Pointer and Alias Analysis

Aliases:
two expressions that denote same mutable memory location

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

Applications of alias analysis:
- 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 ugly
- eliminate redundant loads/stores & dead stores
  (CSE & dead assign elim, for pointer ops)
- automatic parallelization of code
  manipulating data structures
- ...

Kinds of alias info

Points-to analysis
- at each program point, calculate set of $p \to x$ bindings,
  if p points to x
- two variations:
  - **may** points-to: p might point to x
  - **must** points-to: p definitely points to x

Alias-pair analysis
- at each program point, calculate set of ($\texttt{expr}_1$,$\texttt{expr}_2$)
  pairs, if $\texttt{expr}_1$ and $\texttt{expr}_2$ reference the same memory
- **may** and **must** alias-pair versions

+ can handle aliasing of variables, unlike pts-to analysis

– potentially infinite number of alias pairs,
  so want the “minimal” set

Storage shape analysis
- at each program point, calculate an abstract description of
  the structure of pointers etc., e.g. list-like, or tree-like, or
  DAG-like, or ...

A points-to analysis

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

Outline:
- define **may** version first, then consider **must** version
- develop algorithm in increasing stages of complexity
  - pointers only to vars of scalar type
  - add pointers to pointers
  - add pointers to and from structures
  - add pointers to dynamically-allocated storage
  - add pointers to array elements

May-point-to scalars

Domain: Pow(*Var* × *Var*)
- each variable may point to any number of other variables
- may-point-to$_{PP}(P)$ = $\{ X \mid P \to X \in$ Soln(MayPT,*PP*) $\}$

Forward flow functions:

$\text{MayPT}_{P\ :=\ \&X}(\text{in}) = \text{in} - \{P \to *\} \cup \{P \to X\}$

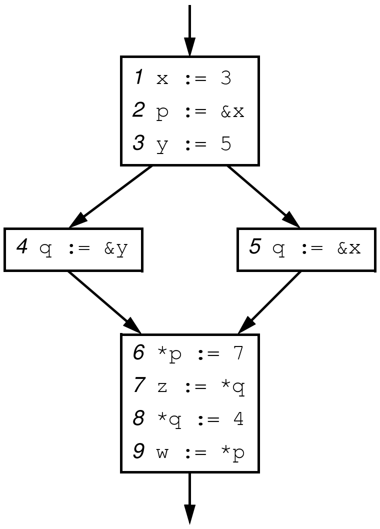
$\text{MayPT}_{P\ :=\ Q}(\text{in}) = \text{in} - \{P \to *\} \cup \{P \to Y \mid Q \to Y \in \text{in}\}$

$\text{MayPT}_{X\ :=\ *P}(\text{in}) = \text{in}$ (assuming P can't point to a ptr)
$\text{MayPT}_{*P\ :=\ X}(\text{in}) = \text{in}$ (assuming P can't point to a ptr)

Meet function: union

What about `nil`?

Example



Must-point-to

How to define must-point-to analysis?

Option 1: analogous to may-point-to, but as must problem

- meet function: intersection

Option 2: interpretation of may-point-to results

- if P may point only to X, then P must point to X, i.e.,

  must-point-to$_{PP}(P)$ = $\{ X \mid \{X\} =$ **may**-point-to$_{PP}(P) \}$

- what if P may point to `nil`? P assigned an integer?

Using alias info

E.g. reaching definitions

At each program point, calculate set of $X \rightarrow S$ bindings,
if X might get its definition from stmt S

Simple flow functions:

$RD_{S:\ X\ :=\ \ldots}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow S\}$

$RD_{S:\ *P\ :=\ \ldots}(in) = in - \{X \rightarrow * \mid X \in$ **must**-point-to$(P)\}$
$\cup \{X \rightarrow S \mid X \in$ **may**-point-to$(P)\}$

Reaching "right hand sides"

A variation on reaching definitions
that skips through trivial copies

$X \rightarrow S$ in set if X might get its definition from rhs of stmt S,
skipping through trivial variable and pointer copies where possible

Can use reaching right-hand sides to construct def/use chains
that skip through copies, e.g. for better constant propagation

Additional flow functions:

$RD_{S:\ X\ :=\ Y}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow S' \mid Y \rightarrow S' \in in\}$

$RD_{S:\ X\ :=\ *P}(in) = in - \{X \rightarrow *\}$
$\cup \{X \rightarrow S' \mid Y \in$ may-point-to$(P) \wedge$
$Y \rightarrow S' \in in\}$

$RD_{S:\ *P\ :=\ Y}(in) = in - \{X \rightarrow * \mid X \in$ **must**-point-to$(P)\}$
$\cup \{X \rightarrow S' \mid X \in$ **may**-point-to$(P) \wedge$
$Y \rightarrow S' \in in\}$

Another use: "scalar replacement"

If we know that a pointer expression $*P$ aliases a variable X (P must point to X) at some point, then can replace $*P$ with X

