Alias Analysis

CS202 Compiler Construction
April 3, 2003

alias: a·li·as

n.
1. An assumed name: *The swindler worked under various aliases*.
2. *Electronics*. A false signal in telecommunication links from beats between signal frequency and sampling frequency.

adv.
1. Also known as; otherwise: *Johnson, alias Johns*.

Jeff Blaisdell
jblaisde@emba.uvm.edu

Topics Overview

- Aliasing
- Alias Analysis Problem
  - general dataflow
  - classifications and features
- Overview of Inter-procedural Analysis Approaches
  - analysis frameworks, algorithm basics, pros/cons
- Modular Inter-procedural Pointer Analysis Using Access Paths
  - a detailed look
- Inter-procedural Wrap-up / Future of Alias Analysis

What Is An Alias?

Pointers or variables *alias* each other if they point to the same *mutable* location in memory

- i.e., when there exists more then one way to access a storage location

Aliasing: an assumed name?

Lets look at a simple example:

- ptr1 is an alias for i:

```
int i;
int *ptr1 = &i;
```

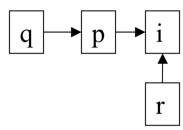
Aliases make code optimization much more difficult

- hard to guarantee whether a value is being changed

How are aliases represented?...

Alias Representation

Directed Alias Graph



- complete alias pairs

  <*q,p> <*p,i> <*r,i> <**q,*p>
  <**q,i> <*p,*r> <**q,*r>

- compact alias pairs

  <*q,p> <*p,i> <*r,i>

- points-to relations

  (q,p) (p,i) (r,i)

Why is this important?

Aliasing: an assumed name?

Example function:

```
void f (int a[], int b[]) {
    int i;
    for (i = 0; i < 9; ++i) {
        a[i + 1] = b[i] + b[i + 1];
    }
}
```

Consider two possibilities:

- a != b (*a and b can not be aliases*)
- a = b (*a and b may be aliases*)

What if it’s possible that a = b?

Aliasing: an assumed name?

a may equal *b*:

2 loads/loop

```
i = 0: LOAD r1, b[0]
       LOAD r2, b[1]
       ADD r3, r1, r2       # Add b[0] and b[1].
       STORE a[1], r3       # Store result in a[1].
i = 1: LOAD r1, b[1]
       LOAD r2, b[2]
       ADD r3, r1, r2       # Add b[1] and b[2].
       STORE a[2], r3       # Store result in a[2].
i = 2: LOAD r1, b[2]
       LOAD r2, b[3]
       ADD r3, r1, r2       # Add b[2] and b[3].
       STORE a[3], r3       # Store result in a[3].
       ...
```

However, what if the compiler could guarantee *a* and *b* never point to the same array?

Aliasing: an assumed name?

a can never equal *b*:

1 load/loop !

```
i = 0: LOAD r1, b[0]
       LOAD r2, b[1]
       ADD r3, r1, r2       # Add b[0] and b[1].
       STORE a[1], r3       # Store result in a[1].
i = 1: LOAD r1, b[2]
       ADD r3, r1, r2       # Add b[1] and b[2].
       STORE a[2], r3       # Store result in a[2].
i = 2: LOAD r2, b[3]
       ADD r3, r1, r2       # Add b[2] and b[3].
       STORE a[3], r3       # Store result in a[3].
       ...
```

Redundant Load Elimination (RLE)

How could the compiler guarantee this?

Solution: Alias Analysis

What is it?

- static, code based analysis performed by compiler
- not an optimization itself, used by many other analyses

Pros

- more accurate memory dependence analyses
- better compiler optimizations
- better scheduling due to fewer pipeline stalls

Cons

- increased memory usage
- analysis time

Alias Analysis Classifications

- *type-based* alias analysis*
- *flow-based* alias analysis*

Compiler responsibility *(mostly)*

Alias Analysis Classifications

Type-based alias analysis

- typically uses *type* compatibility to determine aliases
- most useful with type-safe programming language (Modula-3, Java) (does not support arbitrary pointer type casting)

Flow-based alias analysis

- aliases are based on *point of creation*
- form of *data flow analysis*

Is there a “best” way?

