Flow-Context-Sensitive Pointer Analysis

Yulei Sui
Supervisor: Prof. Jingling
CORG
UNSW

December 24, 2010
Outline

**Pointer Analysis**
Why Pointer Analysis?
Why Pointer Analysis is difficult?

**Flow- Context- Sensitive Pointer Analysis**
Previous Related Works
Two Approaches for Flow Sensitive Analysis
Two Approaches for Context Sensitive Analysis

**Level by Level: Flow- and Context- Sensitive Analysis (Levpa)**

**Recursion Problem in Flow- Context- Sensitive Analysis**
Cloning-based FSCS Recursion Solution
Summary-based FSCS Recursion Solution

**Future Work on Levpa**
Recursion
Partition of Points-to Graph
Why Pointer Analysis?

- **Good news**
  - Static analysis, a sense of the whole program, analysis before actual execution, cheaper than dynamic analysis mostly
  - Many clients
    - Summarize Information for optimisation
    - Improve security vulnerability
    - Create opportunity parallelisation
    - Detect Multi-thread data race
    - Deal with buffer overflow
    - ...

- **Bad news**
  - Hard to be scalable, almost impossible to be fully precise in large program.
  - Too much conservative information.
  - Expensive for Flow- Context- Analysis
Example: Pointer Analysis for Optimization

(a) Local Common Subexpression Elimination

\[ a = \ast x + y \]
\[ b = \ast z + y \]

(Ok?)

(b) Global Common Subexpression Elimination

\[ a = \ast x + y \]
\[ b = \ast z + y \]

\[ c = \ast x + y \]

(Ok?)

(c) Dead Code Elimination

\[ \text{int} \ast a = x \]
\[ \ldots \]
\[ z = \ast a \]

(Ok?)

(d) Code Motion

\[ \text{while}(i \leq \ast x + 2)\{ \]
\[ \ldots \]
\[ z = x \]
\[ \text{while}(i \leq t)\{ \]
\[ \ldots \]
\[ (Ok?) \]

\[ t = \ast x + 2 \]
Example: Pointer Analysis for Detecting Security Vulnerability

- Web application security Vulnerability
- User input accessing databases
- ....
- Let’s see an example

```java
char* p1 = req.getParameter();
char*p2 = "Update moneytable
    set myaccount = 1000000
    where accountID = yulei"
...
stmt.executeQuery(p1);

// How could that be possible for execution?
// if *p1 aliases *p2
```

p1 points to ”Update moneytable set myaccount = 100 where accountID = yulei”
if *p1 and *p2 memory alias, then following may happen:
p1 points to ”Update moneytable set myaccount = 1000000 where accountID = yulei”
Example: Pointer Analysis for Buffer Overflow

```c
int check_password() {
    char buffer[8];
    printf("Enter password: ");
    gets(buffer);
    char* p = buffer; // if p points to buffer
    p[12] += 2;
    // change function’s return address on stack
    // (gcc 4.3/Linux/i86)
    return strcmp(buffer, "secret") == 0;
}

int main(int argc, char *argv[]) {
    if (check_password()) {
        // Directly be here
        printf("Authenticated\n");
    } else {
        printf("Password Incorrect\n");
    }
}
```
Example: Pointer Analysis for Parallelisation and Data Race Detecting

- Data Dependence

```c
for (i=0 ; i<dimension ; i++)
{
    for (j=0 ; j<dimension ; j++)
        d[i] = c[i] + a[i*dimension+j]*b[j];
}
// if d and c points to the same object,
// data dependence between inner loop
```

- Data Race

```c
// x, y are both function pointers
// one thread:
    x();
// another thread
    y();
// if x and y points to the same function (x=&foo, y=&foo)
// and execution without lock properly
```
Why Pointer Analysis is difficult?

