

Context-Sensitive Pointer Analysis

Last time

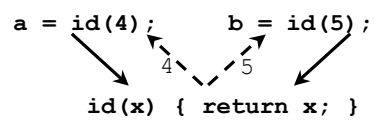
- Flow-insensitive pointer analysis

Today

- Context-sensitive pointer analysis
 - Partial Transfer Functions
 - BDD-based analysis
- The big picture

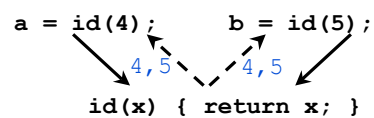
Recall Context Sensitivity

Is x constant?



Context-sensitive analysis

- Computes an answer for every callsite:
 - x is 4 in the first call
 - x is 5 in the second call



Emami 1994

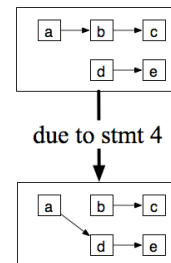
Overview

- Uses invocation graph for context-sensitivity
- Can be exponential in program size
- Handles function pointers

Characterization of Emami

- Whole program
- Flow-sensitive
- Context-sensitive
- May and must analysis
- Alias representation: points-to
- Heap modeling: one heap variable
- Aggregate modeling (fields and arrays)

```
int **a, *b, c, *d, e;  
1: a = &b;  
2: b = &c;  
3: d = &e;  
4: a = &d;
```



Partial Transfer Functions [Wilson et. al. 95]

Key idea

- Exploit commonality among contexts
- Provide one procedure summary (PTF) for all contexts that share the same input/output aliasing relationships
- Think of it as application of memoization to Emami

Partial Transfer Functions – Example

```
main() {
  int *a,*b,c,d;
  a = &c;
  b = &d;
  swap(&a, &b); // S0
  for (i = 0; i<2; i++) {
    bar(&a,&a); // S1
    bar(&b,&b); // S2
    bar(&a,&b); // S3
    bar(&b,&a); // S4
  }
}

void bar(int **i, int **j) { swap(i,j); }
void swap(int **x, int **y) {
  int *temp = *x;
  *x = *y;
  *y = temp;
}
```

How many contexts do we care about?

- Two: the formals either alias or they do not alias

In practice

- Only need 1 or 2 PTF's per procedure
- Complex to implement

6

Binary Decision Diagrams (BDDs)

A data structure

- Extensively used in the model-checking community

Benefits

- Compactly represents sets and relations
- Operations are proportional to the size of the BDD, not the size of the set or relation

How does this apply to pointer analysis?

Andersen-Style Pointer Analysis – Recap

Program

$a := \&b$
 $c := a$
 $a := \&d$
 $e := a$

Constraints

$a \supseteq \{ b, d \}$
 $c \supseteq a$
 $e \supseteq a$

Points-to Relations

$a \rightarrow \{ b, d \}$
 $c \rightarrow \{ b, d \}$
 $e \rightarrow \{ b, d \}$

We've reached a fixed point

Base constraints

- Used to initialize the points-to sets
- Ex: $a := \&b$
- Not needed after initialization

Complex constraints

- Involve pointer dereferences
- Ex: $*a := c$

Simple constraints

- Involve variable names only
- Ex: $c := a$

Procedure calls

- Insert constraints for copying parameters and return values

Andersen-Style Pointer Analysis

Represent two sets

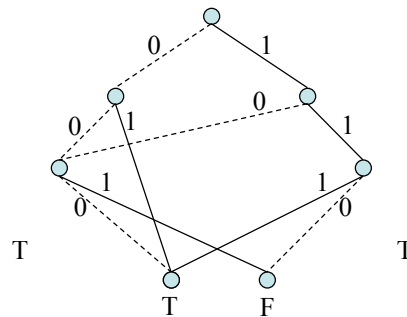
- $C = \{ (a,b) \mid a \supseteq b \}$ // Constraints
- $P = \{ (a,b) \mid a \rightarrow b \}$ // Points-to sets

Iterate until we reach a fixed point:

- $S = \{ (a,c) \mid \exists b. ((a,b) \in C \ \& \ (b,c) \in P) \}$ // Propagate constraints
- $P := P \cup S$

Binary Decision Diagrams (BDDs)

000, 010, 011, 100, 111



Symbolic Pointer Analysis

Encode relations as BDDs

- $C = \{ (a,b) \mid a \supseteq b \}$
- $P = \{ (a,b) \mid a \rightarrow b \}$

Possible strategies

- Encode both C and P as BDDs
- Encode P as a BDD, but not C
- Encode C as a BDD, but not P

Recent work

- Success for Java [Whaley and Lam '04]
 - Can analyze 600K lines of code
- Less successful for C—an order of magnitude smaller programs
- Has not yet been applied to flow-sensitive analyses

The Big Picture

Where do we lose precision?