Data Flow Analysis

Concept

- derive information about the *dynamic* behavior of a program by analyzing the *static* code

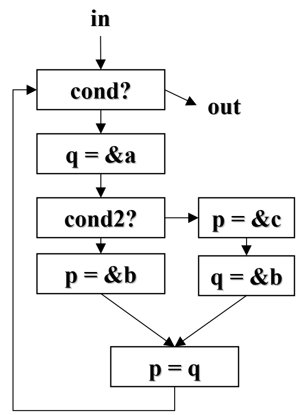
More

- easy for languages with only static data structures (Fortran)
- difficult for languages that allow dynamically allocated data structs (C/C++, Fortran 90, Java, Lisp)
- *forward* or *backward* flow
- typically utilizes *control flow graph* (CFG)

Control Flow Graphs (CFG)

- each statement in the program is a node in the flow graph
- if statement n can be followed by statement m, then there is an edge in the graph from n to m.
- identifies all possible paths through the program

```
while (cond){
    q = &a;
    if (cond2)
        p = &b;
    else{
        p = &c;
        q = &b;
    }
    p = q;
}
```



Data Flow Analysis

Backward analysis: *in* defined in terms of *out*

- depends on what happens 'later'

$$out[n] = \bigcup_{S \in succ[n]} in[S]$$

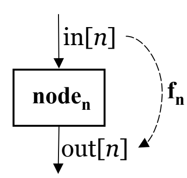
$$in[n] = gen[n] \cup (out[n] - kill[n])$$

Forward analysis: *out* defined in terms of *in*

- depends on what happens 'earlier'

$$in[n] = \bigcup_{P \in pred[n]} out[P]$$

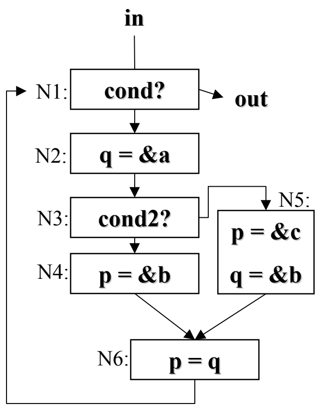
$$out[n] = gen[n] \cup (in[n] - kill[n])$$



f_n *transfer function* relates in[n] and out[n] for same n

Lets look at an example of a forward analysis...

Alias Analysis Example



		In(N)	Gen(N)	Kill(N)	Out(N)
Iteration 1	**N1:**	-	-	-	-
	N2:	-	(q,a)	-	(q,a)
	N3:	(q,a)	-	-	(q,a)
	N4:	(q,a)	(p,b)	-	(q,a)(p,b)
	N5:	(q,a)	(p,c)(q,b)	(q,a)	(q,b)(p,c)
	N6:	(q,a)(q,b) (p,b)(p,c)	(p,a)(p,b)	(p,b)(p,c)	(q,a)(q,b) (p,a)(p,b)
Iteration 2	**N1:**	(q,a)(q,b) (p,a)(p,b)	-	-	(q,a)(q,b) (p,a)(p,b)
	N2:	(q,a)(q,b) (p,a)(p,b)	(q,a)	(q,b)	(q,a) (p,a)(p,b)
	N3:	(q,a) (p,a)(p,b)	-	-	(q,a) (p,a)(p,b)
	N4:	(q,a) (p,a)(p,b)	(p,b)	(p,a)	(q,a)(p,b)
	N5:	(q,a) (p,a)(p,b)	(p,c)(q,b)	(q,a) (p,a)(p,b)	(q,b)(p,c)
	N6:	(q,a)(q,b) (p,b)(p,c)	(p,a)(p,b)	(p,b)(p,c)	(q,a)(q,b) (p,a)(p,b)

