Interprocedural Constant Propagation

[Callahan, Cooper, Kennedy, & Torczon, PLDI 86]

Goal: for each procedure, for each formal, identify whether all calls of procedure pass a particular constant to the formal

- e.g. stride argument passed to LINPACK library routines

Sets up lattice-theoretic framework for solving problem

- store const-prop domain element for each formal
  - initialize all formals to T
- worklist-based algorithm to find interprocedural fixed-point:

```
worklist := {Main};
while worklist ≠ ∅ do
  proc := remove_any(worklist);
  process(proc);
end
process(proc) {
  foreach call site c in proc do
    compute c's actuals from proc's formals;
    c's callee's formals meet= c's actuals;
      if changed or first time,
        add callee to worklist;
}
```


Jump functions

How to quickly compute info at `c`'s actuals from `proc`'s formals?

Define *jump functions* to relate actual parameter at a call site to formal parameters of enclosing procedure

Different degrees of sophistication:

- **all-or-nothing**:
  only if actual is an intraprocedural constant
- **pass-through**:
  also, if formal a constant, then actual a constant
- **symbolic interpretation**:
  do full intraprocedural constant propagation

Can define similar jump functions for procedure results, too

- a total summary function for callers
- push callers on worklist if procedure's result info changes

No experimental results reported!

Interprocedural pointer analysis for C

[Wilson & Lam 95]

A may-point-to analysis
Copes with "full" C

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, at least not portably

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]))
`*(&a + x)`	(a, 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?

Context-sensitive analysis using partial transfer functions

Option 5: instead of fixed *k* of reanalysis,
reanalyze for each distinct calling points-to context

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

Represent function as a set of ordered pairs
(input points-to → output points-to)

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 as **extended parameters**:

- each formal and referenced global gets a node, as does each value referenced through a pointer

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

- represent abstract points-to pattern from callee's perspective, not direct copy of actual aliases in caller
  - treat extended parameter nodes as distinct iff caller nodes are distinct
- only track points-to pattern that's accessed by callee (ignore irrelevant points-to)

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)

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

Cheaper interprocedural pointer analyses

(All are context-insensitive)

Andersen's algorithm [94]: **flow-insensitive** points-to

- a single points-to graph for each procedure, as a whole

Vs. the flow-sensitive points-to algorithm from class:

- the flow-sensitive algorithm has a possibly distinct points-to graph at each program point
- the flow-insensitive points-to graph will be a superset of the union of each of these graphs
- use SSA form to retain effect of flow-sensitivity for local variables

Type-based alias analysis [Diwan *et al.* 98]: just use static types

- pointers of different static types without common subtypes cannot alias

+ "trivial", yet surprisingly effective

– restricted to statically-typed, type-safe languages with restricted multiple subtyping or whole-program knowledge

– may info only

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 Henglein [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 pointer var assigned to/computed from another, *unify* the type variables of their targets
  - near-constant-time unification using union/find data structures
- when done, all unified variables are **may**-aliases, un-unified variables are definitely non-aliasing

Details:

- don't do unification if assigning null or 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;  // alloc1
  foo(x, y);
  ...
}
void baz(int* e, int* f) {
  ...
  int* i = ... ? e : f;
  int* j = new int;  // alloc2
  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
(more 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

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

[Fahndrich *et al.* 00]: a context-sensitive extension

- "polymorphic type inference"

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

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