## 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 related problems:
  - **may** points-to: $p$ may point to $x$
  - **must** points-to: $p$ must point to $x$

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

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

Points-to analysis is simple

Storage shape analysis more abstract

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



## An intraprocedural 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 scalars
  - add pointers to pointers
  - add pointers to dynamically-allocated storage
  - add pointers to array elements



## May-point-to scalars

Domain: Pow(Var × Var)

Flow functions:

**p := &x**

$\text{MAY-PT}_{succc} = \text{MAY-PT}_{pred} - \{p \to *\} \cup \{p \to x\}$

**p := q**

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

Meet function: union

### Must-point-to

How to define must-point-to analysis?

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

- e.g. intersection is meet operation

Option 2: interpretation of may-point-to results

- if p may point to only x, then p must point to x:

$$\text{MUST-PT} = \{\ p\to x \mid p\to x \in \text{MAY-PT and } p\to y \in \text{MAY-PT} \Rightarrow y=x\}$$

- what if p points to `nil`? p assigned an integer?



### 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**

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

**s: x := *p**

$$RD_{succ} = RD_{pred} - \{*{:}x\} \cup \{s{:}x\}$$



### Reaching "right hand sides"

A variation on reaching definitions
that passes definitions through copies

$s{:}x$ in set if $x$ might get its definition from rhs of stmt $s$,
skipping through uninteresting copies and pointer loads
where possible

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

Flow functions:

**s: x := y**

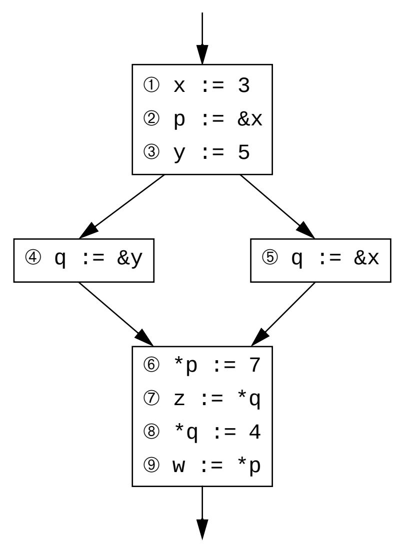
$$RD_{succ} = RD_{pred} - \{*{:}x\} \cup \{s'{:}x \mid s'{:}y \in RD_{pred}\}$$

**s: x := *p**

$$RD_{succ} = RD_{pred} - \{*{:}x\} \cup \{s'{:}x \mid p\to z \in \text{MAY-PT}_{pred} \wedge s'{:}z \in RD_{pred}\}$$



### Example



## Adding pointers to pointers

Flow functions:

**p := *q**

$$\text{MAY-PT}_{succ} = \text{MAY-PT}_{pred} - \{p \to *\} \cup \{p \to t \mid q \to r \in \text{MAY-PT}_{pred} \wedge r \to t \in \text{MAY-PT}_{pred}\}$$

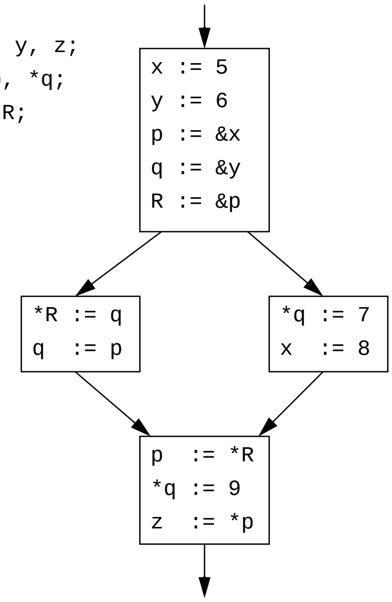
***p := q**

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



## 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

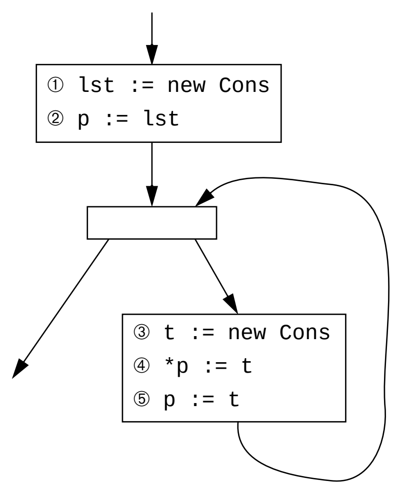
Flow function:

*s*: p := new T

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



## 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 a monotonic flow function!

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

Domain = Pow((Var+Stmt) $\times$ (Var+Stmt))

$s$: `p := new T`
`MAY-PT`$_{succ}$ = `MAY-PT`$_{pred}$ - {p→*} $\cup$ {p→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)