- let us see how many pointers and callsite in c program:

<table>
<thead>
<tr>
<th>linenum</th>
<th>spec</th>
<th>callsite</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4kloc</td>
<td>181.mcf</td>
<td>82</td>
<td>207</td>
</tr>
<tr>
<td>8.6kloc</td>
<td>164.gzip</td>
<td>581</td>
<td>673</td>
</tr>
<tr>
<td>22kloc</td>
<td>icecast-2.3.1</td>
<td>877</td>
<td>1618</td>
</tr>
<tr>
<td>115kloc</td>
<td>sendmail</td>
<td>19,578</td>
<td>31004</td>
</tr>
<tr>
<td>222kloc</td>
<td>176.gcc</td>
<td>22,615</td>
<td>16568</td>
</tr>
<tr>
<td>1905kloc</td>
<td>wine-0.9.24</td>
<td>393,689</td>
<td>336591</td>
</tr>
<tr>
<td>2383kloc</td>
<td>wireshark-1.2.2</td>
<td>245,278</td>
<td>333654</td>
</tr>
</tbody>
</table>

- How could run fast and keep precision
- Arbitrary computation on pointers (in c: casting, arithmetic, indirect call etc.)
- Only Inter-procedure make sense
- Recursion problem makes analysis more complex
- Context- flow- path- sensitive hard to be scalable
- ...
Flow-Context-Sensitive Pointer Analysis

- What is flow sensitive and context sensitive
- Two approaches for flow sensitive analysis
  - Iteration-based flow sensitive analysis
  - SSA-based flow sensitive analysis
- Two approaches for context sensitive analysis
  - Cloning-based context sensitive analysis
  - Summary-based context sensitive analysis
- Flow- and context-sensitive pointer analysis (Levpa)
- Recursion in flow- and context- pointer analysis
  - Cloning-based FSCS recursion solution
  - Summary-based FSCS recursion solution

Each content above will be given a concrete example
Flow Sensitive and Flow Insensitive Pointer Analysis

- Flow Insensitive: can only generate, does not kill any points-to relations, no order in program.
  - Steensgaard: unification-based
  - Anderson: inclusion-based
- Flow Sensitive: both kill and generate points-to relations through Control Flow graph (CFG)
  - Iteration-based
  - SSA-based

```c
// Flow Insensitive
int main(void){
    int x, y, *p;
    p = & x
    /* (p -> x), (p -> y) */
    foo(p);
    p = & y
    /* (p -> x), (p -> y) */
}

// Flow Sensitive
int main(void){
    int x, y, *p;
    p = & x
    /* (p -> x) */
    foo(p);
    p = & y
    /* (p -> y) */
}
```
Context Sensitive and Context Insensitive Pointer Analysis

- Context Insensitive: merge all the calling context, and a unified context for function input, return a unified output to all the call sites (0-limiting context analysis)
- Context Sensitive: distinguish calling from which call site, return the result to the corresponding call site (k-limiting)
  - Cloning-based
  - Summary-based
Flow-Context-Sensitive Pointer Analysis

Previous Related Works

- M. Emami, R. Ghiya, L. J. Hendre. Context-sentive interprocedural points-to analysis in the presence of function pointers. (PLDI 94)
- R.P.Wilson and M.S.Lam. Efficient context-sensitive pointer analysis for C programs. (PLDI 95)
- J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams (PLDI 04)
- Kahlon, V. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis (PLDI 08)
- B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. (POPL 09)
Approaches for Flow and Context Sensitive Analysis

- Two approaches for flow sensitive analysis
  - Iteration-based: iterative data flow analysis until reach a fixed point.
  - SSA-based: generate use-def chain, full sparse.
- Two approaches for context sensitive analysis
  - Cloning-based: clone calling path, choice in precision (K-limiting)
  - Summary-based: two passes, bottom-up (compute TF), top down (apply TF, propagate)
Example: Iteration-based Flow Sensitive Analysis

Emami Masters’ Thesis

process_while (cond,body, in) {
    {
        last_in = in;
        out = points-to(body,in);
        in = merge_info(in,out);
    } while(last_in!=in);
    result = in;
    return result;
}