Alias Analysis: Some Key Issues

- definiteness (*may-alias* vs. *must-alias*)
- inter-procedural analysis
- flow sensitive vs. flow insensitive
- context sensitive vs. context insensitive
- target analysis size
- precision vs. cost

Definiteness

May alias Problem

- indicates possible alias

```
if (cond)
    p = &a;
...
```

*p and a *might* alias each other

Must alias Problem

- indicates definite alias

```
p = &a;
...
```

*p and a *must* alias each other

Inter-procedural Alias Analysis

Intra-procedural alias analysis

- examples shown have been intra-procedural
- does not taken function calls into effect

```
p1 = &a;      ←—— (p1, a)
p2 = &p1;     ←—— (*p2, p1)
...
foo(p2);      ←—— ??
```

Inter-procedural alias analysis

- gather alias information across many procedures
- many different approaches have been developed
- more soon...

Flow Sensitivity

Sensitive

- alias information is computed at each program point in order
- precise

```
p = &a;  <---- (p, a)
p = &b;  <---- (p, b)
```

$\boxed{p \text{ aliases } b \text{ only } (a \text{ killed})}$

Insensitive

- assumes statements executed in arbitrary order
- computes one solution for all points in a procedure
- efficient

```
p = &a;  <---- (p, a)
p = &b;  <---- (p, a) (p,b)
```

$\boxed{p \text{ aliases both } a \text{ and b}}$

Context Sensitivity

Sensitive

- distinguishes among different procedure invocation contexts
- can be (extremely) computationally expensive

Insensitive

- ignore different procedure invocation contexts

```
foo(a);   /* c1 */
a = &b;
foo(a);   /* c2 */
```

flow and context sensitivity are orthogonal properties

Target Analysis Size

Whole program dataflow analysis

- extensive use to-date in the literature
- easy to maintain context sensitivity

Modular or fragment dataflow analysis

- can analyze incomplete programs (e.g.. library modules)
- large programs more suited for modular analysis
- lower memory requirements
- typically less analysis time required

Precision vs. Cost

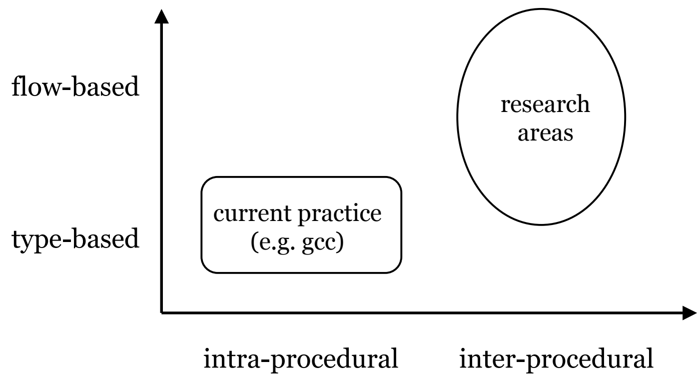
Precision

- refers to number of alias relations reported
- higher precision means less relations reported

Factors effecting precision (and therefore cost)

- use of flow sensitivity
- use of context sensitivity
- how arrays, structs, and the heap are modeled
- the alias representation

Alias Analysis Overview



To enable gcc alias analysis use argument *-fstrict-aliasing*

Topics Overview

- Aliasing
- Alias Analysis Problem
  - general dataflow
  - classifications and features
- → Overview of Inter-procedural Analysis Approaches
  - analysis frameworks, algorithm basics, pros/cons
- Modular Inter-procedural Pointer Analysis Using Access Paths
  - a detailed look
- Inter-procedural Wrap-up / Future of Alias Analysis

