

# **CSC2/455 Software Analysis and Improvement Interprocedural Analyses - I**

---

Sreepathi Pai

March 15, 2021

URCS

# Outline

Introduction to Points-to analysis

Points-to Analysis

Interprocedural Analyses

Postscript

# Outline

Introduction to Points-to analysis

Points-to Analysis

Interprocedural Analyses

Postscript

# Pointers

- Variables that contain addresses
- Support a dereferencing operation
  - translates to a (indirect) function call if pointer to function
  - translates to a memory load/store if pointer to data
- Primary purpose of pointers
  - Treat addresses as data
  - Support indirection

# Pointer values

- Language can contain an “address-of” operator
  - e.g. `&` in C
  - allows obtaining address of an object to assign to a pointer
- Some names are addresses
  - Function names are addresses in C
  - Labels are not, at least in standard C
- Languages usually contain facilities to create/destroy *heap* objects
  - i.e. dynamically allocated objects in heap
  - e.g. `new` and `delete` in C++
  - note: `malloc` (and `free`) in C are not part of language
- Most sane languages do not use pointers directly
  - Some notion of a reference
  - e.g. Java, Python, ...

# The Points-To Problem

Given a pointer (or reference) variable  $x$ , what does it point to?

## Data Pointers

```
void f(int *x, int *y, int N) {  
    for(int i = 1; i < N; i++) {  
        x[i] = y[i-1];  
    }  
}
```

Can we execute different iterations of this loop on different processors? I.e., is this loop parallelizable?

# Unrolling that loop

```
x[1] = y[0]  
x[2] = y[1]  
x[3] = y[2]  
...
```

Statically examining this code, can we say that each write to `x` is independent?



# Parallelization

Case 1:

```
int x[100];  
int y[100];  
f(x, y, 100);
```

Case 2:

```
int x[101];  
f(x, x, 100);
```

# Parallelization

The `restrict` keyword tells the compiler that a restricted pointer is the only way to access the memory region it points to.

```
void f(int * restrict x, int * restrict y)
```

would assume no calls like `f(x, x)` are possible.

CPP Reference: [restrict type qualifier](#)

# Code/Function Pointers

```
int (*cmp)(int x, int y)
```

```
cmp = my_cmp_fn
```

```
*(cmp)(1, 2);
```

## Code/Function Pointers - II

```
x = my_obj.cmp(1, 2)
```

Which function does `cmp` point to?

What if `cmp` is a virtual function?

- recall a virtual function is a function that can be overridden in a child class
- and is called *even* if the calling object's type is the parent class
- also known as *dynamic dispatch*

# Virtual Function Implementation

- Usually through a VTABLE implemented by the parent class
  - An array of function pointers
- Compiler inserts code to fill in this table correctly for each child class
  - points function pointer to child's implementation
- Calls are resolved by looking up this table
  - First look up pointer in vtable
  - Then call function
- Obviously slower than a direct function call
  - Note, all function calls in Python are dynamically dispatched!
- Could be sped up if we knew which function to call at a particular call site

# Uses of points-to analysis

- Can be used to disambiguate data pointers
  - Useful for parallelization
- Can be used to identify callees of an indirect function call
  - If there is only one, no need to look up the vtable
- Lots of other uses
  - Security: Can a user-controlled pointer be used to overwrite sensitive data?
  - Program verification: is there a null pointer dereference?

# Outline

Introduction to Points-to analysis

**Points-to Analysis**

Interprocedural Analyses

Postscript

# The end results of points-to analysis

- For each pointer variable
  - Compute the set of objects it can point to
  - Called the points-to set for that variable
- Variant: Alias analysis
  - Do two pointers potentially point to the same thing?  
(Intersection of points-to sets of the two pointers is not empty)



# Objects that can be pointed to

Points-to sets can contain the following classes of objects:

- Statically known variables
  - These have names
- Heap-allocated variables (e.g. `new`, `malloc`, etc.)
  - These are anonymous
  - How do we handle these?

## Basic facts and relations

How should the set `pointsTo[x]` for a pointer `x` be updated?

- Object creation
  - `x = malloc(...)`
- Assignment
  - `x = y` (where both `x` and `y` are pointers)
- `AddressOf`
  - `x = &y`
- Indirect Assignment to a pointer
  - `*p = &y` (what happens if `p` is a pointer to pointer?)
- Indirect Assignment from a pointer
  - `p = *y` (if `y` is a pointer to pointer)

## Basic facts and relations - II

- Object creation
  - `x = malloc(...)`
  - `pointsTo[x].append(H)` where H is our way of referring to the anonymous object
- Assignment
  - `x = y` (where both x and y are pointers)
  - `pointsTo[x].union(y)` [inclusion-based]
  - `pointsTo[x].union(y)` and `pointsTo[y].union(x)` [equivalence-based]
- AddressOf
  - `x = &y`
  - `pointsTo[x].append(y)`

