Interprocedural pointer analysis for C

[Wilson & Lam 95]

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

- assume malloc returns a block, whose boundaries are safe

Solution: location set = (*block*, *offset*, *stride*)

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

Interprocedural analysis

Caller $\rightarrow$ callee:
analyze callee given pointer relationships of formals

Callee $\rightarrow$ 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

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
– probably ridiculously expensive (exponential time at least)
– 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

- a **partial transfer function**

Each caller computes its input aliases,
converts to abstract callee terms,
then queries ordered pairs for match

- if not found, then do analysis of callee and add results to callee's ordered pairs
- convert result back from callee to caller terms

+ 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, need to 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:

- 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)

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