{sll}(b, \mathrm{NULL})$$

Multi-domain Shape Analysis

- How the size of memory evolves?
- Add numeric domain to abstract numeric quantities

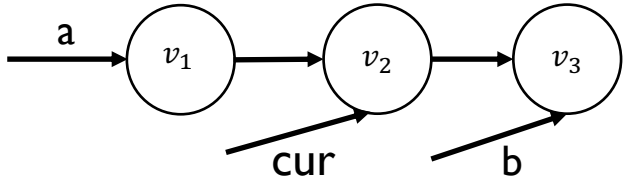
Before concat



Numeric quantity: Dis

dis(a, nil) = 2 dis(b, nil) = 1

After concat



dis'(a, curr) = 1 dis'(b, nil) = 1
dis'(curr, b) = 1

dis(a,nil) + dis(b,nil) = dis'(a, curr) + dis'(b,nil) + dis'(cur, b)

Limitation

- Combination of size and shape domain can not model semantics precisely

- Example: *func* is reverse or traversal

  Shape domain: {sll(x, nil)} func(x) {sll(x, nil)}

  Numeric domain: dis(x, nil) = dis'(x, nil)

Outline

- Shape analysis
- Problem
- Application
- Possible solution

Problem

- Given recursive data structures and a program, we concern the evolvement of locations of nodes in the data structures

Program Syntax

- Recursive data structure

  ONLY contain pointers pointing the instances with the same type

```
Node {
  int data;
  Node* n;
}
```

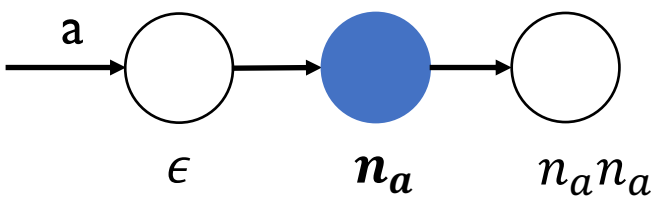
- Program

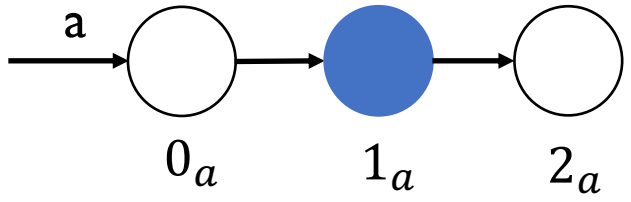
Pointer expression	L ::= x | L->f | nil
Condition.	E ::= x cmp y $(x, y \in L, cmp \in \{=, != \})$
Statement.	S ::= L=L | L = new | free(L) | assert E
Program.	P ::= S; P | while(E)P

Location of Node

- The set of access paths is subset of a regular language
  Single linked list: $\{\mathrm{n}\}^* = \{\epsilon, n, nn, \ldots\}$
  Binary tree: $\{\mathrm{l}, \mathrm{r}\}^* = \{\epsilon, l, r, lr, rl, rr, ll \ldots\}$

- Location of Node is a member of a regular language

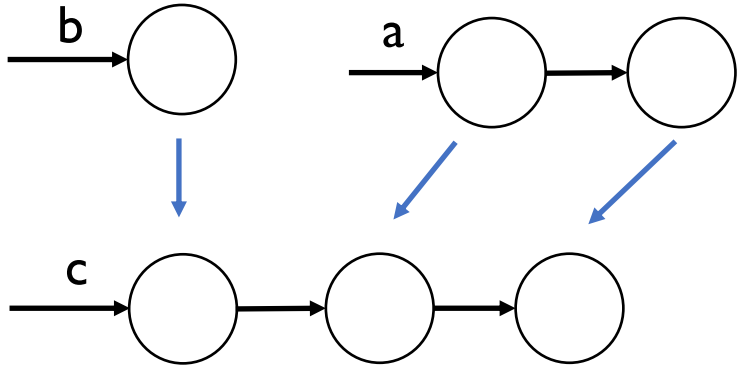




Example

- Insert_head

  Node* Insert_head(Node* a, Node* b)