Inter-procedural Analysis Approaches

Frameworks

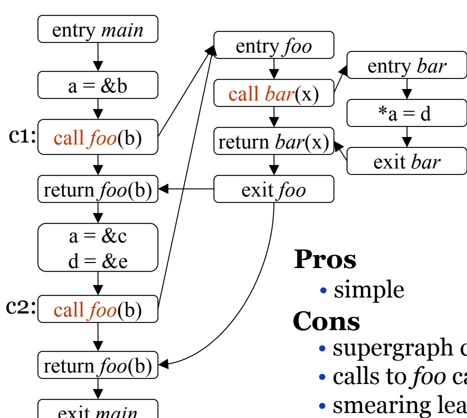
- Supergraph
- Invocation Graph
- Partial Transfer Function (PTF)
- Procedure Call Graph (w/ CISF) (*Ryder et al '99*)
- Procedure Call Graph (w/ CISF) (*Cheng et al'00*)

```
int **a, *b, *c, *d, e;

main( )
{
    a = &b;
    foo(b);   /* c1 */
    a = &c;
    d = &e;
    foo(b);  /* c2 */
}

foo(int * x)
{
    bar(x);
}

bar(int * x)
{
    *a = d;
}
```


adapted from "Survey of Alias Analysis" J. Q. Wu

Control-flow Supergraph

(Landi et al '92,'93)



Algorithm basics

- connect intra-procedural CFGs
- add call and return nodes
- add entry and exit nodes
- context insensitive
- flow sensitive

Pros

- simple

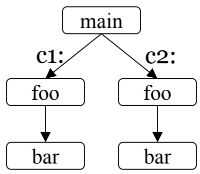
Cons

- supergraph can get *very* large (poor scalability)
- calls to *foo* cannot be distinguished - why?
- smearing leads to spurious (b,e) pair at c2
- whole program analysis
- subset of C features only
- benchmarks tested: < 5000 lines

adapted from "Survey of Alias Analysis" J. Q. Wu

Invocation Graph

(Emami et al '93)



Algorithm basics

- each node is a procedure call site
- reanalyze procedure for every distinct caller
- context sensitive
- flow sensitive

Pros

- very little smearing

Cons

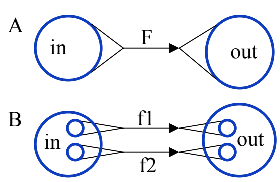
- *extremely* computationally expensive
- number of nodes can increase exponentially
- large memory requirement!
- whole program analysis
- subset of C features only
- benchmarks tested: < 3000 lines

This can be improved with memoization...

adapted from "Survey of Alias Analysis" J. Q. Wu

Partial Transfer Function

(Wilson et al'95)



A) complete transfer function maps entire input domain for a procedure to corresponding outputs

B) partial transfer functions map subsets of domain to outputs

This means we only need to create PTFs for inputs that potentially occur

Algorithm basics

- invocation graph framework
- uses memoization based on *input pattern*
- context sensitive
- flow sensitive

Pros

- handles all C features!
- SPEC92 benchmark: < 24,000 lines

Cons

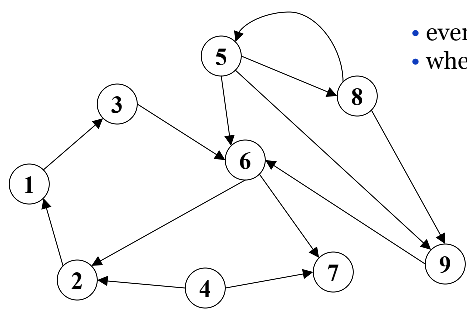
- still exponential worst case
- whole program analysis
- multiple PTFs per procedure

quick definition of SCCs...

Strongly Connected Component (SCC)

- Let G = (V, E, *n*) be a directed graph with entry point *n*
- SCC = maximum subset { $U \subseteq V \mid \forall (v_1,v_2 \in U)\ \exists$ path between v_1,v_2 }

