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

Pointers to scalars

Flow functions:

`p := &x`

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

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

`p := q`

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

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

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:

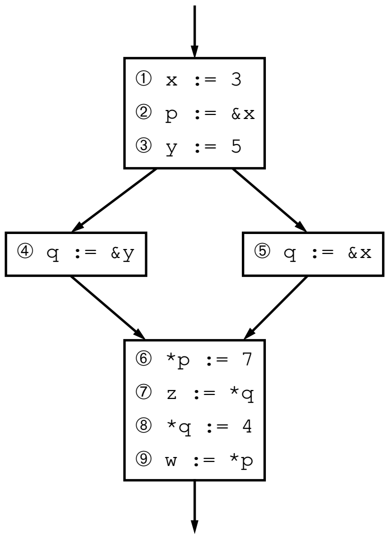
`s: *p := x`

$\mathrm{OUT}_{reach} = \mathrm{IN}_{reach} - \{*{:}z \mid p \rightarrow z \in \mathrm{IN}_{must}\} \cup \{s{:}z \mid p \rightarrow z \in \mathrm{IN}_{may}\}$

`s: x := *p`

$\mathrm{OUT}_{reach} = \mathrm{IN}_{reach} - \{*{:}x\} \cup \{*{:}x \mid p \rightarrow z \in \mathrm{IN}_{may} \wedge *{:}z \in \mathrm{IN}_{reach}\}$

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`

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

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

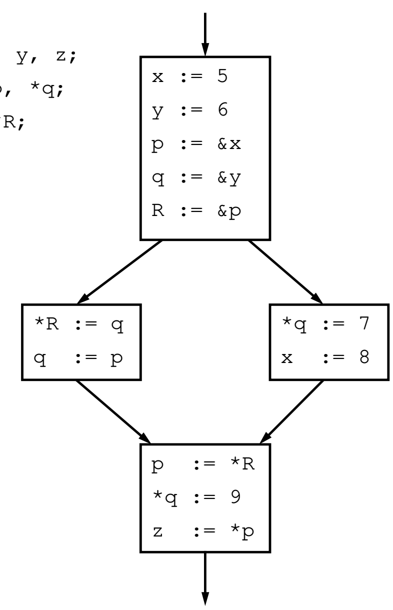
`*p := q`

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

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

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)`

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

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

Alternatives:

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

Interprocedural pointer analysis for C

[Wilson & Lam 95]

Key problems:

- how to represent pointer info in presence of casts, ptr arithmetic, etc.?
- how to perform analysis interprocedurally, maximizing benefit at reasonable cost?

Pointer representation

Ignore static type information,
since casts can violate it

Ignore subobject boundaries,
since pointer arithmetic can cross them

- assume malloc returns a block, whose boundaries are safe

Solution: location set = (*block*, *offset*, *stride*)

- represent all memory locations $\{offset + i * stride \mid i \in \text{Ints}\}$
- if stride = 0, then precise info
- if stride = 1, then only know block
- simple ptr arithmetic updates offset

Examples:

Expression	Location Set
`scalar`	(scalar, 0, 0)
`struct.F`	(struct, *offsetof*(F), 0)
`array[i]`	(array, 0, *sizeof*(array[i]))
`array[i].F`	(array, *offsetof*(F), *sizeof*(array[i]))
`*(&p + X)`	(p, 0, 1)

Interprocedural analysis

Caller → callee:
analyze callee given pointer relationships of formals

Callee → caller:
update pointer relationships after call returns

Supergraph-based approach:

+ simple
– may be too expensive
– smears effects of callers together,
hurting results after call returns

Context-sensitive interprocedural analysis

Idea: reanalyze callee for each distinct caller
(a kind of procedure specialization, but only at analysis time)

+ avoids smearing among direct callers
  (but smears across indirect callers)
– may do unnecessary work

Extension: reanalyze callee for *k* levels of calling context

+ less smearing
– more unnecessary work

Ultimate: reanalyze callee for each distinct calling path
[Emani *et al.* 94]

+ avoids all smearing
– probably ridiculously expensive (exponential time at least)
– recursion?

Partial transfer functions

Instead of fixed *k* of reanalysis,
reanalyze only for distinct caller effects

Can model analysis of callee as
a function from input aliases to output aliases

Represent as a set of ordered pairs
(input aliases, output aliases)

Each caller computes its input aliases (in callee terms),
then queries ordered pairs for match

- if not found, then do analysis of callee and add
  results to callee's ordered pairs

+ avoids smearing
+ reuse results of other callers where possible
  to save time
– worst-case: O(N * sizeof(set of aliases domain))

Experimental results

For C programs < 5K lines,
analysis time was < 16 seconds and
avg # of analyses per fn was < 1.4

Analysis results were used to better parallelize 2 C programs

Question: with bigger C programs,
how will avg # of analyses per fn grow?