Pointer and Alias Analysis

Aliases:
two expressions that denote same mutable memory location

Introduced through

- pointers
- call-by-reference
- array indexing
- C unions, Fortran equivalence

Applications:

improved side-effect analysis:
if assign to one expression,
what other expressions are modified?

- if certain, not a problem
- if uncertain, things can get tricky

automatic parallelization of code manipulating data structures

automatic reorganization of data structures

optimization of garbage collection strategies

An algorithm for computing aliases of pointers

At each program point, calculate set of $p \rightarrow x$ bindings,
if p points to x

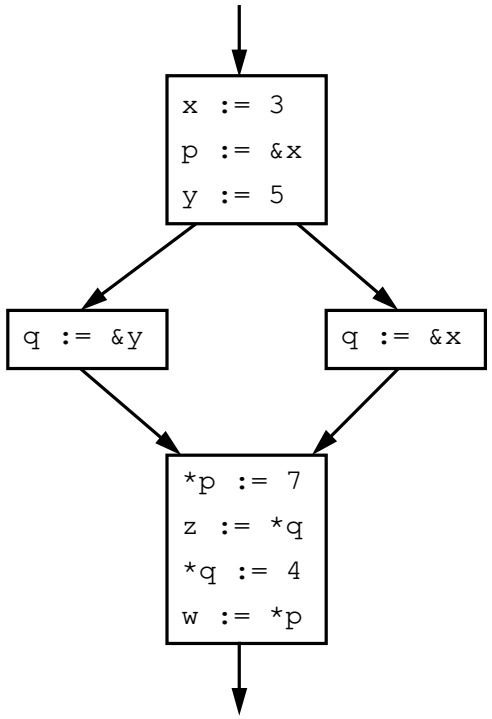
Two related problems:

- **may** aliases: p may point to x
- **must** aliases: p must point to x

Develop algorithm in increasing stages of complexity

- pointers only to scalars
- add pointers to pointers
- add pointers to array elements
- add pointers to record fields
- add pointers to dynamically-allocated storage

Example



Pointers to scalars

Flow functions:

`p := &x`

$OUT_{must} = IN_{must} - \{p \rightarrow *\} \cup \{p \rightarrow x\}$

$OUT_{may} = IN_{may} - \{p \rightarrow *\} \cup \{p \rightarrow x\}$

`p := q`

$OUT_{must} = IN_{must} - \{p \rightarrow *\} \cup \{p \rightarrow t \mid q \rightarrow t \in IN_{must}\}$

$OUT_{may} = IN_{may} - \{p \rightarrow *\} \cup \{p \rightarrow t \mid q \rightarrow t \in IN_{may}\}$

Meet functions:

must: intersection

may: union

Using alias info

E.g. reaching definitions

At each program point, calculate set of $s{:}x$ bindings,
if x might get its definition from stmt s

Flow functions:

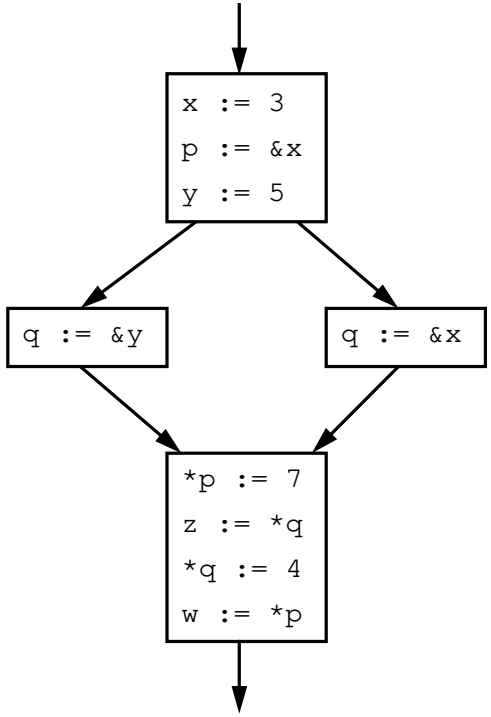
s: *p := x

$$OUT_{reach} = IN_{reach} - \{*{:}z \mid p \rightarrow z \in IN_{must}\} \cup \{s{:}z \mid p \rightarrow z \in IN_{may}\}$$

s: x := *p

$$OUT_{reach} = IN_{reach} - \{*{:}x\} \cup \{*{:}x \mid p \rightarrow z \in IN_{may} \wedge *{:}z \in IN_{reach}\}$$