Merge Rules
MustSet={ (x,y,must) | (x,y,must) ∈ (s1∩ s2) }
MaySet={ (x,y,may) | (x,y,rel) ∈ (s1∪ s2) ∨
      (x,y,must) /∈ MustSet }
Merge(s1,s2) = MustSet ∪ MaySet

Here is an example:
s1 = {(a,a1,must), (b,b1,must),(c,c1,must)}
s2 = {(a,a1,must), (b,b1,may)}
MustSet = { (a,a1,must) }
MaySet = { (b,b1,may), (c,c1,may) }
Merge(s1,s2)={(a,a1,must), (b,b1,may),(c,c1,may)}

// First Approximation
s1: {(b,c,must)}
s2: {(b,c,must),(a,c,must)}
s3: {(b,d,must),(a,c,must)}
s4: {(b,c,may), (b,d,may),(a,c,may)}

// at s4 it merge s3 with s1

// Second Approximation
s1: {(b,c,may), (b,d,may),(a,c,may)}
s2: {(b,c,may), (b,d,may),(a,c,may),(a,d,may)}
s3: {(b,d,must),(a,c,may),(a,d,may)}
s4: {(b,c,may), (b,d,may),(a,c,may),(a,d,may)}

// at s4 it merge s3 with s1

third approximation same as second
**Example: SSA-based Flow Sensitive Analysis**

*Hardokef POPL 09*

- **SSA**: Static Single Assignment, Only one def for each variable
- **Build SSA for variable** \( b \) **Steps**: (1) Build CFG, (2) Build Dominator Tree, (3) Place \( \Phi \) function

```c
int *a, *b, c, d;
b = &c;
while(cond) {
    a = b;
    if(cond)
        b = &d;
    *b = c
}
```

![Dominator Tree](image1)

![CFG](image2)

![Placing \( \Phi \)](image3)
Example: SSA-based Flow Sensitive Analysis (cont)

Hardokef POPL 09

• (4) Rename

1. \( b_1 = \&c \)
   \( b = \Phi(b, b_1) \)
   \( b = \&d \)
   \( b = \Phi(b, b) \)

2. \( b_1 = \&c \)
   \( b_2 = \Phi(b, b_1) \)
   \( b = \&d \)
   \( b = \Phi(b_2, b) \)

3. \( b_1 = \&c \)
   \( b_2 = \Phi(b_1) \)
   \( b_3 = \&d \)
   \( b = \Phi(b_2, b_3) \)

4. \( b_1 = \&c \)
   \( b_2 = \Phi(b_4, b_1) \)
   \( b_3 = \&d \)
   \( b_4 = \Phi(b_2, b_3) \)
Example: SSA-based Flow Sensitive Analysis (cont)

Hardokef POPL 09

- (5) Generate use-def chain $b$
  - Do not need to main all points-to set for $b$ at each assignment point
  - Search for $b$’s points-to set via use-def chain

```c
int *a, *b, c, d;
b_1 = &c;
while(cond) {
    b_2 = Φ(b_4, b_1);
    a = b_2;
    if(cond)
        b_3 = &d;
    b_4 = Φ(b_2, b_3);
    *b_4 = c
}
```
Example: Cloning-based Context Sensitive Analysis

Emami (PLDI 94)
Example: Summary-based Context Sensitive Analysis

Wilson (PLDI 95)

```c
f(int* p, int* q, int* r)
{
    *p = *q;
    *q = *r;
}
main()
{
    int x, y, z;
    int *x0, *y0, *z0;
    x0=&x, y0=&y, z0=&z;
    if(cond1)
    C1: f(&x0, &y0, &z0);
        p → p₁ p₂
        q → q₁ q₂
        r → r₁ r₂
        TF for C1
        TF for C2
        C1 and C2 use same TF
    else if(cond1)
    C2: f(&z0, &x0, &y0);
        x0—> y
        y0—> z
        z0—> z
        C1 and C2 use same TF
    else
    C3: f(&x0, &y0, &x0);
        p → p₁ p₂
        q → q₁ q₂
        r
        TF for C3
        recompute a new PTF
        alias among formal parameters
    }
```
Level by Level: Flow- and Context-Sensitive Analysis (Levpa)