- Let's revisit our running example from last week

Revisiting Our Earlier Example

Flow-insensitive context-sensitive (FICS)

```
int** foo(int **p, **q)
{
    int **x;           p1 → {b}
                      p2 → {d}
    x = p;             q1 → {f}
    . . .             q2 → {g}
    x = q;             x1 → {b, f}
    return x;          x2 → {d, g}
}

int main()
{
    int **a, *b, *d, *f, a → {b, d, f, g}
        c, e;           b → {c, e}
                      d → {c, e}
    a = foo(&b, &f);    f → {c, e}
    *a = &c;            g → {c, e}
    a = foo(&d, &g);
    *a = &e;
}
```

Revisiting Our Earlier Example (cont)

Flow-sensitive context-sensitive (FSCS)

```

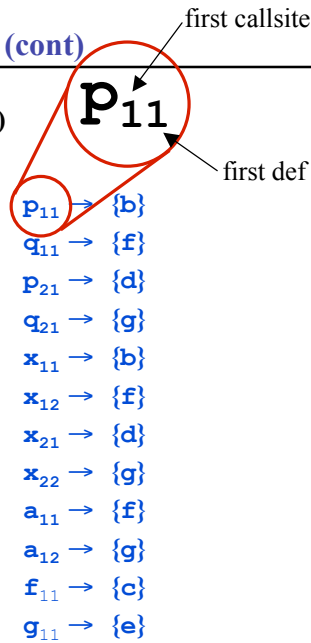
int** foo(int **p, **q)
{
    int **x;

    x = p;
    . . .
    x = q;
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}

```



Revisiting Our Earlier Example (cont)

Flow-insensitive context-insensitive (FICI)

```

int** foo(int **p, **q)
{
    int **x;

    x = p;
    . . .
    x = q;
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;

    a = foo(&b, &f);
    *a = &c;
    a = foo(&d, &g);
    *a = &e;
}

```

p → {b, d}
 q → {f, g}
 x → {b, d, f, g}

a → {b, d, f, g}
 b → {c, e}
 d → {c, e}
 f → {c, e}
 g → {c, e}

Revisiting Our Earlier Example (cont)

Flow-sensitive context-insensitive (FSCI)

```
int** foo(int **p, **q)
{
    int **x;           p → {b, d}
    x = p;             q → {f, g}
    . . .              x1 → {b, d}
    x = q;             x2 → {f, g}
    return x;
}

int main()
{
    int **a, *b, *d, *f,
        c, e;          a1 → {f, g}
                                a2 → {f, g}
                                f1 → {c}
                                g1 → {c}
    a = foo(&b, &f);     f2 → {c, e} (weak update)
    *a = &c;             g2 → {c, e} (weak update)
    a = foo(&d, &g);
    *a = &e;
}
```

CIS 570 Lecture 13

Context-Sensitive Pointer Analysis

16

Imprecision

Weak updates

- Occur more often in flow-insensitive and context-insensitive analyses

The callgraph

- When function pointers are used, pointer analysis is needed to build the callgraph
- Imprecision in pointer analysis leads to imprecision in the callgraph
 - A conservative callgraph has more edges than a less conservative callgraph
- Imprecision in the callgraph leads to further imprecision in the pointer analysis

CIS 570 Lecture 13

Context-Sensitive Pointer Analysis

17

Approximations

Many ways to approximate

- Recall that the constraint graph has nodes representing variables and edges representing constraints
- The many dimensions of pointer analysis represent different ways of collapsing the constraint graph

Flow-insensitive

- Andersen:
 - Collapse all constraints (assignments) pertaining to a given variable into a single node
- Steensgaard:
 - Collapse all nodes that have been assigned to one another into a single node
 - Allows information to flow from rhs to lhs as well as from lhs to rhs

Andersen 94

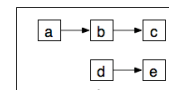
Overview

- Uses subset constraints
- Cubic complexity in program size, $O(n^3)$

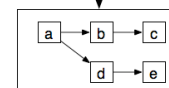
Characterization of Andersen

- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling?
- Aggregate modeling: fields

```
int **a, *b, c, *d, e;  
1: a = &b;  
2: b = &c;  
3: d = &e;  
4: a = &d;
```



due to stmt 4



source: Barbara Ryder's Reference Analysis slides

Steensgaard 96

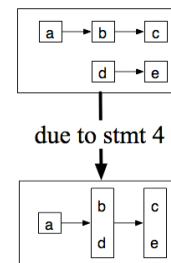
Overview

- Uses unification constraints
- Almost linear in terms of program size
- Uses fast union-find algorithm
- Imprecision from merging points-to sets

```
int **a, *b, c, *d, e;  
1: a = &b;  
2: b = &c;  
3: d = &e;  
4: a = &d;
```

Characterization of Steensgaard

- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling: none
- Aggregate modeling: possibly



source: Barbara Ryder's Reference Analysis slides
CIS 570 Lecture 13 Context-Sensitive Pointer Analysis

20

More Approximations

Context-insensitive analysis

- Collapse all constraints arising from different callsites of a procedure into a single node

Partial Transfer Functions

- Collapse constraints for all callsites of a procedure that share the same aliasing relationships

Field-insensitive

- Collapse all fields of a structure into a single node

Field-based

- Collapse all instances of a struct type into one node per field
- Example: one node for all instances of `student.name`, and another node for all instances of `student.gpa`

CIS 570 Lecture 13 Context-Sensitive Pointer Analysis

21

Yet More Approximations

Address Taken

- Collapse all objects that have their address taken into a single node
- Assume that all pointers point to this node

Heap naming

- One heap:
 - Collapse all heap objects into a single node
- Static allocation site
 - Collapse all instances of objects that are allocated at the same program location into a single node

Concepts

Partial Transfer Functions

- Exploit commonality among contexts

BDD's

- Compact data structure
- Efficient operations on sets

Sources of imprecision

Next Time

Next lecture

- Program slicing