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 $p \rightarrow x$ 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

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 $p \rightarrow x$ 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

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

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

p := q

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

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

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

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

s: x := *p

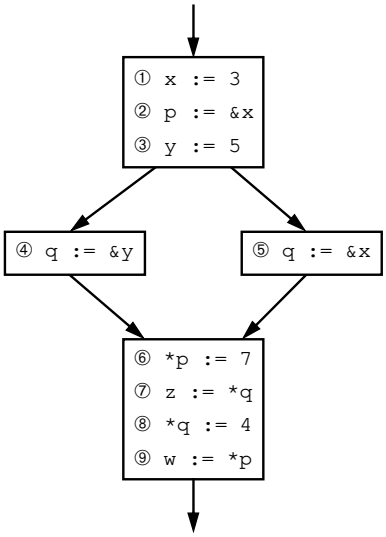
$$RD_{succ} = RD_{pred} - \{*{:}x\} \cup \{s{:}x\}$$

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

s: x := *p

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

Example



Adding pointers to pointers

Flow functions:

p := *q

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

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

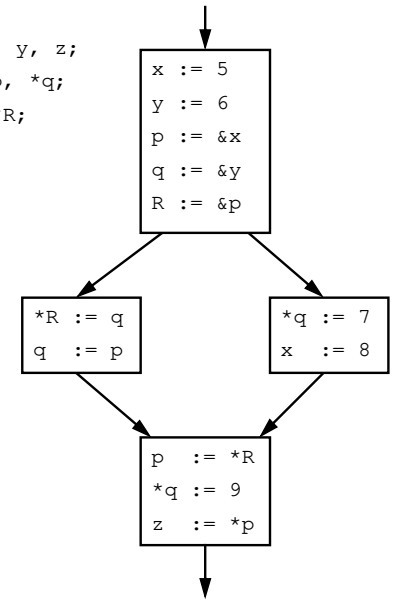
***p := q**

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

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

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

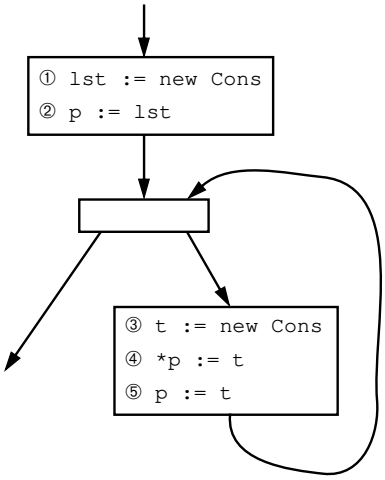
Flow function:

s: `p := new T`

$\texttt{MUST-PT}_{succ} = \texttt{MUST-PT}_{pred} - \{p \rightarrow *\} \cup \{p \rightarrow newvar\}$

$\texttt{MAY-PT}_{succ} = \texttt{MAY-PT}_{succ} - \{p \rightarrow *\} \cup \{p \rightarrow newvar\}$

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`

$\texttt{MUST-PT}_{succ} = \texttt{MUST-PT}_{pred} - \{p \rightarrow *\} \cup \{p \rightarrow loc_s\}$

$\texttt{MAY-PT}_{succ} = \texttt{MAY-PT}_{succ} - \{p \rightarrow *\} \cup \{p \rightarrow loc_s\}$

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)

Interprocedural pointer analysis for C

[Wilson & Lam 95]

A may point-to analysis

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

Treat memory as composed of blocks of bits

- each local, global variable is a block
- malloc returns a block

Block boundaries are safe

- casts, pointer arithmetic won't cross blocks

Location sets

A location set represents a set of memory locations within a block

Location set = (*block*, *offset*, *stride*)

- represent all memory locations {*offset* + i * *stride* | i ∈ Ints}
- if stride = 0, then precise info
- if stride = 1, then only know block
- simple pointer 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)

At each program point,
a pointer may point to a set of location sets

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

A.k.a. polyvariant analysis (vs. monovariant 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
– exponentially expensive
– 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
(a **transfer function**)

Represent function as a set of ordered pairs
(input alias pattern → output alias pattern)

Only represent those pairs that occur during analysis,
identified lazily (a **partial transfer function**)

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

Caller/callee mapping

To compute input context for a call site,
translate into terms of callee

Modeled in paper as **extended parameters**:

- each formal and referenced global,
each value referenced through a pointer from an
extended parameter gets a node

Goal: make input context as general as possible
(to be reusable across many call sites)

- represent abstract alias pattern from callee's perspective,
not direct copy of actual may/must aliases in caller
- only track alias pattern that's accessed by callee
(ignore irrelevant aliases)

Tricky details:

- constructing callee model of aliases from caller aliases
- checking new caller against existing callee input patterns
- mapping back from callee output pattern to real caller
aliases
- pointers to structs & struct members ("nested" pointers)

Example

```
int** global;
void P(int*** ap, int**** bp) {
  *ap = **bp;
  **bp = global;
}

void main() {
  int m = 5;
  int* d = &m;
  global = &d;
  int** x = &d;
  int*** xp = &x;
  int n = 6;
  int* e = &n;
  int** f = &e;
  int* h = &n;
  int** g = &h;
  int*** y;
  if (...) y = &f; else y = &g;
  int**** yp = &y;
  P(xp, yp);
}
```

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 two C programs

Questions:

- with bigger programs, how will # analyses per fn grow?
  i.e. how will analysis time scale?
- what is impact of alias info on other optimizations?

[Ruf 96]: for smallish C programs (< 15K lines),
context-*insensitive* alias analyses are just as effective as
context-*sensitive* ones

Almost-Linear-Time Pointer Analysis

[Steensgaard, POPL 96]

Goal: scale interprocedural analysis to million-line programs

- flow-sensitive, context-sensitive analysis too expensive
- aim for linear time analysis

Approach: treat alias analysis as a **type inference** problem
(inspired by a similar analysis by Heinglein [91])

- give each variable an associated "type variable"
  - each struct or array gets a single type variable
  - each alloc site gets a type variable
- make one linear pass through the entire program;
  whenever one var assigned to/computed from another,
  unify their type variables
  - near-constant-time unification using union/find data structures
- when done, all unified variables are **may** aliases,
  un-unified variables are disjoint

Details:

- don't do unification if assigning non-pointers
  (conditional join stuff in paper)
- pending list to enable one single pass through program

Example

```
void foo(int* a, int* b) {
  ... /* are a and b aliases? */ ...
}
int g;
void bar() {
  ...
  int* x = &g;
  int* y = new int;
  foo(x, y);
  ...
}
void baz(int* e, int* f) {
  ...
  int* i = ... ? e : f;
  int* j = new int;
  foo(i, j);
  ...
}
void boz(int* p, int* q) {
  ... /* are p and q aliases? */ ...
  baz(p, q);
}
```


Results

Analyze 75K-line program in 15 seconds,
25K-line program in 5.5 seconds

\+ fast

\+ linear time complexity

[Morgenthaler 95]:
do this analysis *during parsing*, for 50% extra cost

Quality of alias info?

- Steensgaard: pretty good, except for smearing struct
  elements together
  - another Steensgaard paper extends algorithm to avoid smearing
    struct elements together, but sacrifices near-linear-time
    bound

– no MUST alias info

Type inference is an intriguing framework for fast, coarse
program analysis

[DeFouw, Chambers, & Grove, POPL98]: for OO systems