- **Good news**
  - Partition the points-to graph analyze pointers level by level
  - SSA-based flow sensitive analysis on full sparse SSA
  - Summary-based context sensitive analysis with transfer function based on BDD

- **Bad news**
  - Sensitive to recursion
  - Still have some bugs cannot pass majority specs of SPEC2006 and some specs of SPEC2000
# Lepa Analysis Process

**Framework**

- build call graph
- build points-to graph (Steensgaard)
- compute points-to level
- build SCC DAG
- analyze a level
- bottom-up in topological order

**Bottom-up phase**

- start bottom-up phase
- create mu chi list for call sites
- build SSA
- build constrain graph
- constrain resolution
- apply transfer function
- evaluate transfer function
- cycle in SCC
- points-to cycle

**Top-down phase**

- start top-down phase
- expand dereference
- propagate points-to set
- cycle in SCC
- increase build call graph
- reanalyze level, update TF
- analyze next level

**Level by Level: Flow- and Context- Sensitive Analysis**


- Recursion Problem in Flow- Context- Sensitive Analysis

- Future Work on Levpa
**Example: Levpa Analysis Process**

*A Motivation Example*

```c
L1: #include <stdio.h>
L2: int obj, t;
L3: void foo(int** , int**);
L4: main()
L5:     int **x, **y;
L6:     int *a, *b, *c, *d,*e;
L7:     x=&a; y =&b;
L8:     foo(x,y);
L9:     *b = 5;
L10:    if(t){ x=&c; y =&e;}
L11:    else { x= &d; y = &d;}
L12:    c = &t;
L13:    foo(x,y);
L14:    *e = 10;
L15: }
L16:    void foo(int **p, int **q){
L17:        *p = *q;
L18:        *q = &obj;
L19: }
```

Points-to graph with Steensgaard Alg

points-to level \{x, y, p, q\} = 2 points-to level \{a,b,c,d,e\} = 1 points-to level \{t, obj\} = 0
analysis order: 2− >1− >0
Example: Levpa Analysis Process

Bottom-up Phase of level 2 (build SSA)

L1: #include <stdio.h>
L2: int obj, t;
L3: void foo(int**, int**);
L4: main()
L5: 
L6: int **x, **y;
L7: int *a, *b, *c, *d, *e;
L8: x1=&a; y1 =&b;
L9: foo(x1,y1);
L10: *b = 5;
L11: if(t){ x2 = &c; y2 = &e; }
L12: else { x3 = &d; y3 = &d; }
L13: x4 = \Phi(x2,x3); y4 = \Phi(y2,y3);
L14: c = &t;
L15: foo(x4,y4);
L16: *e = 10;
L17: }
L18: void foo(int **p, int **q){
L19: *p = *q;
L20: *q = &obj;
L21: }

(1) function call and pointer dereference have no side effect on top level variable
(2) no need to create callsites mu chi list and evaluate transfer function for top level variable
(3) build SSA directly: first build SSA form for foo and then for main
Example: Levpa Analysis Process

Top-down Phase of level 2 (propagate and expand)

L1: #include <stdio.h>
L2: int obj, t;
L3: void foo(int**, int**);
L4: main()
L5: int **x, **y;
L7: x1 = &a; y1 = &b;
L8: foo(x1, y1); points-to set: {(p=a), (q=b)}
L9: *b = 5;
L10: if(t) { x2 = &c; y2 = &e; }
L11: else { x3 = &d; y3 = &d; }
L12: x4 = \( \Phi(x_2, x_3) \); y4 = \( \Phi(y_2, y_3) \);
L13: c = &t;
L14: foo(x4, y4); points-to set: {(p=c), (p=d)} {(q=d), (q=e)}
L15: *e = 10;
L16: }
L17: void foo(int **p, int **q) {
L18: \( \mu(b,d,e) \) //or \( \mu(b, q -> b), \mu(d, q -> d), \mu(e, q -> e) \)
L19: \*p1 = \*q1;
L20: \( \chi(a,c,d) \) //or \( a=\chi(a, p -> a), c=\chi(c, p -> c), d=\chi(d, p -> d) \)
L21: \*q1 = &obj;
L22: \( \chi(b,d,e) \) //or \( b=\chi(b, q -> b), d=\chi(d, q -> d), e=\chi(e, q -> e) \)
L23: }

