Context-Sensitive Pointer Analysis

Last time

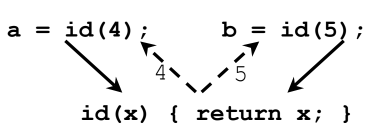
– Flow-insensitive pointer analysis

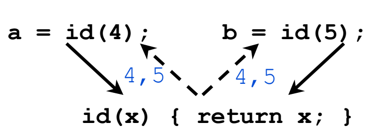
Today

– 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

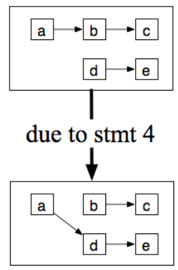
Overview

– 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

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	Constraints	Points-to Relations
`a := &b`	$a \supseteq \{ b, d \}$	a → { b, d }
`c := a`	$c \supseteq a$	c → { b, d }
`a := &d`	$e \supseteq a$	e → { b, d }
`e := a`		

We've reached a fixed point

Base constraints
- Used to initialize the points-to sets
- Ex: `a := &b`
- Not needed after initialization

Simple constraints
- Involve variable names only

  Ex: `c := a`

Complex constraints
- Involve pointer dereferences

  Ex: `*a := c`

Procedure calls
- Insert constraints for copying parameters and return values

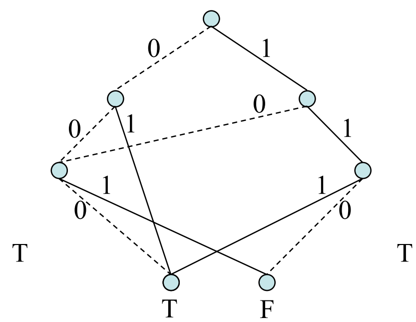
Andersen-Style Pointer Analysis

Represent two sets
- **C** = { (a,b) | $a \supseteq b$ } // Constraints
- **P** = { (a,b) | a → b } // Points-to sets

Iterate until we reach a fixed point:
- **S** = { (a,c) | ∃b.((a,b) ∈ **C** & (b,c) ∈ **P)** } // Propagate constraints
- **P := P U S**

Binary Decision Diagrams (BDDs)

`000, 010, 011, 100, 111`



Symbolic Pointer Analysis

Encode relations as BDDs

- **C** = { (a,b) | a ⊇ b }
- **P** = { (a,b) | a → 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;

   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;
}
```

$p_1 \rightarrow \{b\}$

$p_2 \rightarrow \{d\}$

$q_1 \rightarrow \{f\}$

$q_2 \rightarrow \{g\}$

$x_1 \rightarrow \{b, f\}$

$x_2 \rightarrow \{d, g\}$

$a \rightarrow \{b, d, f, g\}$

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

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

$f \rightarrow \{c, e\}$

$g \rightarrow \{c, e\}$

Revisiting Our Earlier Example (cont)

first callsite

p_{11}

first def

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;
}
```

$p_{11} \rightarrow \{b\}$
$q_{11} \rightarrow \{f\}$
$p_{21} \rightarrow \{d\}$
$q_{21} \rightarrow \{g\}$
$x_{11} \rightarrow \{b\}$
$x_{12} \rightarrow \{f\}$
$x_{21} \rightarrow \{d\}$
$x_{22} \rightarrow \{g\}$
$a_{11} \rightarrow \{f\}$
$a_{12} \rightarrow \{g\}$
$f_{11} \rightarrow \{c\}$
$g_{11} \rightarrow \{e\}$

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;
}
```

$p \rightarrow \{b, d\}$
$q \rightarrow \{f, g\}$
$x \rightarrow \{b, d, f, g\}$

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

Revisiting Our Earlier Example (cont)

Flow-sensitive context-insensitive (FSCI)

```
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;
}
```

$p \rightarrow \{b, d\}$
$q \rightarrow \{f, g\}$
$x_1 \rightarrow \{b, d\}$
$x_2 \rightarrow \{f, g\}$

$a_1 \rightarrow \{f, g\}$
$a_2 \rightarrow \{f, g\}$
$f_1 \rightarrow \{c\}$
$g_1 \rightarrow \{c\}$
$f_2 \rightarrow \{c, e\}$ **(weak update)**
$g_2 \rightarrow \{c, e\}$ **(weak update)**

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

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

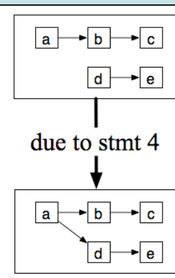
– Andersen:
  – Collapse all constraints (assignments) pertaining to a given variable into a single node
– Steensgaard:
  – Collapse all nodes that have been assigned to one another into a single node
  – Allows information to flow from rhs to lhs as well as from lhs to rhs

Andersen 94

Overview

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

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



Characterization of Andersen

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

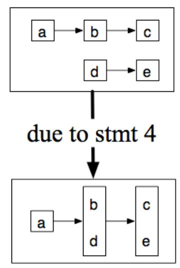
source: Barbara Ryder's Reference Analysis slides

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

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`

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