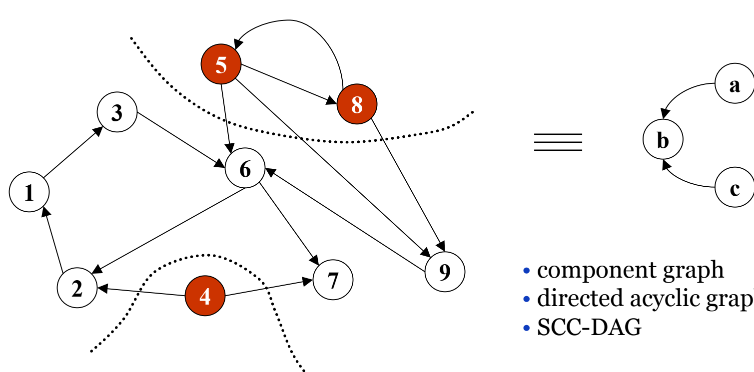
(i.e., subgraph where any node is reachable from any other in that SCC)



- every node is a trivial SCC
- where are the maximum subgraphs?

Strongly Connected Component (SCC)

- SCC = maximum subset { $U \subseteq V \mid \forall (v_1,v_2 \in U)\ \exists$ path between v_1,v_2 }



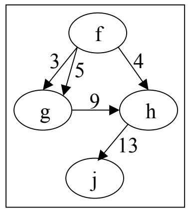
- component graph
- directed acyclic graph (DAG)
- SCC-DAG

Back to inter-procedural AA aproaches...

Procedure Call Graph w/ CISF

(Ryder et al'99)

```
procedure f()
begin
    call g()
    call h()
    call g()
end
procedure g()
begin
    call h()
end
procedure h()
begin
    call j()
end
procedure j()
begin
end
```



Algorithm basics

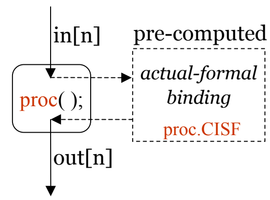
- each *procedure* is a graph node
- each *call* is an edge
- context independent summary function
- relevant context representation
  <RC,(points-to pairs)>
- context sensitive
- flow sensitive

Pros

- modular (SCC-DAG)
- one CISF per procedure only

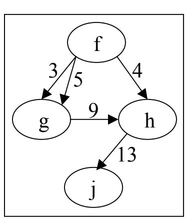
Cons

- subset of C features only
- benchmarks tested: < 6000 lines



Procedure Call Graph w/ CISF

(Cheng et al'00)



Algorithm basics

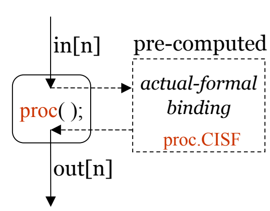
- approach similar to last (*Ryder et al '99*)
- context independent summary function
- access path representation (AP,AP)
- context sensitive
- flow insensitive

Pros

- very efficient computations
- modular (SCC-DAG)
- handles all C features!
- SPEC92,95 benchmarks: < 200,000 lines!