(1) propagate points-to set of level 2 to callsites:
L8: {(p,a,must), (q,b,must)}, L14: {(p,c,may), (p,d,may)} {(q,d,may), (q,e,may)}
(2) expand the dereference(*p,*q) for level 2 variable in top-down order (no dereference of level 2 variable in main)
(3) mu is read side effect and chi is write side effect, write side effect depends on read side effect.
(4) points-to relation: (=>: must, ->: may)
Example: Levpa Analysis Process

Bottom-up Phase of level 1 (build SSA and evaluate TF at function foo)

L17: void foo(int **p, int **q)
L18: \( \mu(b_1, q = \rightarrow b) \)
L19: \( \mu(d_1, q = \rightarrow d) \)
L20: \( \mu(e_1, q = \rightarrow e) \)
L21: \*p = \*q; // must indicates must be modified if condition meets
L22: \( a_2 = \chi(a_1, \text{must}, p = \rightarrow a) \)
L23: \( c_2 = \chi(c_1, \text{may}, p = \rightarrow c) \) // may indicates may be modified if condition meets
L24: \( d_2 = \chi(d_1, \text{may}, p = \rightarrow d) \)
L25: \*q = &obj; // must indicates must be modified if condition meets
L26: \( b_2 = \chi(b_1, \text{must}, q = \rightarrow b) \)
L27: \( e_2 = \chi(e_1, \text{may}, q = \rightarrow e) \)
L28: \( d_3 = \chi(d_2, \text{may}, q = \rightarrow d) \)
L29: }

(1) build SSA including mu chi variable which has already been expanded in top-down phase.
(2) TF(procname, varname) = <(local(var), dep(var), contextcondition)>

local points-to set: explicated points-to relation within current proc, dependence set: formal-in parameters points-to set outside proc.

TF: var can be modified to local or dependence set only if contextcondition holds at the callsite

local(var): put all relevant local points-to set here
dep(var): put all the relevant \( \mu \) set here
contextcondition: put the corresponding \( \chi \) points-to relation here

TF(foo,a) = <\{\phi\},\{(b, q = \rightarrow b), (d, q = \rightarrow d), (e, q = \rightarrow e), p = \rightarrow a\}> (a set depends on b, d, e set)
TF(foo,c) = <\{\phi\},\{(b, q = \rightarrow b), (d, q = \rightarrow d), (e, q = \rightarrow e), p = \rightarrow c\}>
TF(foo,b) = < \{(obj, q = \rightarrow b)\},\{\phi\}, q = \rightarrow b >
TF(foo,e) = < \{(obj, q = \rightarrow e)\},\{\phi\}, q = \rightarrow e >
TF(foo,d) = < \{(obj, q = \rightarrow d)\},\{(b, q = \rightarrow b), (d, q = \rightarrow d), (e, q = \rightarrow e)\}, q = \rightarrow d \lor p = \rightarrow d >
Example: Levpa Analysis Process

