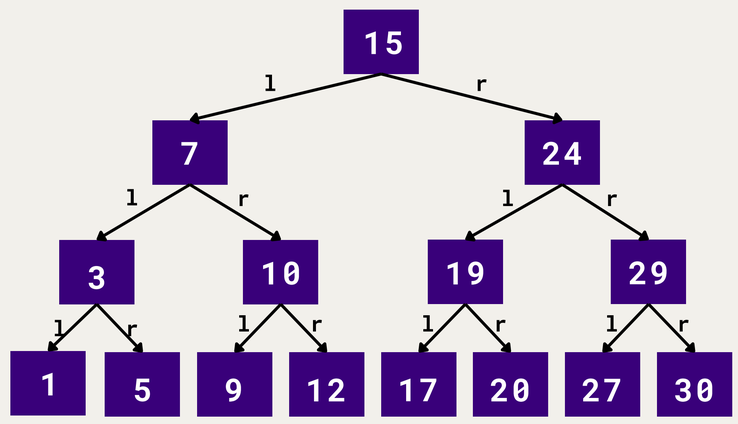
Heap Reference Analysis

Gargi Bakshi, Palle Bhavana
Guide: Prof. Uday Khedker
Department of Computer Science and Engineering, IIT Bombay

Source Code

```
1  typedef struct treeNode
2  {  int data;
3     struct treeNode* left;
4     struct treeNode* right;
5  } TN;
6
7  typedef struct q
8  {  int data;
9     struct q* next;
10 } QN;
11
12 TN* read_BST();
13 QN* insert(QN*, int);
14 QN* find(int key)
15 {  TN* p=read_BST();
16    QN* q=NULL;
17    while(p!=NULL)
18    {  q=insert(q,p->data);
19       if(key<p->data)
20          p=p->left;
21       else if(key>p->data)
22          p=p->right;
23       else break;
24    }
25    return q;
26 }
```

Example Tree



Key	List
5	15 → 7 → 3 → 5
7	15 → 7
15	15
10	15 → 7 → 10
20	15 → 24 → 19 → 20

Observations

- Entire memory isn't needed for any key
- The memory needed varies with the key
- This is not easy to determine at compile time
- Yet, compile-time optimizations are possible

Optimized Code

```
14 QN* find(int key)
15 {  TN* p=read_BST();
16    QN* q=NULL;
17    while(p!=NULL)
18    {  q=insert(q,p->data);
19       if(key<p->data)
20       {p->right=NULL; p=p->left;}
21       else if(key>p->data)
22       {p->left=NULL; p=p->right;}
23       else break;
24    }
25    p=NULL; return q;
26 }
```

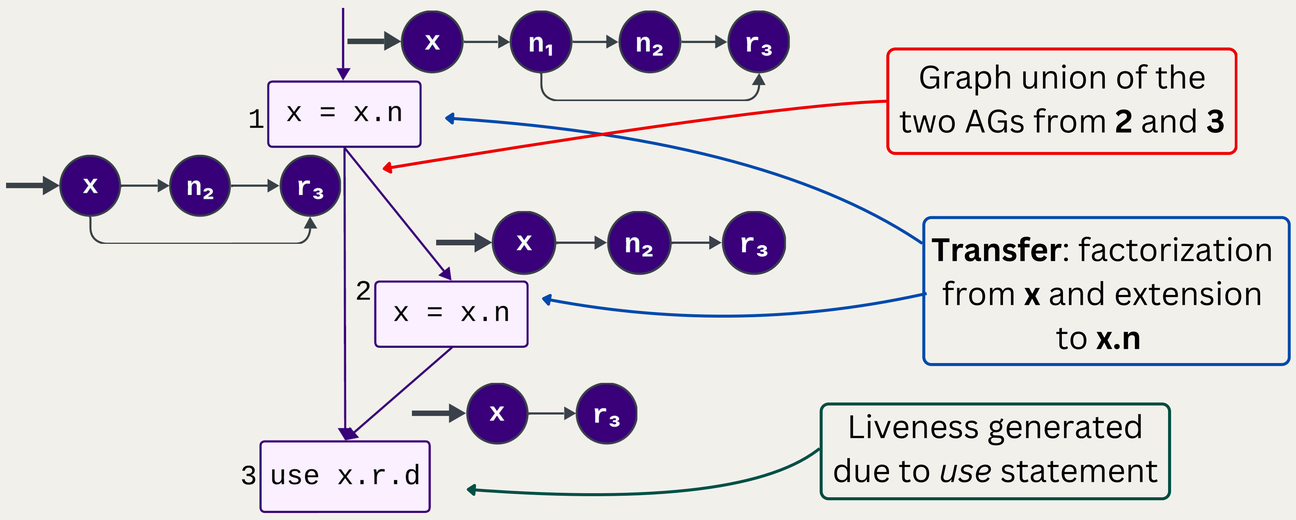
Challenges

- Stack/Static locations: have compile time names
- Heap locations don't have compile time names - accessed using **pointers**
- *Pointer expr -> Address* mapping keeps changing
- Cannot use *address-based view* of memory locations for optimizations

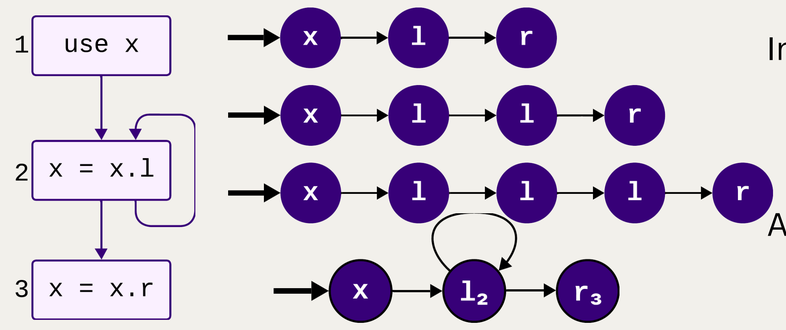
What is HRA?

