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

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

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 ...

Points-to analysis is simple

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

Storage shape analysis more abstract

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 $\times$ Var)

Forward flow functions:

$$PT_{\mathtt{p\ :=\ \&x}}(\mathrm{in}) = \mathrm{in} - \{p \to *\} \cup \{p \to x\}$$

$$PT_{\mathtt{p\ :=\ q}}(\mathrm{in}) = \mathrm{in} - \{p \to *\} \cup \{p \to v \mid q \to v \in \mathrm{in}\}$$

Meet function: union

What about `p := nil`?

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

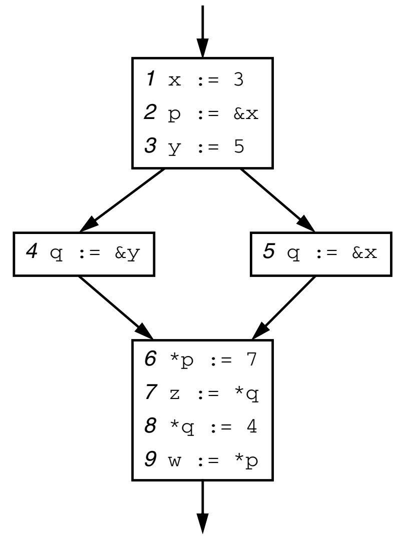
Option 2: interpretation of may-point-to results

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

  must-point-to(`p`) = { `x` | {`x`} = may-point-to(`p`)}

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

Example



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:

$RD_{s:\, x\ :=\ ...}(in) = in - \{x \rightarrow *\} \cup \{x \rightarrow s\}$

$RD_{s:\, *p\ :=\ ...}(in) = in - \{x \rightarrow * \mid \forall x \in \text{must-point-to}(p)\}$
$\cup \{x \rightarrow s \mid \forall x \in \text{may-point-to}(p)\}$

Reaching "right hand sides"

A variation on reaching definitions
that passes definitions through copies

$x \rightarrow s$ 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

Additional flow functions:

$RD_{s:\, x\ :=\ y}(in) = in - \{x \rightarrow *\} \cup \{x \rightarrow s' \mid y \rightarrow s' \in in\}$

$RD_{s:\, x\ :=\ *p}(in) = in - \{x \rightarrow *\}$
$\cup \{x \rightarrow s' \mid p \rightarrow y \in \text{may-point-to}(p) \wedge$
$y \rightarrow s' \in in\}$

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

Load part also known as "redundant load elimination"

Adding pointers to pointers

Now allow a pointer to point to a pointer

- loads may return pointers, stores may store pointers

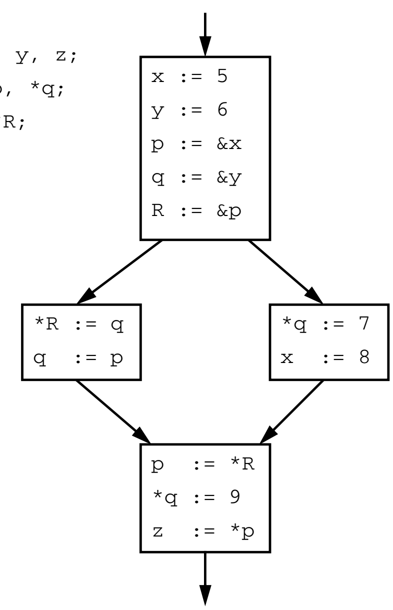
New flow functions:

$PT_{\texttt{p := *q}}(\text{in}) = \text{in} - \{\texttt{p}\to *\} \cup \{\texttt{p}\to v \mid \texttt{q}\to r \in \text{in} \wedge r\to v \in \text{in}\}$

$PT_{\texttt{*p := q}}(\text{in}) = \text{in} - \{r\to * \mid \{r\} = \text{in}(\texttt{p})\}$
$\quad \cup \{r\to v \mid \texttt{p}\to r \in \text{in} \wedge \texttt{q}\to v \in \text{in}\}$

Example

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



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 + Loc×Field

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

Old PT domain: sets of $v_1 \to v_2$ pairs = Pow(Var × Var)

New PT domain: sets of $l_1 \to l_2$ pairs = Pow(Loc × Loc)

Some new forward flow functions:

$PT_{\texttt{p := \&x.f}}(\text{in}) = \text{in} - \{\texttt{p}\to *\} \cup \{\texttt{p}\to \texttt{x.f}\}$

$PT_{\texttt{p := x.f}}(\text{in}) = \text{in} - \{\texttt{p}\to *\} \cup \{\texttt{p}\to l \mid \texttt{x.f}\to l \in \text{in}\}$

$PT_{\texttt{p := (*q).f}}(\text{in}) = \text{in} - \{\texttt{p}\to *\}$
$\quad \cup \{\texttt{p}\to l \mid \texttt{q}\to r \in \text{in} \wedge r\texttt{.f}\to l \in \text{in}\}$

$PT_{\texttt{x.f := q}}(\text{in}) = \text{in} - \{\texttt{x.f}\to *\} \cup \{\texttt{x.f}\to l \mid \texttt{q}\to l \in \text{in}\}$

$PT_{\texttt{(*p).f := q}}(\text{in}) = \text{in} - \{r\texttt{.f}\to * \mid \{r\} = \text{in}(\texttt{p})\}$
$\quad \cup \{r\texttt{.f}\to l \mid \texttt{p}\to r \in \text{in} \wedge \texttt{q}\to l \in \text{in}\}$

Adding pointers to dynamically-allocated memory

`p := new T`

- `T` could be scalar, pointer, structure, ...

Issue: each execution creates a new location

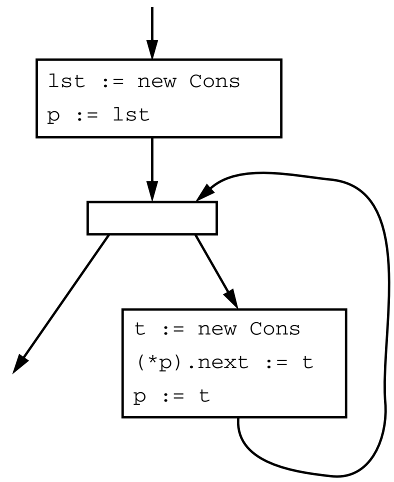
Idea: generate new var of type `T` to stand for new location

- make Var domain unbounded
- *newvar*: return next unused element of Var

Flow function:

$PT_{\texttt{p := new T}}(\text{in}) = \text{in} - \{\texttt{p}\rightarrow *\} \cup \{\texttt{p}\rightarrow \textit{newvar}\}$

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 a monotonic flow function!

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

- Loc = Var + **Stmt** + Loc×Field

Fixed flow function:

$PT_{s\texttt{: p := new T}}(\text{in}) = \text{in} - \{\texttt{p}\rightarrow *\} \cup \{\texttt{p}\rightarrow s\}$

Summary nodes abstract a set of possible locations
⇒ cannot strongly update a summary node

$PT_{\texttt{*p := q}}(\text{in}) = \text{in} - \{r\rightarrow * \mid \{r\} = \text{in}(\texttt{p}) \wedge \mathbf{r} \in \textbf{Loc}\}$
$\cup \{r\rightarrow v \mid \texttt{p}\rightarrow r \in \text{in} \wedge \texttt{q}\rightarrow v \in \text{in}\}$

Alternative summarization strategies:

- summary node for each type `T`
- *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