**Bottom-up Phase of level 1 (apply TF and build SSA at function main)**

L2: int obj, t;
L3: void foo(int**, int**);
L4: main()
L5: int **x, **y;
L6: int *a, *b, *c, *d,*e;
L7: x1=&a1; y1 =&b1;
L8: \(\mu(b_1, true)\) //true means all conditions meet and will be used at callsite
L9: foo(x1,y1);
L10: points-to set: \{p=>a, q=> b\}
L11: \(a_2 = \chi(a_1, must, true)\);
L12: \(b_2 = \chi(b_1, must, true)\);
L13: \(*b_1 = 5;\)
L14: if(t){ x2 =&c1; y2 =&e1;}
L15: else { x3 = &d1; y3 = &d1;}
L16: \(x_4 = \Phi(x_2,x_3); y_4 = \Phi(y_2,y_3);\)
L17: \(c_1 = &t;\)
L18: \(\mu(d_1, true)\)
L19: \(\mu(e_1, true)\)
L20: foo(x4,y4);
L21: \(c_2 = \chi(c_1, may, true)\)
L22: \(e_2 = \chi(e_1, may, true)\)
L23: \(d_2 = \chi(d_1, may, true)\)
L24: \(*e_1 = 10;\)

(1) Apply TF (use the info in foo and points-to set at the callsite) to create mu chi list for each callsite
(1.1) points-to set of level 2 was propagated last phase (L9: \{p=>a, q=> b\} L17: \{p=>c, p=>d, q=> e, q=>d\})
(1.2) Use the points-to set info at callsite and put into TF will decide if this callsite have read or write side effect
(2) Build SSA form for level 1 variable at callsite
(3) As the main function is root function, no need to compute TF of main
Example: Levpa Analysis Process

Top-down Phase of level 1

L2: int obj, t;
L3: void foo(int**, int**);
L4: main()
L5: {
L6: int **x, **y;
L7: int *a, *b, *c, *d, *e;
L8: x = &a; y1 = &b1;
L9: foo(x1, y1);
L10: *b1 = 5;
L11: if(t) { x2 = &c1; y2 = &e1; }
L12: else { x3 = &d1; y3 = &d1; }
L13: x4 = Φ(x2, x3); y4 = Φ(y2, y3);
L14: c1 = &t;
L15: μ(d1, true) μ(e1, true)
L16: foo(x4, y4);
L17: c2 = χ(c1, may, true) e2 = χ(e1, may, true) d2 = χ(d1, may, true)
L18: *
L19: }
L20: void foo(int **p, int **q){
L21: μ(b, q => a), μ(d, q => d), μ(e, q => e)
L22: *p1 = *q1;
L23: a2 = χ(a1, p => a), c2 = χ(c1, p => c), d2 = χ(d1, p => d)
L24: *
L25: b2 = χ(b1, q => b), d2 = χ(d1, q => d), e2 = χ(e1, q => e)
L26: }

(1) propagate the points-to set to callsite
    there is no callsite use level 1 variable as formal-in paramters, so no operation to propagate to callsite.
(2) points-to set of each variable at level 1 can be found through use-def chain
    entries stores in TF(local + dependence)
Example: Levpa Analysis Process

Bottom-up and Top-down Phase of level 0

```c
t2: int obj, t;
L3: void foo(int**, int**);
L4: main()
L5: {
L6:   int **x, **y;
L7:   int *a, *b, *c, *d,*e;
L8:   x = &a; y = &b;
L9:   foo(x, y);
L10:  *b = 5;
L11:  if(t){ x = &c; y = &e;}
L12:  else { x = &d; y = &d;}
L13:  x = \Phi(x, y); y = \Phi(y, y);
L14:  c = &t;
L15:  \mu(d, true) \mu(e, true)
L16:  foo(x, y);
L17:  c = \chi(c, may, true) e = \chi(e, may, true) d = \chi(d, may, true)
L18:  *e = 10;
L19: }
L20: void foo(int **p, int **q)
L21: {
L22:   \mu(b, q => b), \mu(d, q => d), \mu(e, q => e)
L23:   *p = *q;
L24:   a = \chi(a, p => a), c = \chi(c, p => c), d = \chi(d, p => d)
L25:   *q = &obj;
L26:   b = \chi(b, q => b), d = \chi(d, q => d), e = \chi(e, q => e)
L27: }
```

(1) bottom-up: apply TF for level 0 variable, there is no need to build transfer function build for level 0 variable. no side effect of level build SSA, no side effect so no need to compute transfer function
(2) top-down: there is no callsite use level 0 variable as formal-in paramters so no operation to propagate to callsite, no expand dereference as well