# Pointers to pointers

```
int **p;  
int *a;  
int x, y;  
  
p = &a;    // p points to a  
a = &x;    // a points to x  
*p = &y;   // what does 'p' point to? what does 'a' point to?
```

## Basic facts and relations – II contd

- Indirect Assignment to a pointer
  - `*p = &y` (what happens if `p` is a pointer to pointer?)
  - `forall o in pointsTo[p]: pointsTo[o].append(y)`
- Indirect Assignment from a pointer
  - `p = *y` (if `y` is a pointer to pointer)
  - `forall o in pointsTo[y]: pointsTo[p].union(o)`

# Flow-insensitive Points-To Analysis

```
h: a = malloc(...)  
i: b = malloc(...)  
j: c = malloc(...)
```

```
a = b;  
b = c;  
c = a;
```

- A flow *insensitive* analysis ignores control flow:
  - $\text{pointsTo}[a] = \{h, i\}$
  - $\text{pointsTo}[b] = \{i, j\}$
  - $\text{pointsTo}[c] = \{j, h, i\}$
- A flow *sensitive* analysis respects control flow:
  - $\text{pointsTo}[a] = \{h\}$  at point h, then  $\text{pointsTo}[a] = \{i\}$  at first assignment
  - etc.
- Note: updates in previous relations slides were all flow-insensitive!

## What about function calls?

```
int f(int x, int y) {  
    x = y;  
}  
  
a = 1;  
b = 2;  
f(a, b)  
// what are values of a and b?
```

## What about function calls?

```
int f(int **x, int *y) {
    *x = y;
}

int *a;
int b, c;

a = &c;
b = 2;
f(&a, &b)
// what does a point to here?
```



# Inter-procedural Analysis

- C is call-by-value language
  - Arguments cannot be changed by function
- But pointers can be arguments
  - And *pointees* can be changed!
- In general, incorporating the effects of function calls is called interprocedural analysis
  - Context-insensitive
  - Context-sensitive
- Note: interprocedural analyses aren't limited to functions with pointers as arguments

# Flavours of Points-To Analysis

- Flow:
  - Flow-sensitive
  - Flow-insensitive
- Context:
  - Context-sensitive
  - Context-insensitive

# Outline

Introduction to Points-to analysis

Points-to Analysis

**Interprocedural Analyses**

Postscript

# Strategies for handling effects of functions

Can we reuse our CFG-based analyses to handle function calls?

- Inlining, perhaps?
- What are the limitations of inlining?

# Context-insensitive analysis

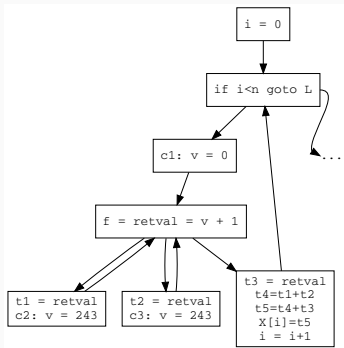
- Treat function calls as control flow
  - Functions form node in CFG
  - Invocations are treated as "gotos" to entry of function
  - The returns as goto to location after function call
- This results in CFG with:
  - multi-entry nodes
  - multi-exit nodes that are not branches (example in a few slides)

## Example

```
    for(i = 0; i < n; i++) {  
c1:      t1 = f(0);  
c2:      t2 = f(243);  
c3:      t3 = f(243);  
          X[i] = t1 + t2 + t3;  
    }  
  
int f(int v) {  
    return (v+1);  
}
```

Is the value of  $X[i]$  constant?

# CFG for Example



- values for:
  - `v` into `f`
  - `retval` out of `f`
  - `t1`, `t2` and `t3`?
  - `X[i]`?

# Context-sensitivity

- In treating function calls as control-flow, we lost the ability to detect *context*
- Context is the call stack for the function
  - Can be treated as a string, called a call string
  - If  $c_1$  calls  $c_2$ , which then calls  $c_3$ , the context for  $c_3$  is  $(c_1, c_2)$
- How many contexts can there be?
  - consider indirect function calls
  - consider recursive functions
- $k$ -limiting context sensitivity
  - Limit context to  $k$  immediate call *sites* (important!)
  - 0 is context-insensitive



# Cloning-based Context-Sensitive Analysis

```
    for(i = 0; i < n; i++) {  
c1:      t1 = f1(0);  
c2:      t2 = f2(243);  
c3:      t3 = f3(243);  
        X[i] = t1 + t2 + t3;  
    }  
  
int f1(int v) {  
    return (v+1);  
}  
  
int f2(int v) {  
    return (v+1);  
}  
  
int f3(int v) {  
    return (v+1);  
}
```

- Create a clone for each unique calling context and then apply context-insensitive analysis
- Is this the same as inlining?
  - See textbook for a differentiating example

## Interprocedural Points-To Analysis?

Next class ...

# Outline

Introduction to Points-to analysis

Points-to Analysis

Interprocedural Analyses

Postscript

# References

- Chapter 12 of the Dragon Book
  - Section 12.1
  - Section 12.4