

## Today

- ◆ **Pointer analysis**
  - What is pointer analysis?
  - May and must analysis
  - Interprocedural analysis
- ◆ **Most interesting analyses of imperative languages are impossible unless supported by pointer analysis**

## Pointer and Alias Analysis

- ◆ **Two expressions are aliases if they denote the same memory location**
- ◆ **Aliases are introduced by**
  - pointers
  - call-by-reference
  - array indexing
  - C unions
  - type casts

## Useful for what?

- ◆ **Improve the precision of analyses that require knowing what is modified or referenced (e.g. const prop, CSE, ...)**
  - ◆ **Eliminate redundant loads/stores and dead stores**
- ```

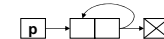
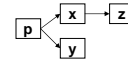
x := *p;                *x := ...;
...                    // is *x dead?
y := *p; // replace with y := x?
    
```
- ◆ **Parallelization of code**
    - Can recursive calls to quicksort be run in parallel? Yes, provided that they reference distinct regions of the array
  - ◆ **Identify objects to be tracked in error detection tools**

```

x.lock();
...
y.unlock(); // same object as x?
    
```

## Kinds of alias information

- ◆ **Points-to information (must or may versions)**
  - At program point, compute a set of pairs of the form  $p \rightarrow x$ , where  $p$  points to  $x$
  - Can represent this information in a points-to graph
- ◆ **Alias pairs**
  - At each program point, compute the set of all pairs  $(e_1, e_2)$  where  $e_1$  and  $e_2$  must/may reference the same memory
- ◆ **Storage shape analysis**
  - At each program point, compute an abstract description of the pointer structure
  - Data structure correctness arguments typically contingent on shape



## Intraprocedural Points-to Analysis

- ◆ **Want to compute may-points-to information**

- ◆ **Lattice:**

$L: 2^{\text{Var} \times \text{Var}}$        $\{x \rightarrow y, y \rightarrow z\}$   
                                           $\{(x, y), (y, z)\}$

$L = \emptyset$

$T = \text{Full set}$        $\cup = \checkmark$

## Transfer Functions

$$F_{x := k}(in) = in - \{(x, *)\}$$

$$F_{x := a + b}(in) = in - \{(x, *)\}$$

## Transfer Functions

$$\begin{array}{c} \text{in} \\ \downarrow \\ \boxed{x := y} \\ \downarrow \\ \text{out} \end{array} \quad F_{x:=y}(in) = in - \{ (x, *) \} \cup \{ (x, z) \mid (y, z) \in in \}$$

$$\begin{array}{c} \text{in} \\ \downarrow \\ \boxed{x := \&y} \\ \downarrow \\ \text{out} \end{array} \quad F_{x:=\&y}(in) = in - \{ (x, *) \} \cup \{ (x, y) \}$$

## Transfer Functions

$$\begin{array}{c} \text{in} \\ \downarrow \\ \boxed{x := *y} \\ \downarrow \\ \text{out} \end{array} \quad F_{x:=*y}(in) =$$

$$\begin{array}{c} \text{in} \\ \downarrow \\ \boxed{*x := y} \\ \downarrow \\ \text{out} \end{array} \quad F_{*x:=y}(in) =$$

## Transfer Functions for Intraprocedural May-Alias Analysis

$$\begin{aligned}
 kill(x) &= \bigcup_{v \in Vars} \{(x, v)\} \\
 F_{x:=k}(S) &= S - kill(x) \\
 F_{x:=a+b}(S) &= S - kill(x) \\
 F_{x:=y}(S) &= S - kill(x) \cup \{(x, v) \mid (y, v) \in S\} \\
 F_{x:=\&y}(S) &= S - kill(x) \cup \{(x, y)\} \\
 F_{x:=*y}(S) &= S - kill(x) \cup \{(x, v) \mid \exists t \in Vars. [(y, t) \in S \wedge (t, v) \in S]\} \\
 F_{*x:=y}(S) &= \text{let } V := \{v \mid (x, v) \in S\} \text{ in} \\
 &\quad S - (\text{if } V = \{v\} \text{ then } kill(v) \text{ else } \emptyset) \\
 &\quad \cup \{(v, t) \mid v \in V \wedge (y, t) \in S\}
 \end{aligned}$$

## Example of using points-to information

### ◆ In constant propagation:

$$F_{*x:=y}(M_{cp}, P_{ptsto}) = \text{let } V := \{v \mid (x, v) \in P_{ptsto}\} \text{ in} \\
 \text{let } \{v_1, \dots, v_n\} := V \text{ in} \\
 \text{let } M := M_{cp}[v_1 \mapsto \perp, \dots, v_n \mapsto \perp] \text{ in} \\
 \text{if } V = \{v\} \text{ then } M[v \mapsto M_{cp}(y)] \text{ else } M$$

