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
- automatic reorganization of data structures
- optimization of garbage collection strategies

Kinds of alias info

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

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

Shape analysis
- at each program point, calculate an abstract description of
  the structure of pointers etc.

Points-to analysis is simple

Alias-pairs analysis more general than points-to analysis,
but more complicated

Shape analysis more abstract

An intraprocedural points-to analysis

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

Define both **may** and **must** versions

Develop algorithm in increasing stages of complexity
- pointers only to scalars
- add pointers to pointers
- add pointers to dynamically-allocated storage
- add pointers to array elements

Pointers to scalars

Domain: Pow(Var $\times$ Var)

Flow functions:

p := &x

$\mathrm{MUST\text{-}PT}_{succ} = \mathrm{MUST\text{-}PT}_{pred} - \{p \rightarrow *\} \cup \{p \rightarrow x\}$

$\mathrm{MAY\text{-}PT}_{succ} = \mathrm{MAY\text{-}PT}_{succ} - \{p \rightarrow *\} \cup \{p \rightarrow x\}$

p := q

$\mathrm{MUST\text{-}PT}_{succ} = \mathrm{MUST\text{-}PT}_{pred} - \{p \rightarrow *\} \cup \{p \rightarrow t \mid q \rightarrow t \in \mathrm{MUST\text{-}PT}_{pred}\}$

$\mathrm{MAY\text{-}PT}_{succ} = \mathrm{MAY\text{-}PT}_{succ} - \{p \rightarrow *\} \cup \{p \rightarrow t \mid q \rightarrow t \in \mathrm{MAY\text{-}PT}_{succ}\}$

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

Simple flow functions:

s: *p := x

$$\begin{aligned} RD_{succ} = RD_{pred} \; & - \; \{*{:}z \mid p \to z \in \text{MUST-PT}_{pred}\} \\ & \cup \; \{s{:}z \mid p \to z \in \text{MAY-PT}_{succ}\} \end{aligned}$$

s: x := *p

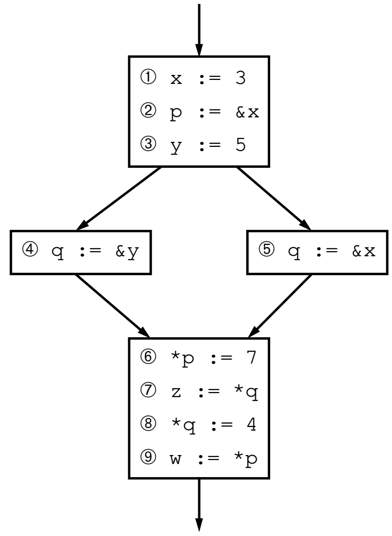
$$\begin{aligned} RD_{succ} = RD_{pred} \; & - \; \{*{:}x\} \\ & \cup \; \{s{:}x\} \end{aligned}$$

Better flow function (for some uses):
follow pointers to "original" r.h.s.

s: x := *p

$$\begin{aligned} RD_{succ} = RD_{pred} \; & - \; \{*{:}x\} \\ & \cup \; \{*{:}x \mid p \to z \in \text{MAY-PT}_{succ} \wedge \\ & \qquad *{:}z \in RD_{pred}\} \end{aligned}$$

Example



Adding pointers to pointers

Flow functions:

p := *q

$$\begin{aligned} \text{MUST-PT}_{succ} = \text{MUST-PT}_{pred} - \{p \to *\} \; \cup & \\ \{p \to t \mid\; & q \to r \in \text{MUST-PT}_{pred} \wedge \\ & r \to t \in \text{MUST-PT}_{pred}\} \end{aligned}$$

$$\begin{aligned} \text{MAY-PT}_{succ} = \text{MAY-PT}_{succ} - \{p \to *\} \; \cup & \\ \{p \to t \mid\; & q \to r \in \text{MAY-PT}_{succ} \wedge \\ & r \to t \in \text{MAY-PT}_{succ}\} \end{aligned}$$