- **H**eap **R**eference **A**nalysis identifies the memory that may be used in some run of the program (**live memory**)
- Access path view of memory instead of address-based view of memory
- Abstraction of the live parts of heap memory at each program point
- Insert **nullification statements** at appropriate program points for improved garbage collection
- Use **access graphs** to represent an unbounded set of *access paths*

Example

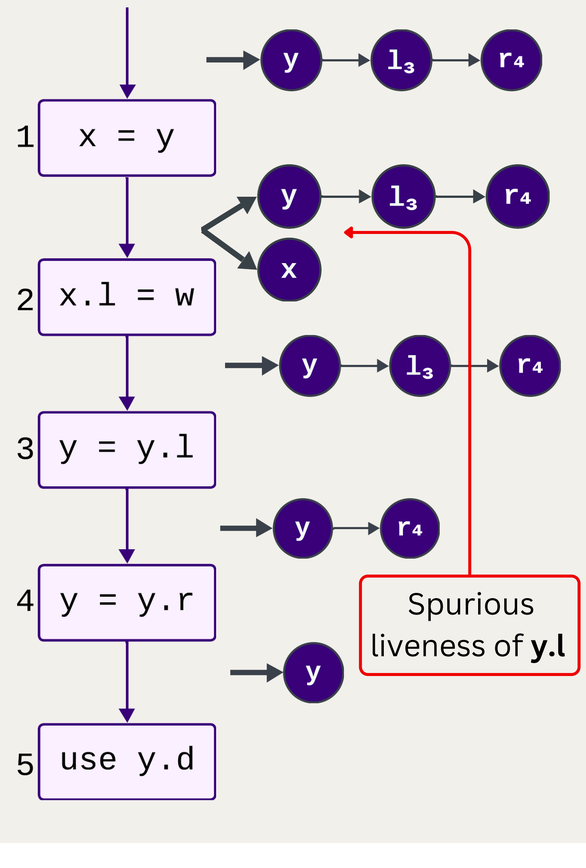


Access Graphs

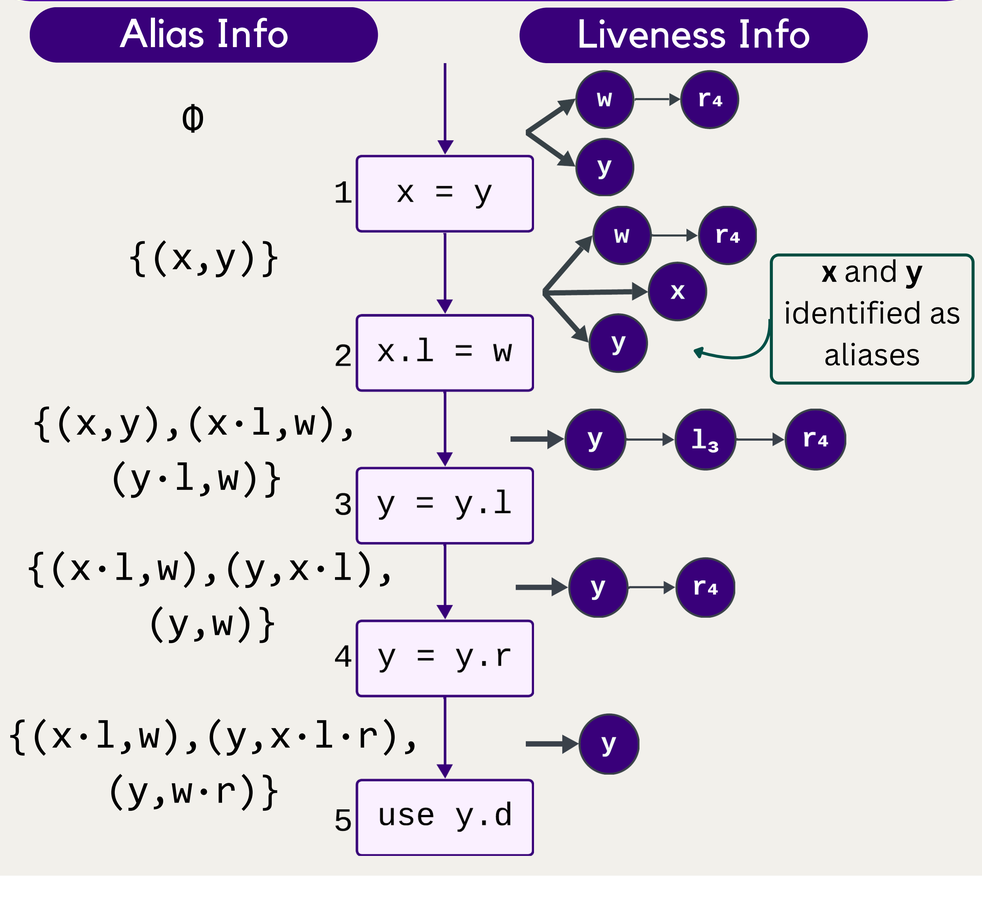


In the presence of loops, we get infinitely many access paths. Access graphs are finite for such cases

Default Liveness



Alias-Aware Liveness



Aliases

- For aliases generated through statements like **x = y**, x and y should be considered identical
- Incorporate **alias information** in explicit liveness analysis
- Alternating rounds of alias and liveness analysis - like **LFCPA**

Implementation

- Implementation using **SLIM** - Simplified LLVM IR Modelling - IR
- Access graph operations: *union, path removal, factorization, extension* in C++
- Using **VASCO** framework to implement and run the analysis
- Work in progress