Node* c = insert_head(a, b);

$$I_a = \{0_a, 1_a\}$$

$$I_b = \{0_b\}$$

$$I_c = \{0_c, 1_c, 2_c\}$$

Find a transformer to abstract the evolvement of locations of nodes

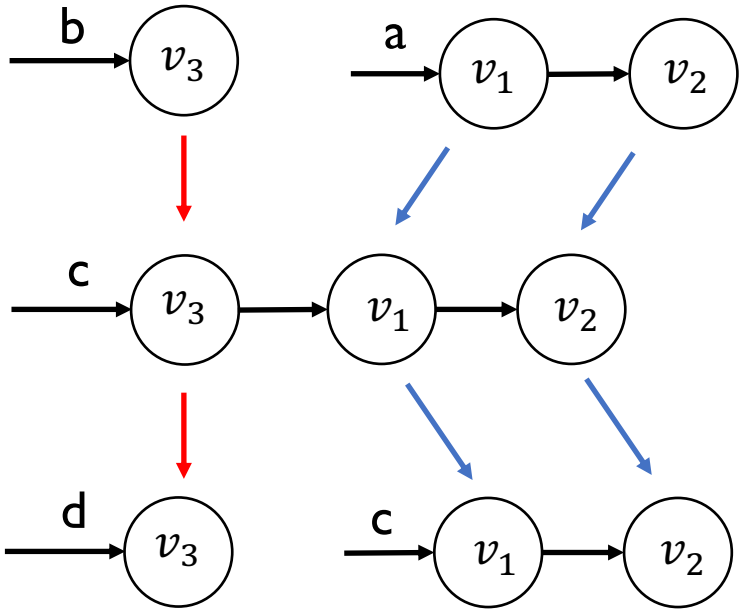
$$f_{a\to c}(i_a) = i_c + 1$$
$$f_{b\to c}(0_b) = 0_c$$

Outline

- Shape analysis
- Problem
- Application
  - Functional correctness verification
  - Termination analysis
  - Worst-Case-Execution-Time analysis
- Possible solution

Application: Functional Correctness

- Check the consistency of implementation and functional specification
  - Functional specification in documentation, assertion
  - A particular form of taint specification



$\mathrm{sll}(a, nil) * \mathrm{sll}(b, nil)$

c = insert_head(a, b)

$\mathrm{sll}(c, \mathrm{nil})$

d = delete_front(c)

$\mathrm{sll}(c, nil) * \mathrm{sll}(d, nil)$

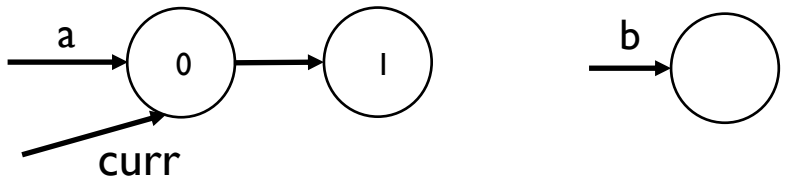
Application: Termination Analysis

- Prove a program halts for arbitrary valid inputs

```
Node* insert_tail(Node* a, Node* b) {
    Node* curr = a;
    while (curr->next != NULL) {
      curr = curr->next;
    }
    curr->next = b;
    return a;
}
```

$\text{sll}(a, \text{nil}) * \text{sll}(b, nil)$

$\text{sll}(a, cur) * \boldsymbol{sll(cur, nil)} * \text{sll}(b, nil)$



Node accessed by curr: 0_a

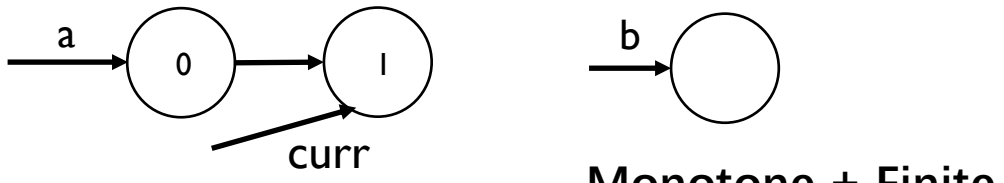
Application: Termination Analysis

- Prove a program halts for arbitrary valid inputs

```
Node* insert_tail(Node* a, Node* b) {
    Node* curr = a;
    while (curr->next != NULL) {
      curr = curr->next;
    }
    curr->next = b;
    return a;
}
```

