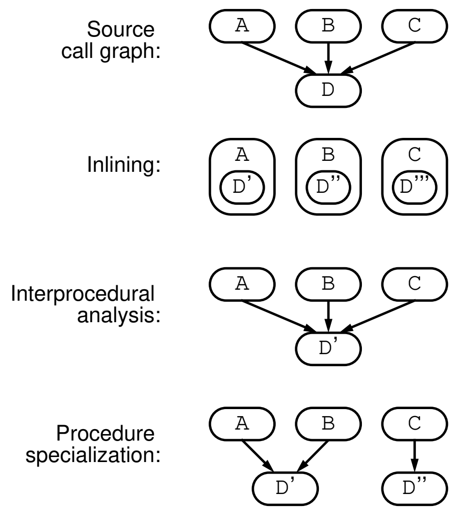
Procedure specialization

A.k.a. procedure cloning, customization

Halfway between inlining and interprocedural analysis



Abstract process

Given set of call sites of procedure `P`

e.g. $\{c_1, c_2, c_3, c_4, c_5\}$

Partition into equivalence classes of “similar” call sites

e.g. $\{\{c_1, c_2\}, \{c_3, c_4\}, \{c_5\}\}$

Copy `P` for each class, change calls accordingly

Do interprocedural analysis on changed call graph

Versus inlining:

+ less code explosion
+ works in presence of recursion

Versus interprocedural analysis:

+ better optimization

Question: how to do partitioning?

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 modified or not modified, not a problem
  - if uncertain, things can get tricky
- eliminate redundant loads/stores & dead stores
- 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 array elements
- add pointers to pointers
- add pointers to dynamically-allocated storage

Pointers to scalars

Flow functions:

p := &x

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

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

p := q

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

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

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

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

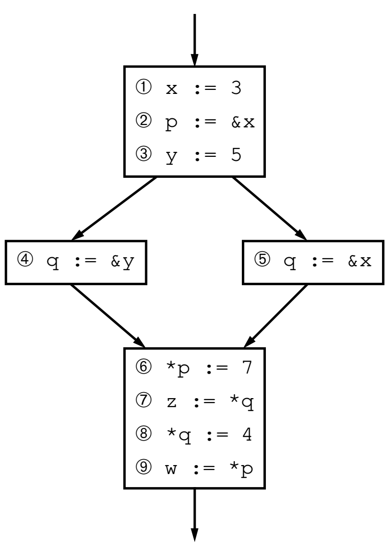
s: x := *p

$$OUT_{reach} = IN_{reach} - \{*{:}x\} \cup \{s{:}x\}$$

better (for some applications):

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

Example



Adding pointers 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

$$\mathrm{OUT}_{must} = \mathrm{IN}_{must} - \{p{\rightarrow}*\} \cup \{p{\rightarrow}t \mid q{\rightarrow}r \in \mathrm{IN}_{must} \wedge r{\rightarrow}t \in \mathrm{IN}_{must}\}$$

$$\mathrm{OUT}_{may} = \mathrm{IN}_{may} - \{p{\rightarrow}*\} \cup \{p{\rightarrow}t \mid q{\rightarrow}r \in \mathrm{IN}_{may} \wedge r{\rightarrow}t \in \mathrm{IN}_{may}\}$$

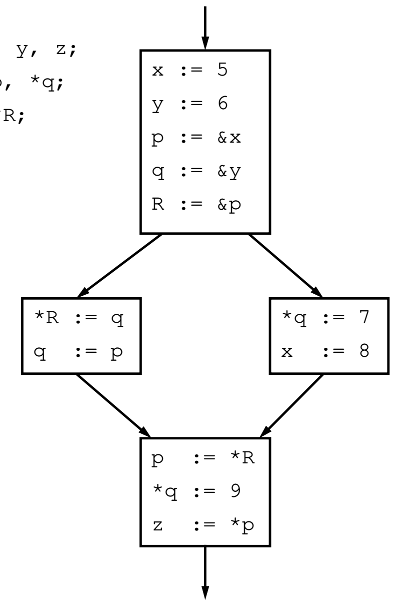
***p := q**

$$\mathrm{OUT}_{must} = \mathrm{IN}_{must} - \{r{\rightarrow}* \mid p{\rightarrow}r \in \mathrm{IN}_{must}\} \cup \{r{\rightarrow}t \mid p{\rightarrow}r \in \mathrm{IN}_{must} \wedge q{\rightarrow}t \in \mathrm{IN}_{must}\}$$

$$\mathrm{OUT}_{may} = \mathrm{IN}_{may} - \{r{\rightarrow}* \mid p{\rightarrow}r \in \mathrm{IN}_{must}\} \cup \{r{\rightarrow}t \mid p{\rightarrow}r \in \mathrm{IN}_{may} \wedge q{\rightarrow}t \in \mathrm{IN}_{may}\}$$

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 location each time analyze statement

- not an idempotent flow function!

Simple solution:
create a special summary node for each `new` stmt

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

$$\mathrm{OUT}_{must} = \mathrm{IN}_{must} - \{p{\rightarrow}*\}$$

$$\mathrm{OUT}_{may} = \mathrm{IN}_{may} - \{p{\rightarrow}*\} \cup \{p{\rightarrow}\mathrm{new}_s\}$$

Alternatives:

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