Example



Adding pointers to record fields

```
p := &r.f
```

Treat each field as a distinct (scalar) variable

Adding aliases to array elements

Array index expressions can generate aliases:

`a[i]` aliases `b[j]` if
`a` aliases `b` and `i` equals `j`

Can have pointers to array elements:

```
p := &a[i]
```

Can have pointer arithmetic, for array addressing:

```
p := &a[0]; ...; p++
```

How to model arrays?

- could treat whole array as big monolithic location
- could try to reason about array index expressions
  ⇒ array dependence analysis (later)

Adding pointers to pointers

Flow functions:

`p := *q`

$$OUT_{must} = IN_{must} - \{p \rightarrow *\} \cup \{p \rightarrow t \mid q \rightarrow r \in IN_{must} \wedge r \rightarrow t \in IN_{must}\}$$

$$OUT_{may} = IN_{may} - \{p \rightarrow *\} \cup \{p \rightarrow t \mid q \rightarrow r \in IN_{may} \wedge r \rightarrow t \in IN_{may}\}$$

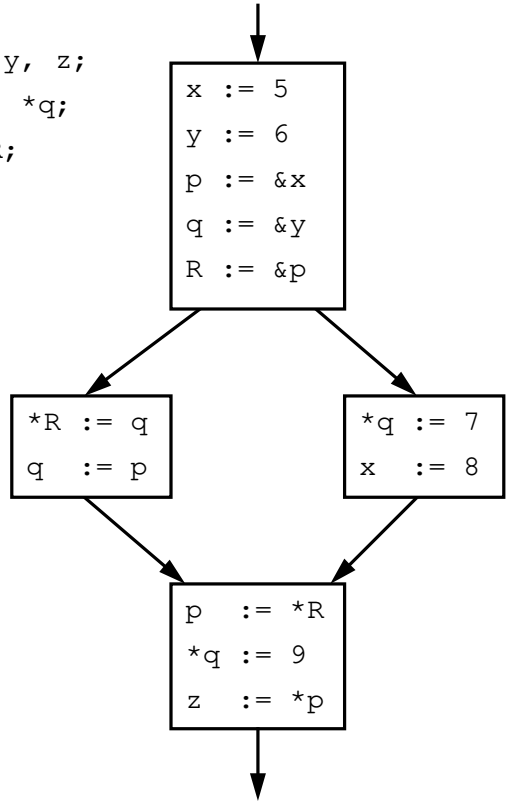
`*p := q`

$$OUT_{must} = IN_{must} - \{r \rightarrow * \mid p \rightarrow r \in IN_{must}\} \cup \{r \rightarrow t \mid p \rightarrow r \in IN_{must} \wedge q \rightarrow t \in IN_{must}\}$$

$$OUT_{may} = IN_{may} - \{r \rightarrow * \mid p \rightarrow r \in IN_{must}\} \cup \{r \rightarrow t \mid p \rightarrow r \in IN_{may} \wedge q \rightarrow t \in IN_{may}\}$$

Example

```
int x, y, z;
int *p, *q;
int **R;
```



Adding pointers to dynamically-allocated memory

`p := new T(`x_1`, ..., `x_n`)`

Issue: each time executed, creates a new location

- can't create a new variable each time analyze statement
- can't have single variable stand for all dynamic locations

Simple solution:

create a special summary node for each new expr

`s: p := new T(`x_1`, ..., `x_n`)`

$OUT_{must} = IN_{must} - \{p \rightarrow *\}$

$OUT_{may} = IN_{may} - \{p \rightarrow *\} \cup \{p \rightarrow new_s\}$

Alternatives:

- summary node for each type `T`
- k-limited summary
  - maintain distinct nodes up to k links removed from root vars
- ...