Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopes

Aliases vs Optimizations

- Example:

  p.x := 5; q.x := 7; a := p.x;

  - Does reaching definition analysis show that the definition of p.x reaches a?
  - (Or: do p and q refer to the same variable/object?)
  - (Or: *can* p and q refer to the same thing?)

Aliases vs Optimizations

- Example

```
void f(int *p, int *q) {
  *p = 1; *q = 2;
  return *p;
}
```

  - How do we account for the possibility that p and q might refer to the same thing?
  - Safe approximation: since it's possible, assume it is true (but rules out a lot)

Types and Aliases (1)

- In Java, ML, MiniJava, and others, if two variables have incompatible types they cannot be names for the same location
  - Also helps that programmer cannot create arbitrary pointers to storage in these languages

Types and Aliases (2)

- Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class)
- Implication: need to propagate type information from the semantics pass to optimizer
  - Not normally true of a minimally typed IR
- Items in different alias classes cannot refer to each other

Aliases and Flow Analysis

- Idea: Base alias classes on points where a value is created
  - Every new/malloc and each local or global variable whose address is taken is an alias class
  - Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  - Use to calculate "may alias" information (e.g., p "may alias" q at program point s)

Using "may-alias" information

- Treat each alias class as a "variable" in dataflow analysis problems
- Example: framework for available expressions
  - Given statement s: M[a]:=b,

    gen[s] = { }

    kill[s] = { M[x] | a may alias x at s }

May-Alias Analysis

- Without alias analysis, #2 kills M[t] since x and t might be related
- If analysis determines that "x may-alias t" is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

- Code

M[p]
mem

```
1: u := M[t]
2: M[x] := r
3: w := M[t]
4: b := u+w
```

Where are we now?

- Dataflow analysis is the core of classical optimizations
- Still to explore:
  - Discovering and optimizing loops
  - SSA – Static Single Assignment form