- both for load & store

Example:

```
a := 5
...
w := &a
...
b := *w
```


Adding pointers to pointers

Now allow a pointer to point to a pointer

- loads may return pointers, stores may store pointers

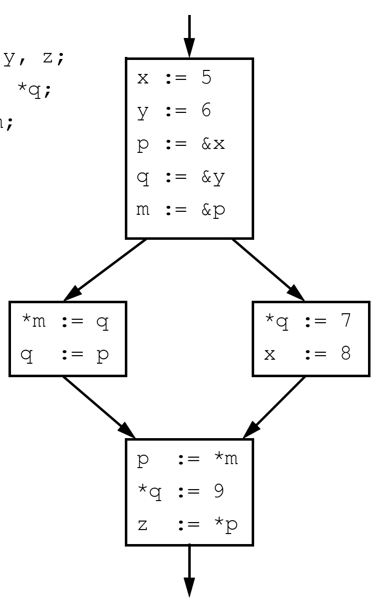
Revised flow functions for loads and stores:

$$\text{MayPT}_{P\ :=\ *Q}(\text{in}) = \text{in} - \{P \to *\} \cup \{P \to X \mid Q \to R \in \text{in} \wedge R \to X \in \text{in}\}$$

$$\text{MayPT}_{*P\ :=\ Q}(\text{in}) = \text{in} - \{R \to * \mid \{R\} = \text{in}(P)\} \cup \{R \to X \mid P \to R \in \text{in} \wedge Q \to X \in \text{in}\}$$

Example

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



Adding pointers to structs/records/objects/...

A variable can be a structure with a collection of named fields

- a pointer can point to a field of a structure variable
- a field can hold a pointer

Introduce location domain: $Loc = Var \cup Loc \times Field$

- either a variable or a location followed by a field name

Old PT domain: sets of $v_1 \to v_2$ pairs = $\text{Pow}(Var \times Var)$

New PT domain: sets of $l_1 \to l_2$ pairs = $\text{Pow}(Loc \times Loc)$

Some new forward flow functions:

$$\text{MayPT}_{P\ :=\ \&X.F}(\text{in}) = \text{in} - \{P \to *\} \cup \{P \to X.F\}$$

$$\text{MayPT}_{P\ :=\ X.F}(\text{in}) = \text{in} - \{P \to *\} \cup \{P \to L \mid X.F \to L \in \text{in}\}$$

$$\text{MayPT}_{P\ :=\ (*Q).F}(\text{in}) = \text{in} - \{P \to *\} \cup \{P \to L \mid Q \to R \in \text{in} \wedge R.F \to L \in \text{in}\}$$

$$\text{MayPT}_{X.F\ :=\ Q}(\text{in}) = \text{in} - \{X.F \to *\} \cup \{X.F \to L \mid Q \to L \in \text{in}\}$$

$$\text{MayPT}_{(*P).F\ :=\ Q}(\text{in}) = \text{in} - \{R.F \to * \mid \{R\} = \text{in}(P)\} \cup \{R.F \to L \mid P \to R \in \text{in} \wedge Q \to L \in \text{in}\}$$

Adding pointers to dynamically-allocated memory

`P := new τ`

- τ could be scalar, pointer, structure, ...

Issue: each execution of `new` creates a new location

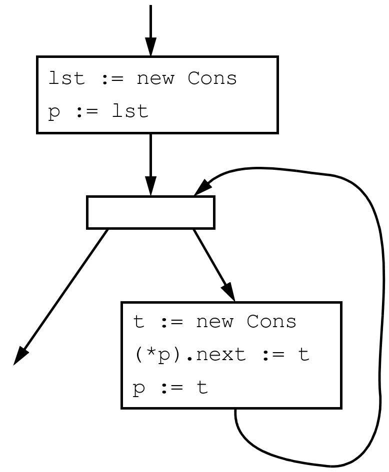
Idea: introduce new set of possible memory locations: *Mem*

Extend *Loc* to also allow a location to be a *Mem*:
Loc = *Var* ∪ ***Mem*** ∪ *Loc*×*Field*

Flow function:
$\text{MayPT}_{P\ :=\ \text{new}\ \tau}(\text{in}) = \text{in} - \{P\to^*\} \cup \{P\to\text{'}\mathit{newvar}\text{'}\}$

- *newvar*: return next unallocated element of *Mem*

Example



A monotonic, finite approximation

Can't allocate a new memory location
each time analyze `new` statement

- infinite *Mem* ⇒ infinite *Loc* ⇒ infinitely tall Pow(*Loc* × *Loc*)!
- not a monotonic flow function!

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

- *Loc* = *Var* ∪ ***Stmt*** ∪ *Loc*×*Field*

Fixed flow function:
$\text{MayPT}_{S:\ P\ :=\ \text{new}\ \tau}(\text{in}) = \text{in} - \{P\to^*\} \cup \{P\to S\}$

Summary nodes represent a *set* of possible locations
⇒ cannot strongly update a summary node
$\text{MayPT}_{*P\ :=\ Q}(\text{in}) = \text{in} - \{R\to^* \mid \{R\} = \text{in}(P) \wedge R \notin \mathbf{Stmt}\} \cup \{R\to X \mid P\to R \in \text{in} \wedge Q\to X \in \text{in}\}$

Alternative summarization strategies:

- summary node for each type τ
- *k*-limited summary
  - maintain distinct nodes up to *k* links removed from root vars, then summarize together

Adding pointers to array elements

Array index expressions can generate aliases:
`a[i]` aliases `b[j]` if:

- `a` aliases `b` and `i` equals `j`
- more generally, `a` and `b` overlap, and `&a[i]` = `&b[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?

Option 1: reason about array index expressions
⇒ array dependence analysis

Option 2: use a summary node to stand for all elements of the array