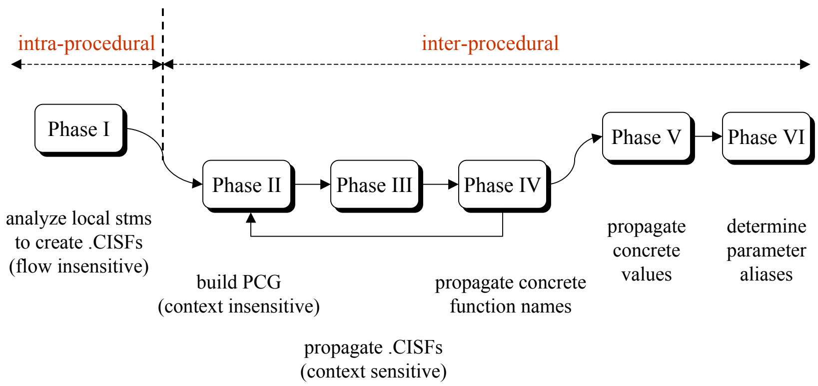
Cons

- flow insensitive ∴ less precision



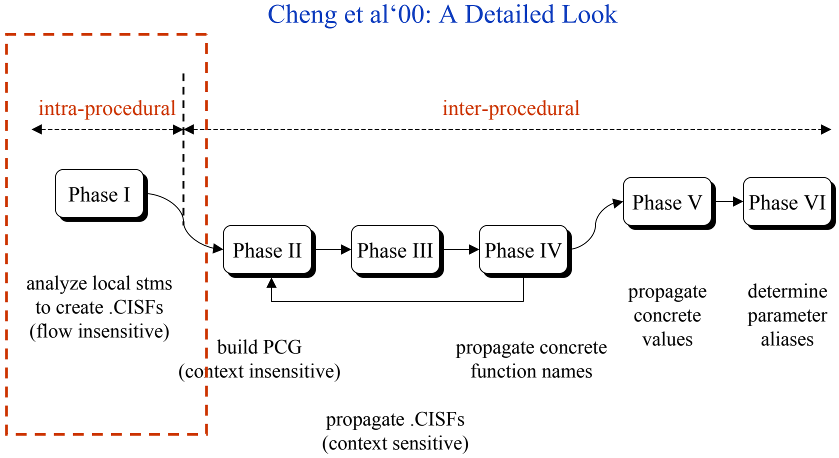
Modular Inter-procedural Pointer Analysis Using Access Paths

Cheng et al‘00: A Detailed Look



Modular Inter-procedural Pointer Analysis Using Access Paths

Cheng et al‘00: A Detailed Look



Intra-procedural Stage

Phase I - analyze local stms

- each function is analyzed as isolated compilation module
- {formal parameters, return values, globals} assumed unknown values
- summary behavior of each function is calculated (func.CISF) incl.:

  a. set of memory locs accessed across func boundaries
  b. set of call site names
  c. set of pointer definitions involving *a*.

- APs represent physical memory locs by how they are accessed from an initial variable (*store-less model*)

Intra-procedural Stage

Phase I - cont…

- algorithm for construction of an access paths from memory exps

1.	AP(v)	= v	*v is a variable*
2.	AP(exp())	= AP(exp)_n()	*exp is {direct,indirect} call-site, n is unique id*
3.	AP(*exp)	= AP(exp)*	*exp is not a function name*
4.		= AP(exp)	*exp is a function name*
5.	AP(exp[i])	= AP(exp)	*exp is of array type and not a formal parameter*
6.		= AP(exp)*	*exp is arbitrary ptr or formal param of array*
7.	AP(exp **op** exp1)	= AP(exp)	*op is binary operator and exp is ptr type var*
8.	AP(exp->field)	= AP(exp)*.so_eo	
9.	AP(exp.field)	= AP(exp).so_eo	
10.	AP(exp.field1.field2)	= AP(exp).so3_eo3	*so3 = so1+so2 and eo3 = so1+eo2*

Example: (assume key: so = 0, eo = 3)

sp->key = &g1;

rule 8: AP(sp->key) = AP(sp)*.0_3
rule 1: AP(sp) = sp
∴ AP(sp->key) = sp*.0_3

But what about recursive data structures?...

Intra-procedural Stage

Phase I - cont…

- recursive data structures
- use *recursive-sensitivity* parameter '*k*'

```
typedef struct S
{
    int * key;          /* 0_3 */
    struct S * next;    /* 4_7 */
}

S * sp;
int g1;

sp->next->next->key = &g1;
```

AP Points-to relations (AP(lhs), AP(rhs))	**'*k*'**
(sp*.4_7*.4_7*.0_3, g1)	$k >= 3$
(sp*.4_7*.0_3, g1)	$k = 2$
(sp*.0_3, g1)	$k = 1$

EVAL(AP, S_{ptr}) Closure Function

Purpose

- S_{ptr} = set of points-to relations
- returns normalized *right-most* APs from corresponding input AP

- result of EVAL(AP, S_{ptr}) = {set of APs}   *why? either:*
  - ptr is bound based on a conditional event
  - flow insensitive nature of the algorithm