$\mathrm{sll}(a, \mathrm{nil}) * \mathrm{sll}(b, nil)$

$\mathrm{sll}(a, cur) * \boldsymbol{sll(cur, nil)} * \mathrm{sll}(b, nil)$



Monotone + Finite

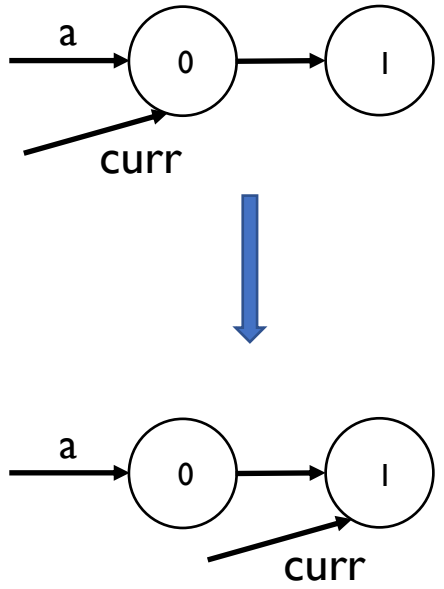
Node accessed by curr: 1_a Must terminate

Application: Worst-Case-Execution-Time analysis

- Determine the worst-case time complexity

```
Node* insert_tail(Node* a, Node* b) {
    Node* curr = a;
    while (curr->next != NULL) {
      curr = curr->next;
    }
    curr->next = b;
    return a;
}
```

$O(dis(a, nil))$



$dis(a, nil) = 2$

Outline

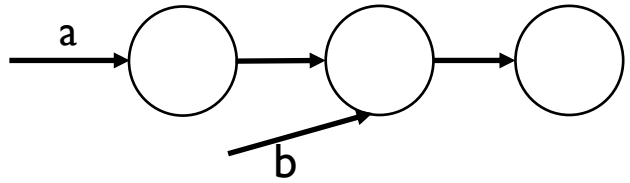
- Shape analysis
- Problem
- Application
- Possible solution

Insight

- Separation logic bridges shape domain to numeric domain

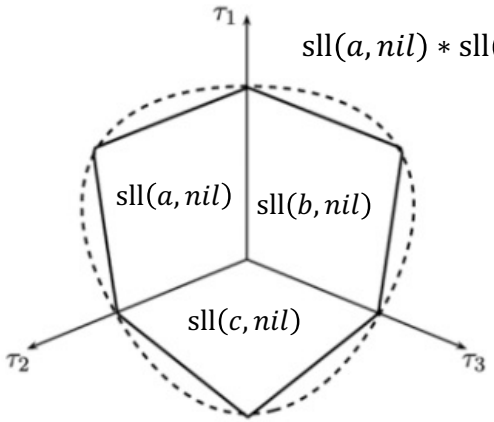
  Memory abstracted by a separating conjunction is well structured