## Pointers to Dynamically Allocated Memory

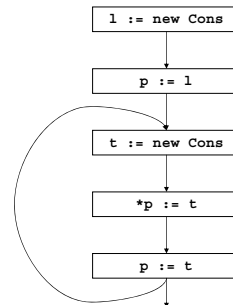
### ◆ Handle statements such as

$x := \text{new } T$

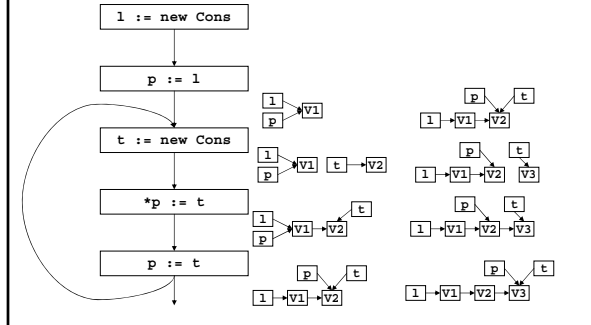
### ◆ One idea: Generate a new variable each time the new statement is analyzed to stand for the new location:

$$F_{x:=\text{new } T}(S) = S - kill(x) \cup \{\text{newvar}()\}$$

## Example



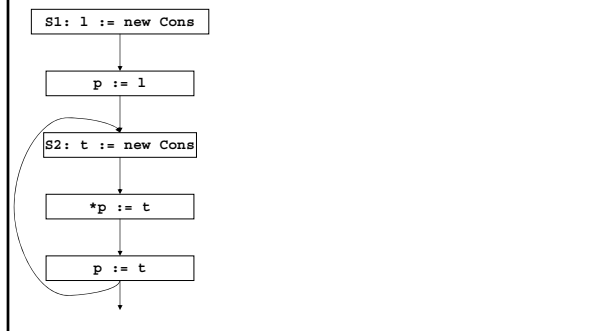
## Example



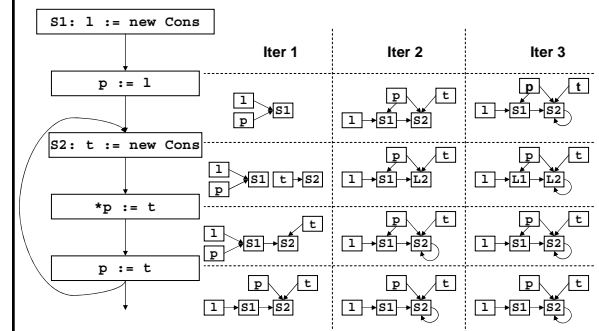
## Oops...

- ◆ Lattice has infinite height
- ◆ Instead, we need to summarize the infinitely many allocated objects in a finite way
  - Introduce summary nodes, which will stand for a whole class of allocated objects
- ◆ For example: For each new statement with label L, introduce a summary node  $loc_L$ , which stands for the memory allocated by statement L
  - $F_{L, x: new}(S) = S - kill(x) \cup (x, loc_L)$
- ◆ Summary nodes can use other criterion for merging

## Example Revisited



## Example Revisited



## Array Aliasing, Pointers to Arrays

- ◆ Array indexing can cause aliasing:
  - $a[i]$  aliases  $b[j]$  if:
    - $a$  aliases  $b$  and  $i = j$
    - $a$  and  $b$  overlap, and  $i = j + k$ , where  $k$  is the amount of overlap.
- ◆ Can have pointers to elements of an array
  - $p := \&a[i]; \dots; p++;$
- ◆ How can arrays be modeled?
  - Could treat the whole array as one location
  - Could try to reason about the array index expressions: array dependence analysis
    - Primarily important for numerical computation
    - E.g. scientific code in Fortran, DSP codes

- ◆ We looked at
  - Intraprocedural points-to analysis
  - Handling dynamically allocated memory
  - Handling pointers to arrays
- ◆ But, intraprocedural pointer analysis is not enough
  - Sharing data structures across multiple procedures is one of the big benefits of pointers: instead of passing the whole data structures around, just pass pointers to them (eg C pass by reference)
  - So pointers end up pointing to structures shared across procedures
  - If you don't do an interprocedural analysis, you'll have to make conservative assumptions at functions entries and function calls

## Conservative approximation on entry

- ◆ Say we don't have interprocedural pointer analysis
- ◆ What should the information be at the input of the following procedure:

```

global g;
void p(x,y) {
    ...
}
    
```

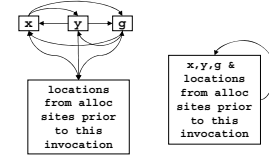


## Conservative approximation on entry

- ◆ Here are a few solutions:

```

global g;
void p(x,y) {
    ...
}
    
```



- ◆ They are very conservative!
- ◆ We can try to do better

## Interprocedural pointer analysis

- ◆ Main difficulty in performing interprocedural pointer analysis is scaling
- ◆ One can use a bottom-up summary based approach (Wilson & Lam 95), but even these are hard to scale

## Representing pointer information for C

- ◆ Problem:
  - C types can be subverted by type casts: an int can in fact be a pointer
  - Pointer arithmetic can lead to subobject boundaries being crossed
- ◆ So, ignore the type system and subobject boundaries
  - Instead use a representation that decides what's a pointer based on how it is used
- ◆ Treat memory as composed of blocks of bits:
  - each local, global variable is a block
  - each malloc returns a block
- ◆ Assume that casts and pointer arithmetic do not cross object boundaries

## Summary

- ◆ Pointer analysis is critical for verification and optimizations
- ◆ Lots of approaches exist
- ◆ Production compilers tend to be pretty dumb
  - Imprecision leads to lost opportunities for optimization
  - In many cases imprecision can be avoided through clever programming practice
- ◆ Bug-finding tools have to be better
  - Imprecision leads to false positives
  - These can easily render a tool useless
- ◆ State of the art involves making interprocedural flow, path, and context sensitivity scale to real programs