Example

```
   …
1.   temp = fa;
   …
2.  *temp = fb;
   …
```

AP(*temp) = temp*
AP(fb) = fb

currently: S_{ptr}= (temp, fa*)

S_{ptr}= (EVAL(temp*,S_{ptr}) , EVAL(fb, S_{ptr})) ∪ S_{ptr}
= ({fa*} , {fb*}) ∪ S_{ptr}
= (fa*, fb*) ∪ S_{ptr}

Intra-procedural Stage Pseudocode

Phase I - analyze local stms cont…

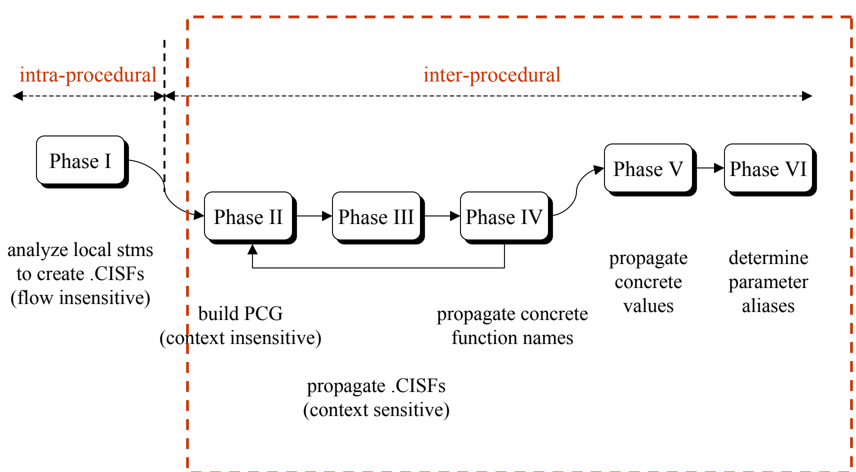
- intra-procedural pseudocode
- S_{ptr}(*proc*)= set of points-to relations assoc. with procedure *proc*

Intra-procedural_pointer_analysis(proc)
{
  S_{ptr} (proc) = ∅;
  do {
    apply EVAL(AP(*exp), S_{ptr}(proc)) for ∀ptr type *exps* in proc;
    for (each "*lhs* = *rhs*" ∈ proc | lhs,rhs are ptr *exp* and rhs is not null)
      construct points-to relations s.t. *lhs* = *rhs*->*;
    for (each structure/union assignment "lhs = rhs" ∈ proc)
      construct points-to relations s.t. *lhs* = *rhs*->* for every ptr type field f;
  } while (new APs or points-to relations are added);
}

results in creating proc.CISF

Modular Inter-procedural Pointer Analysis Using Access Paths

Cheng et al'00: A Detailed Look



Inter-procedural Stages

Phase II - build PCG
- depth first search from main() to construct PCG

Phase III - propagate .CISFs
- CISFs $\in$ each SCC are propagated iteratively until fixed point reached
- CISFs $\in$ prog propagate along SCC-DAG in *reverse topological order*

Phase IV
- propagate concrete function names to continue building PCG

Phases V
- propagate all concrete values *top-down* along SCC-DAG

Phases VI
- determine parameter aliases

propagation example...

Phase II-IV: CISF propagation

```
void fn1(  (**fa)(int*) , (*fb)(int*) )
{
     (**temp)(int*);
      temp = fa;
    *temp = fb;
}

void fn3(int * s)
{ ... }

main
{
     void (*fn2)(int*);
     fn1(&fn2, fn3);
}
```

recall: these 2 stms add (fa*, fb*) to S_{ptr}(fn1) in phase 1

How to propagate (fa*, fb*) of fn1 to main?