$$\mathrm{sll}(a, b) * \mathrm{sll}(b, nil)$$



Natural to embed a numeric domain for a conjunction to index heap

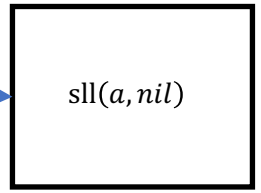
Heap Indexing



sll$(a, nil) *$ sll(b, nil) $* sll(c, nil)$

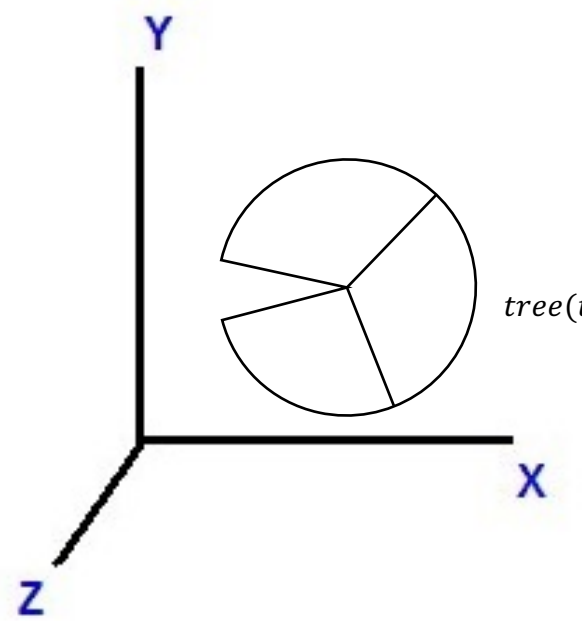
decompose

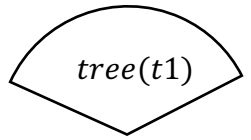
Indexing partition



$I_a \subset \{\epsilon, n_a, n_a n_a, \ldots\}$

$f_{a \to t_1} : I_a \to I_{t_1}$





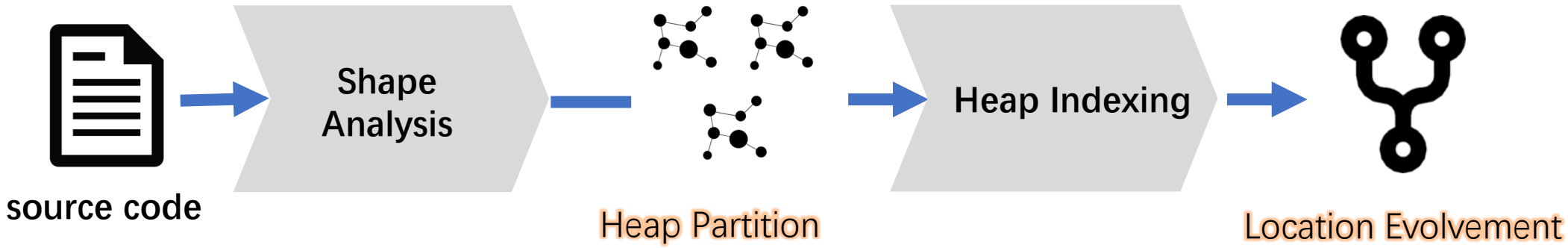
$I_{t_1} \subset \{\epsilon, l, r, lr, ll, rl, rr \ldots\}$

compose

$f_{a \to t_1}$
$f_{b \to t_1}$
$f_{c \to t_1}$

Find a transformer to abstract the evolvement of locations of nodes

Workflow


source code
Shape Analysis
Heap Partition
Heap Indexing
Location Evolvement

Example

- *Insert_head*

```
Node* Insert_head(Node* a, Node* b) {
    assert(b->next == nullptr);
    b->next = a;
    a = b;
    return a;
}
```

Phase 1: Shape Analysis

- Shape invariants at all program locations

```
Node* Insert_head(Node* a, Node* b) {
    assert(b->next == nullptr);
    b->next = a;
    a = b;
    return a;
}
```

