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 pointer analysis

Caller → callee: analyze callee given pointer relationships of formals

Callee → caller: update pointer relationships after call returns

Option 1: supergraph-based, context-insensitive approach

+ simple
– may be too expensive
– smears effects of callers together, hurting results after call returns

Context-sensitive interprocedural analyses

Option 2: reanalyze callee for each distinct caller

+ avoids smearing among direct callers
  (but smears across indirect callers)
– may do unnecessary work

Option 3: reanalyze callee for *k* levels of calling context

+ less smearing
– more unnecessary work

Option 4: reanalyze callee for each distinct calling path
[Emani *et al.* 94, ...]

+ avoids all smearing
– cost is exponential in call graph depth
– recursion?

Partial transfer functions

Option 5: instead of fixed *k* of reanalysis,
reanalyze only for distinct caller effects

Model analysis of callee as a summary function
from input aliases to output aliases
(a transfer/flow function for the call node)

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

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

Compute pairs lazily

+ 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 from a call site,
translate into terms of callee

Modeled in paper as **extended parameters**:

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

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);
  P(y, &xp);
}
```

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 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 qux(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
(recent versions: Word97 (2.1Mloc) in 1 minute)

+ 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

[Das 00]:
extension with higher precision results that analyzes Word97
in 2 minutes

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

[DeFouw, Chambers, & Grove 98]: for OO systems