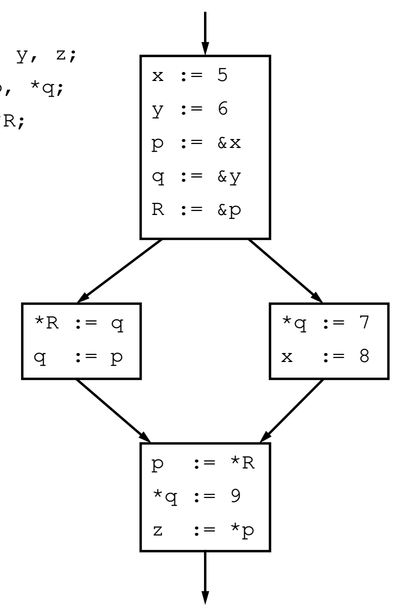
***p := q**

$$\begin{aligned} \text{MUST-PT}_{succ} = \text{MUST-PT}_{pred} & \\ - \; \{r \to * \mid\; & p \to r \in \text{MUST-PT}_{pred}\} \\ \cup \; \{r \to t \mid\; & p \to r \in \text{MUST-PT}_{pred} \wedge \\ & q \to t \in \text{MUST-PT}_{pred}\} \end{aligned}$$

$$\begin{aligned} \text{MAY-PT}_{succ} = \text{MAY-PT}_{succ} & \\ - \; \{r \to * \mid\; & p \to r \in \text{MUST-PT}_{pred}\} \\ \cup \; \{r \to t \mid\; & p \to r \in \text{MAY-PT}_{succ} \wedge \\ & q \to t \in \text{MAY-PT}_{succ}\} \end{aligned}$$

Example

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



Adding pointers to dynamically-allocated memory

```
p := new T
```

Issue: each execution creates a new location

Idea: generate new var to stand for new location
- make Var domain unbounded
- *newvar*: return next unused element of Var

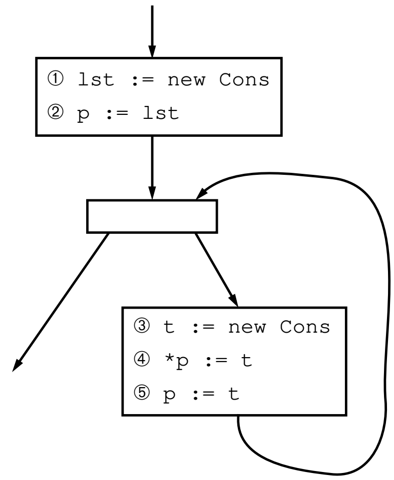
Flow function:

s: `p := new T`

$\text{MUST-PT}_{\text{succ}} = \text{MUST-PT}_{\text{pred}} - \{\text{p}\to*\} \cup \{\text{p}\to \textit{newvar}\}$

$\text{MAY-PT}_{\text{succ}} = \text{MAY-PT}_{\text{succ}} - \{\text{p}\to*\} \cup \{\text{p}\to \textit{newvar}\}$

Example



A monotonic, finite approximation

Can't create a new variable each time analyze statement
- lattice is infinitely tall if Var domain is infinite!
- not an idempotent flow function!

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

Domain = Pow((Var+Stmt) × (Var+Stmt))

s: `p := new T`

$\text{MUST-PT}_{\text{succ}} = \text{MUST-PT}_{\text{pred}} - \{\text{p}\to*\} \cup \{\text{p}\to \text{loc}_s\}$

$\text{MAY-PT}_{\text{succ}} = \text{MAY-PT}_{\text{succ}} - \{\text{p}\to*\} \cup \{\text{p}\to \text{loc}_s\}$

Alternatives:
- summary node for each type `T`
- *k*-limited summary
  - maintain distinct nodes up to *k* links removed from root vars, then merge 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`
- `a` and `b` overlap, and ...

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)