Alias (Pointer) Analysis

CIS410/510 Program Analysis and Transformation

Alias analysis

- Last time
  - Dataflow analysis examples
  - Single-function scope

- Today
  - Alias analysis
  - Introduction to interprocedural analysis
How do aliases arise in various languages?

- Pointers (e.g., C, C++):
  ```
  int *p, i; p = &i;
  ```

- Parameter passing by references (e.g., C++)
  ```
  foo(int &x, int &y);
  //...
  foo(x, x);
  foo(x, glob);
  ```

- Arrays (C, Java)
  ```
  int i, j, a[128];
  i = j; // C: a[i] and a[j] are aliased
  ```
  ```
  B[] b = new B[10]; // Java
  A[] a = b; // a and b are aliased
  ```

---

Alias analysis: Motivation

```c
void foo(int x, int y, int a) {
    int *p;

    p = &a;
    x = 5;

    y = x + 1; // Is x constant here?
}
```
The importance of pointer analysis

```c
void foo(int x, int y, int a) {
    int *p;

    p = &a;
    x = 5;
    *p = 23;
    y = x + 1;
    // Is x constant here?
    // - If p does not point to x, then x = 5
    // - If p definitely points to x, then x = 23
    // - If p might point to x, then we have two reaching definitions that reach that last statement, so x is not constant.
}
```

Trivial pointer analysis

```c
void foo(int x, int y, int a) {
    int *p;

    p = &a;
    x = 5;
    *p = 23;
    y = x + 1;
    // No analysis
    // - Assume that nothing must alias
    // - Assume that everything may alias everything else
    // - YUCK!
    // - Use type information somehow?

    // Is x constant here?
    // - With our trivial analysis we assume that p may point to x, so x is not constant.
```
Slightly better approach (for C)

```c
void foo(int x, int y, int a) {
    int *p;
    p = &a;
    x = 5;
    *p = 23;
    y = x + 1;
}
```

- **“Address Taken”**
  - Assume that nothing *must* alias
  - Assume that all pointer dereferences *may* alias *each other*.
  - Assume that variables whose addresses are
taken (and globals) *may* alias all pointer
dereferences.

- **Is x constant here?**
  - With “Address Taken”, *p and a may alias,
  but neither aliases with x.

Address Taken (cont.)

```c
void foo(int x, int y, int a) {
    int *p, *q;
    q = &x;
    p = &a;
    x = 5;
    *p = 23;
    y = x + 1;
    // Is x constant here?
    // With “Address Taken”, we now assume that
    // *p, *q, a, and x all alias.
}
```
A better Points-To analysis

Goal
- At each program point, compute set of $(p \rightarrow x)$ pairs if $p$ points to $x$

Properties
- Use dataflow analysis
- *May* information (will look at must information later)

May Points-To analysis

- Domain: $2^{\text{var}} \times \text{var}$
  - Recall cross product of sets: $A \times B = \{(x, y) \mid x \in A, y \in B\}$
- Direction: forward
- Transfer functions for:
  - $s: p = &x$;
  - $s: p = q$;
- Meet function: $\cup$
- What if we have pointers to pointers?
  - e.g., `int **q; p = *q;`
**May Points-To analysis (pointers to pointers)**

### Additional flow functions

- \( s: \ p = \ast q; \)
  
  \[
  \text{out}[s] = \{ (p \rightarrow t) \mid (q \rightarrow r) \in \text{in}[s] \land (r \rightarrow t) \in \text{in}[s] \} \cup \\
  (\text{in}[s] - \{ (p \rightarrow x) \forall x \})
  \]

- \( s: \ \ast p = q; \)
  
  \[
  \text{out}[s] = \{ (r \rightarrow t) \mid (p \rightarrow r) \in \text{in}[s] \land (q \rightarrow t) \in \text{in}[s] \} \cup \\
  (\text{in}[s] - \{ (r \rightarrow x) \forall x \mid (p \rightarrow r) \in \text{in}[s] \})
  \]

---

**May Points-To analysis (cont.)**

- What about
  
  \( x = 3; \)
  
  \( \ast p = x; \)

  \[
  \text{out}[s] = \text{in}[s]
  \]

\[
\text{out}[s] = \text{in}[s]
\]
Dealing with dynamically allocated memory

- **Issue**
  - Each new allocation creates new storage location, e.g., `p = new T;`

- **Proposal?**
  - Generate (at compile time) a new “variable” to stand for new storage

- **Flow function**
  - `s: p = new T;`
  - `out[s] = {(p→newvar)} ∪ (in[s] – {(p→x) ∀x})`

- **Problem**
  - Domain is unbounded!
  - Iterative dataflow analysis may not converge

---

Dynamically allocated memory (cont.)