actual-formal binding

1) (fa*, fb*) $\in$ S_{ptr} (fn1)
2) *fa* replaced with &fn2*
3) EVAL((&fn2)*, S_{ptr} (main)) = {fn2}
   (fn2, fb*)
4) *fb* replaced with fn3*
5) EVAL(fn3*, S_{ptr} (main)) = {fn3}

fn1.CISF

<- - (fn2, fn3) propagates to main

Inter-procedural Stage Pseudocode

Complete Phases II - VI

- points-to relations (PTR)
- extended access paths (EAP) find dependencies of dynamically allocated objects (not covered today)

what does the whole inter-procedural algorithm look like?

next page...


```
Inter-procedural_pointer_analysis(prog)
{
   do{
      resolve func ptrs for each indirect call site;
      use depth-first search to construct SCC-DAG ∀ reachable p∈prog;
      for (each SCC ∈ prog - in a bottom-up order)
      {
         determine EAP for each AP of each p ∈ SCC;
         iteratively propagate PTR within SCC;
         re-determine EAP for each p ∈ SCC iff new PTR created;
         propagate CISF of SCC to caller SCC(s);
      }
      for (each SCC ∈ prog - in a top-down order)
      {
         iteratively propagate function names within SCC;
         propagate function names from SCC to callee SCC(s);
      }
   }while( call graph changed in previous iteration )

   for (each SCC ∈ prog - in a top-down order)
   {
      iteratively propagate concrete values within SCC;
      propagate concrete values from SCC to callee SCC(s);
   }
   determine aliases among parameters;
}
```

Inter-procedural Wrap-up

Reasons current compilers don't use inter-procedural AA

- benefits of the optimizations not well explored
- analysis is very expensive
- analysis is extremely complex
- scalability to large programs
- need (most) all source code for efficient analysis (library?)

Benefits

- precise pointer analysis (*incl. software analysis tools*)
- code parallelization
- compiler client optimizations (i.e., constant propagation)
- OO class analysis

Trends

- increase in number of procedures used (OO, smaller + more)
- program sizes increasing
- analysis costs decreasing (better hardware)

∴ inter-procedural analysis is becoming more important!

Future Of Alias Analysis

Until ~1990

- inter-procedural alias analysis primarily used to find aliases between formal parameters and globals
- its use on (complete) languages with general pointers was inconceivable

Recently

- much successful research done analyzing realistic programs with general pointers
- this research has yet to go main stream (i.e., gcc, etc.)

Summary

- Aliasing
- Alias Analysis Problem
  - general dataflow (forward, backward, {in, out, gen, kill})
  - classification (*type-based, flow-based*)
  - features (*definiteness, flow/context sensitivity, size, precision/cost*)
- Overview of Inter-procedural Analysis Approaches
  - supergraph, invocation graph, PTF, procedure call graph (CISF)
- Modular Inter-procedural Pointer Analysis Using Access Paths
  - a detailed look
- Inter-procedural Alias Analysis Importance/Future

References

1) "Efficient Context-Sensitive Pointer Analysis for C Programs", Wilson et al, Proceeding of the ACM SIGPLAN'95 Conference on PLDI, pp.1-12, June 1995
2) "Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation" Cheng et al, PLDI 2000, Vancouver, British Columbia, Canada
3) "Optimization that makes C++ faster than C" Mark Mitchell, Dr. Dobb's Journal, October 2000
4) "Survey of Alias Analysis" Johnson Qiang Wu, www.princeton.edu/~jqwu/Memory/survey.html
5) "Dataflow analysis of software fragments" Atanas Rountev, PhD dissertation, Rutgers University, 2002
6) "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers", Emami et al, Proceedings of ACM SIGPLAN'94 Conference on PLDI, pp. 242-256, June 1994.
7) "A safe approximate algorithm for inter-procedural pointer aliasing", Landi et al, Proceeding of the ACM SIGPLAN'92 Conference on PLDI, pp.235-248, June 1992.
8) "Interprocedural Pointer Alias Analysis", Hind et al, ACM Transactions on Programming Languages, Vol. 21, No. 4, July 1999
9) "Type-Based Alias Analysis", Diwan et al, ACM SIGPLAN Conference on PLDI, pp. 106-117, June 1998