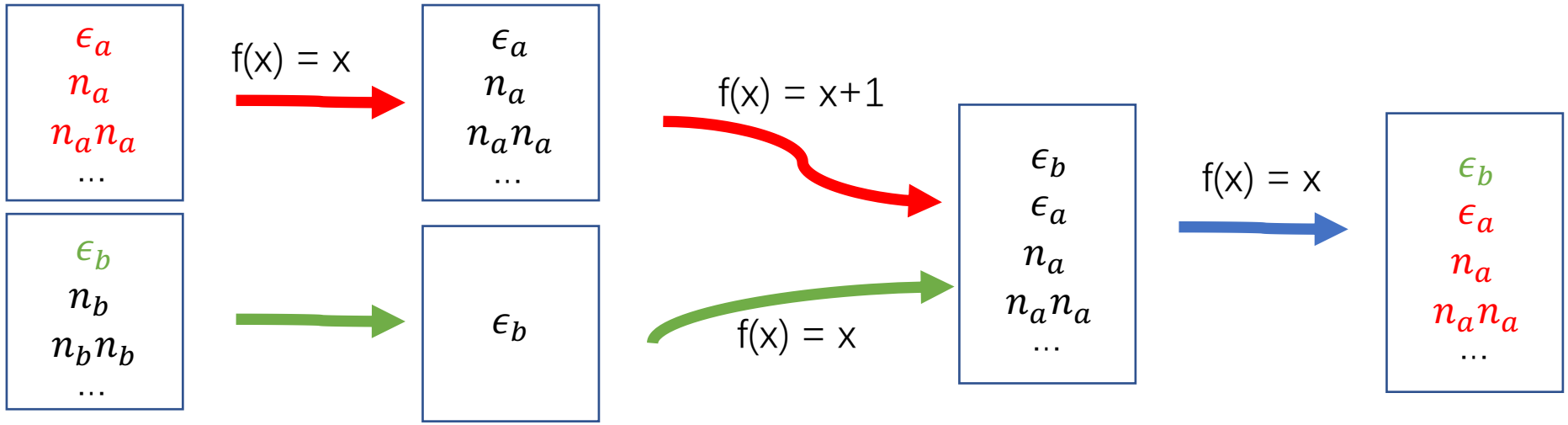
Line	Shape invariant
0	$\mathrm{sll}(a, nil) * \mathrm{sll}(b, nil)$
1	$\mathrm{sll}(a, nil) * \mathrm{sll}(b, nil)$
2	$\mathrm{sll}(b, nil)$
3	$sll(b, nil) \wedge a = b$
4	$sll(ret, nil) \wedge ret = a \wedge a = b$

Phase 2: Heap Indexing

- Construct index and relational index domain

```
Node* Insert_head(Node* a, Node* b) {
    assert(b->next == nullptr);
    b->next = a;
    a = b;
    return a;
}
```

Line	Partition
0	$H0a$: sll(a, nil). $H0b$: sll(b, nil)
1	$H1a$: sll(a, nil). $H1b$: sll(b, nil)
2	$H2b$: sll(b, nil)



Reference

- Thomas Reps, A relational approach to interprocedural shape analysis, TOPLAS 2004
- Sumit Gulwani, A combination framework for tracking partition sizes, POPL 2009
- Stephen Magill, Automatic numeric abstractions for heap-manipulating programs, POPL 2010
- Noam Rinetzky, From shape analysis to termination analysis in linear time, CAV 2016
- Xavier Rival, A relational shape abstract domain, NFM 2017
- ThanhVu Nguyen, SLinG: Using dynamic analysis to infer program invariants in separation logic, PLDI 2019

Thank you for your listening!

Heap vs Numeric

```
assert(sll(a, b) * sll(b, nil));
Node* curr = a;
while (b->next != NULL) {
  curr = curr->next;
  b = b->next;
}
```

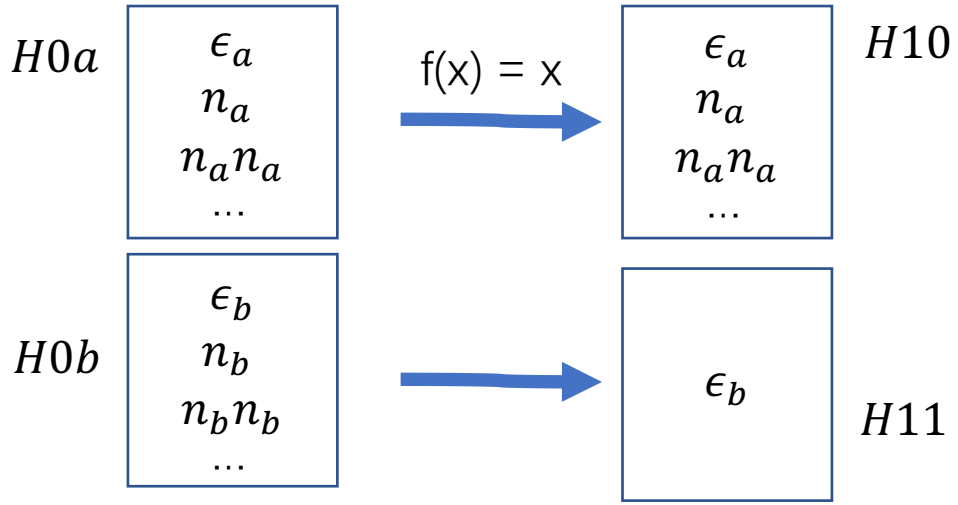
```
assert(m>=n && n >= 0)
int i = m;
while (n >= 0) {
    i--;
    n--;
}
```

