Alias Analysis

Last time
  – Interprocedural analysis

Today
  – Intro to alias analysis (pointer analysis)
Aliasing

What is aliasing?
- When two expressions denote the same mutable memory location
- e.g., \( p = \text{new Object}; \)
  \[ q = p; \quad \Rightarrow \quad *p \text{ and } *q \text{ alias} \]

How do aliases arise?
- Pointers
- Call by reference (parameters can alias each other or non-locals)
- Array indexing
- C \textbf{union}, Pascal variant records, Fortran \texttt{EQUIVALENCE} and \texttt{COMMON} blocks
Aliasing Examples

Pointers \((e.g., \text{in C})\)

```
int *p, i;
p = &i;
```

*\(p\) and \(i\) alias

Parameter passing by reference \((e.g., \text{in Pascal})\)

```
procedure proc1(var a:integer; var b:integer);
\ldots
proc1(x,x);
proc1(x,glob);
```

\(a\) and \(b\) alias in body of \textit{proc1}

\(b\) and \textit{glob} alias in body of \textit{proc1}

Array indexing \((e.g., \text{in C})\)

```
int i,j, a[128];
i = j;
```

\(a[i]\) and \(a[j]\) alias
What Can Alias?

Stack storage and globals

```c
void fun(int p1) {
    int i, j, temp;
    ...
}
```

Heap allocated objects

```c
n = new Node;
n->data = x;
n->next = new Node;
...
```
What Can Alias? (cont)

Arrays

for (i=1; i<=n; i++) {
    b[c[i]] = a[i];
}

Can c[i₁] and c[i₂] alias?

Fortran

Java

do b[c[i₁]] and b[c[i₂]] alias for any two iterations i₁ and i₂?
Alias Analysis

Goal: Statically identify aliases
- Can memory reference m and n access the same state at program point p?
- What program state can memory reference m access?

Why is alias analysis important?
- Many analyses need to know what storage is read and written
e.g., available expressions (CSE)
  \[*p = a + b;*\]
  \[y = a + b;\]
  If *p aliases a or b, the second expression is not redundant (CSE fails)

- e.g., Reaching definitions (constant propagation)
  \[d_1: \ x = 3;\]
  \[d_2: \ *p = 4;\]
  \[d_3: \ y = x;\]
  If *p aliases x, d_2 reaches this point;
  otherwise, both d_1 and d_2 reach

Otherwise we must be very conservative
Trivial Alias Analyses

Easiest approach
- Assume that nothing *must* alias
- Assume that everything *may* alias everything else
- Yuck!

Address taken: A slightly better approach (for C)
- 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
  
  e.g.,
  
  ```
  p = &a;
  . . .
  a = 3; b = 4;
  *q = 5;
  ```

  *q* and *a* may alias, so *a* may be 3 or 5, but *q* does not alias *b*, so *b* is 4

Enhance with type information?
Flow and Context Sensitive Analysis

Maintain points-to relations with context and flow info

- \( p_{cs} \rightarrow \{ x, y \} \) indicates that the pointer \( p \) contains the address of \( x \) and \( y \) when in the \( c \)th static call to the containing procedure and at statement \( s \)

Procedure calls

- Insert constraints for copying parameters and return value

Base constraints

- Used to initialize the points-to sets
- Ex: \( a := \&b \)
- Not needed after initialization

Simple constraints

- Involve variable names only
  Ex: \( c := a \)

Complex constraints

- Involve pointer dereferences
  Ex: \( *a := c \)
Flow-sensitive context-sensitive (FSCS)

```c
int** foo(int **p, **q)
{
    int **x;
    x = p;
    . . .
    x = q;
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}
```
Flow-sensitive context-insensitive (FSCI)

```c
int** foo(int **p, **q)
{
    int **x;

    x = p;  // p → {b, d}
    . . .  // Weak update
    x = q;  // q → {f, g}
    return x;
}

int main()
{
    int **a, *b, *d, *f, *c, *e;

    a = foo(&b, &f);  // a1 → {f, c}
    *a = &c;  // a2 →
    a = foo(&d, &g);  // Weak update)
    *a = &e;  // Weak update)
    . . .

```
FICS Example

Flow-insensitive context-sensitive (FICS)

```c
int** foo(int **p, **q)
{
    int **x;
    x = p;
    ...  
    x = q;
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}
```
FICI Example

Flow-insensitive context-insensitive (FICI)

```c
int** foo(int **p, **q)
{
    int **x;

    x = p;
    ...  
    x = q;
    return x;
}

t main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}
```
Flow-Insensitive and Context-Insensitive Pointer Analysis

The defining characteristics

– Ignore the control-flow graph, and assume that statements can execute in any order
– Rather than producing a solution for each program point, produce a single solution that is valid for the whole program

Flow-insensitive and Context-Insensitive pointer analyses

– **Andersen-style analysis**: the slowest and most precise
– **Steensgaard analysis**: the fastest and least precise
– All other flow-insensitive pointer analyses are hybrids of these two
Andersen 94

Overview
- Uses subset constraints
- Cubic complexity in program size, $O(n^3)$

Characterization of Andersen
- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling?
- Aggregate modeling: fields

source: Barbara Ryder’s Reference Analysis slides
Overview
- Uses unification constraints
- Almost linear in terms of program size
- Uses fast union-find algorithm
- Imprecision from merging points-to sets

Characterization of Steensgaard
- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling: none
- Aggregate modeling: possibly

source: Barbara Ryder’s Reference Analysis slides
Andersen vs. Steensgaard

```c
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

**Andersen-style analysis**

```
a ----> b ----> c
  |     |     |
  |     |     |
  d ----> e
```

Due to statement 4:

```
a ----> b ----> c
  |     |     |
  |     |     |
  d ----> e
```

**Steensgaard analysis**

```
a ----> b ----> c
  |     |     |
  |     |     |
  d ----> e
```

Due to statement 4:

```
a ----> b ----> c
  |     |     |
  |     |     |
  d ----> e
```
How hard is this problem?

Undecidable
– Landi 1992
– Ramalingan 1994

All solutions are conservative approximations

Is this problem solved?
– Why haven’t we solved this problem? [Hind 2001]
– Still a number of open issues
  – large programs
  – partial programs
  – modeling the heap (shape analysis)
  – ...

Concepts

What is aliasing and how does it arise

Performing alias analysis by hand

– Flow sensitive and context sensitive (FSCS)
– Flow sensitive and context insensitive (FSCI)
– Flow insensitive and context sensitive (FICS)
– Flow insensitive and context insensitive (FICI)

Pointer analysis is still not a fully solved problem
Next Time

Lecture
  – Analysis with datalog