- **Simple solution**
  - Create a summary “variable” (node) for each allocation statement
  - Domain: `2(Var ∪ Stmt) · (Var ∪ Stmt)` rather than `2Var · Var`
  - **Monotonic** flow function
    - `s: p = new T;`
    - `out[s] = {(p→stmt)} ∪ (in[s] – {(p→x) ∀x})`
  - Less precise (but finite)

- **Alternatives**
  - Summary node for entire heap
  - Summary node for each type
  - K-limited summary
    - Maintain distinct nodes up to k links removed from root variables
**Must points-to analysis**

- Meet function: \( \cap \)
- Analogous flow functions
  
  - s: \( p = \& x \);
    
    \[ \text{out}_{\text{must}}[s] = \{(p \rightarrow x)\} \cup (\text{in}_{\text{must}}[s] - \{(p \rightarrow x) \forall x\}) \]
  
  - s: \( p = q \);
    
    \[ \text{out}_{\text{must}}[s] = \{(p \rightarrow t) | (q \rightarrow t) \in \text{in}_{\text{must}}[s]\} \cup (\text{in}_{\text{must}}[s] - \{(p \rightarrow x) \forall x\}) \]
  
  - s: \( p = *q \);
    
    \[ \text{out}_{\text{must}}[s] = \{(p \rightarrow t) | (q \rightarrow r) \in \text{in}_{\text{must}}[s] \land (r \rightarrow t) \in \text{in}_{\text{must}}[s]\} \cup (\text{in}_{\text{must}}[s] - \{(p \rightarrow x) \forall x\}) \]
  
- Compute this along with *may* analysis
  
  - Why?

**Definiteness of alias information**

- Often need both *may* and *must*
  
  - Consider liveness analysis
    
    Recall: in[s] = use[s] \( \cup \) (out[s] - def[s])

  - Suppose out[s] = \{v\}

  - \( *p \) must alias \( v \) \( \rightarrow \) def[s] = kill[s] = \{v\}

- *May* (possible) alias information
  
  - Indicates what might be true, e.g.,
    
    \[ \text{if (c) } p = \& i; \]

  - *p and i may be aliased*

- *Must* (definite) alias information
  
  - Indicates what is definitely true, e.g.,
    
    \[ p = \& i; \]

  - *p and i must be aliased*
Using Points-To information

```c
void foo(int x, int y, int a)
{
    int *p, *q;
    q = &x;
    p = &a;
    x = 5;
    *p = 23;
    y = x + 1;
}
```

To support constant propagation, first run points-to analysis:

- `q = &x;`  
  - `((q→x))`
- `p = &a;`  
  - `((q→x), (p→a))`
- `x = 5;`  
  - `((q→x), (p→a))`
- `*p = 23;`  
  - `((q→x), (p→a))`
- `y = x + 1;`  
  - `((q→x), (p→a))`

Then run constant propagation

- Since `*p` and `x` are not aliased, `x` is constant in this last statement

Alias (pointer) analysis is an enabling analysis

Using Points-To Information (cont.)

- Example: reaching definitions
  - Compute at each point in the program a set of \((v, s)\) pairs indicating that statement `s` may define variable `v`

- Flow functions
  - `x = y;`
    - `\text{out}_{\text{reach}}[s] = \{(x, s) \cup (\text{in}_{\text{reach}}[s] - \{(x, t) \forall t\}}`  
  - `x = *p;`
    - `\text{out}_{\text{reach}}[s] = \{(x, s) \cup (\text{in}_{\text{reach}}[s] - \{(x, t) \forall t\}}`  
  - `*p = x;`
    - `\text{out}_{\text{reach}}[s] = \{(z, s) \mid (p→z) \in \text{in}_{\text{must-pf}}[s]\} \cup (\text{in}_{\text{reach}}[s] - \{(y, t) \forall t \mid (p→y) \in \text{in}_{\text{must-pf}}[s]\}}`  
  - `...`
Function calls

```c
void foo(int *p) {
    return p;
}
```

```c
int x, y, a;
int *p;
ap = &a;
x = 5;
foo(&x);
y = x + 1;
```

Does the function call modify `x`?

- With our intra-procedural analysis so far, we don't know
- Make worst case assumptions
  - Assume that any reachable pointer may be changed
  - Pointers can be “reached” via globals and parameters
    - May pass through objects in the heap

Representations of aliasing

- Points-to pairs  
  - Pairs where the first member points to the second
  - e.g., (a -> b), (b -> c)

- Alias pairs  
  - Pairs that refer to the same memory
  - e.g., (**a, b), (**b, c), (**a, c)
  - Completely general
  - More concise than points-to pairs

- Equivalence sets
  - All memory references in the same set are aliases
  - e.g., {*a, b}, {*b, c, **a}
Introduction to interprocedural analysis

- Procedural abstraction
  - Key to structured programming
  - Makes analysis harder

Example:

```c
x = 5;
foo(p);
y = x+1;
```

Example:

```c
void f(int x) {
    if (x)
        foo();
    else
        bar();
}
```

Function calls and pointers

- Function calls can affect our points-to sets
  - e.g.,
    ```c
    p1 = &x;
p2 = &p1;
    ...
    foo();
    ```
  - `{{p1->x),(p2->p1}}`

- Be conservative → lose a lot of information
Interprocedural analysis

- Goal
  - Avoid making overly conservative assumptions about the effects of procedures and the state of the call sites

Terminology:

```c
int a, e; // Globals
void foo(int &b, &c) // Formal parameters (passed by reference)
{
    b = c;
}
main()
{
    int d; // Local variables
    foo(a, d); // Actual parameters
}
```