Phase 2: Heap Indexing

- Construct index and relational index domain

```
Node* Insert_head(Node* a, Node* b) {
    assert(b->next == nullptr);
    b->next = a;
    a = b;
    return a;
}
```

Line	Partition	
0	$H0a$: $\mathrm{sll}(a, nil)$.	$H0b$: $\mathrm{sll}(b, nil)$
1	$H1a$: $\mathrm{sll}(a, nil)$.	$H1b$: $\mathrm{sll}(b, nil)$

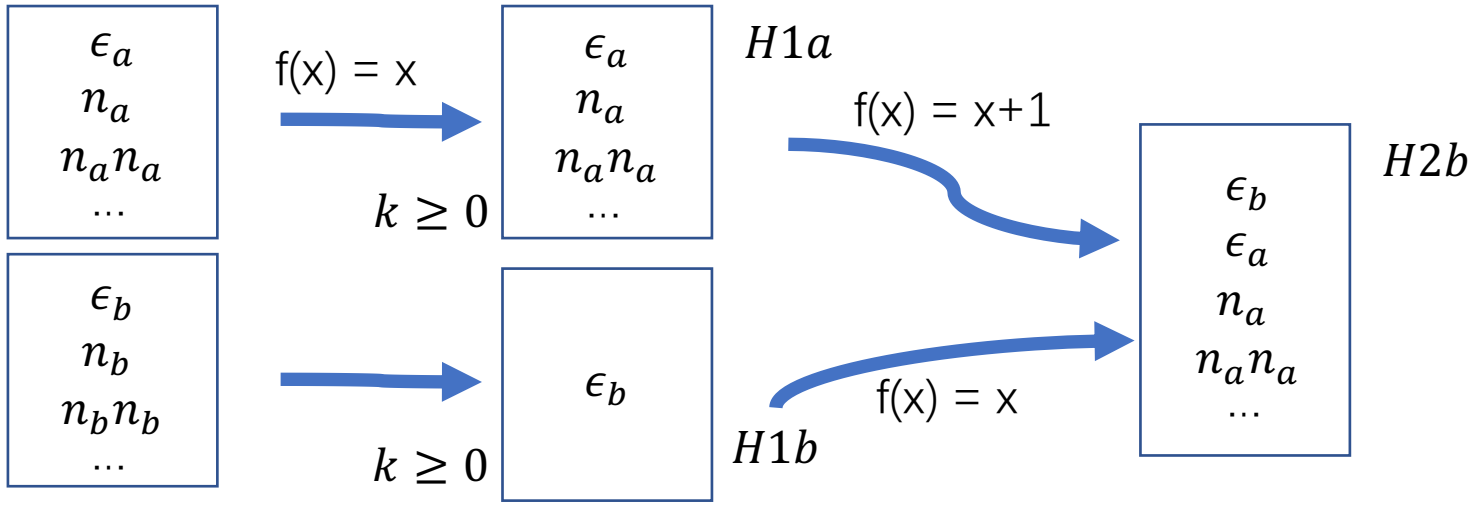


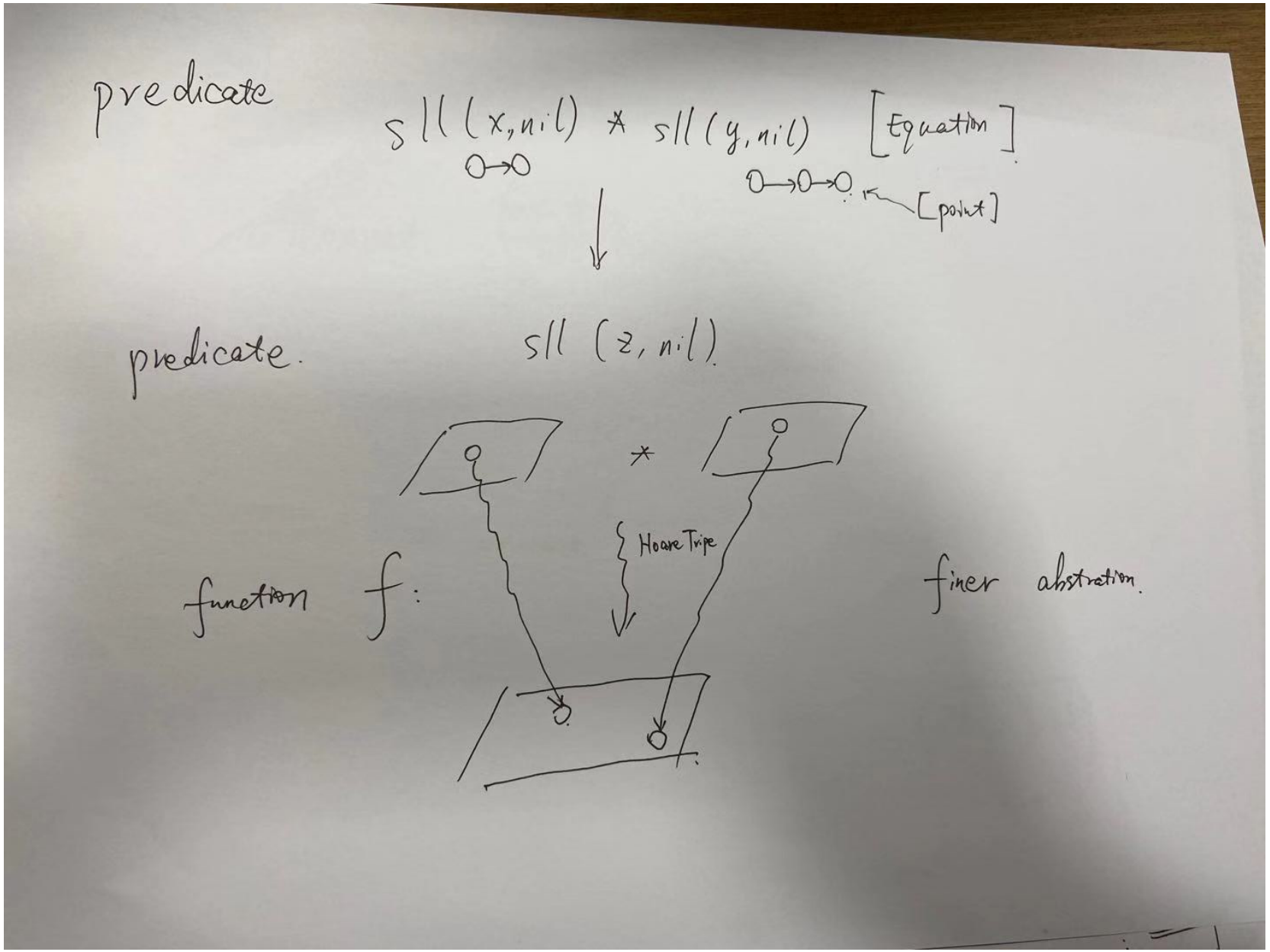
Phase 2: Heap Indexing

- Construct index and relational index domain

```
Node* Insert_head(Node* a, Node* b) {
    assert(b->next == nullptr);
    b->next = a;
    a = b;
    return a;
}
```

Line	Partition
0	$H0a$: sll(a, nil). $H0b$: sll(b, nil)
1	$H1a$: sll(a, nil). $H1b$: sll(b, nil)
2	$H2b$: sll(b, nil)




predicate
sll(x, nil) * sll(y, nil)
[Equation]
[point]
predicate.
sll(z, nil)
Hoare Tripe
function f:
finer abstration.

Memo

- Technical detail II
  - Encode location as string
  - Model transformer by Uninterpreted function
  - Solve out transformer by Z3