Alias Analysis & Points-to Analysis

Hwansoo Han
May vs. Must Information

“May” information
- The information is true on some path through a CFG

“Must” information
- The information is true on all paths through a CFG

Examples
- \( p = \&x \) on one path: \( p \) may be an alias of \( x \)
- \( P = \&x \) on all paths: \( P \) must be an alias of \( x \)
Flow-insensitive vs. Flow-sensitive

- **Flow-insensitive information**
  - The information is true everywhere on a CFG
  - E.g. “p is an alias of x” holds everywhere

- **Flow-sensitive information**
  - The information is true on a specific point (basic block) of CFG
  - E.g. “p is an alias of x” in block B7.

- **Four possible combinations**

<table>
<thead>
<tr>
<th></th>
<th>May information</th>
<th>Must information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow-insensitive</td>
<td>F.-I. May info.</td>
<td>F.-I. Must info.</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>F.-S. May info.</td>
<td>F.-S. Must info.</td>
</tr>
</tbody>
</table>

3
Alias Analysis

- Two variables are *aliased*
  - If they refer to the same storage location
    - More difficult to analyze
  - Causes in programming constructs
    - Pointers, call-by-reference
    - `union` type (storage overlap)
    - equivalence, common in Fortran

```c
void foo()
{
    int a, b, *x, *y;
    x = y = &a;
    *y = 4;
    bar(b, a, *x);
    cout << a, b, *x;
}

void bar(int& y, int& z, int& w)
{
    y = z + 5;
    w = z + 5;  // w = y OK?
    z = z + 5;  // z = y OK?
}
```
Points-to Analysis (Pointer Analysis)

- Why need to know objects pointed by pointers
  - What is referenced & what is modified
  - Almost the same concept as alias analysis
- Used in various optimization techniques
  - Slicing, dependence graph, constant propagation, code motion, call graph construction

```c
// dependent?
int p->data = 
  = q->data;

// constant?
int x = 0;
int *p = 17;
int y = x + 3;

// hoisting?
for (...) {
  p->data = 0;
}

// who's callee?
(*p)(a,b,c);
q->foo();
```
Flow-Sensitive Alias Analysis

- Forward data flow problem
  - DFA solution provides flow-sensitive information!

- DFI is a set of aliases
  - Aliases are represented by a set of pairs \((p,a)\)
    - \(p\) : a pointer variable
    - \(a\) : any pointer/nonpointer variable
  - \(V = \{v \mid v \subseteq \text{a set of all aliases in the program}\}\)

- DFE

\[
\text{in}[B] = \bigwedge_{P \in \text{Pred}(B)} \text{out}[P] \rightarrow \bigwedge = \bigcup \text{“may” information} \\

f_B = f_{S_n}(\ldots(f_{S_2}(f_{S_1}(\text{in}[B])))\ldots) = \text{out}[B] \quad \text{where } B = (S_1,S_2,\ldots,S_n)
\]
Flow-Sensitive Alias Analysis in C

- **Statement-level transfer functions**
  - Possible types of statements

```c
int x, i, *p, *q, *r, **pp, b[100];
...
S1  p = malloc(int);
S2  q = &x;
S3  pp = &p;
S4  q = &b[i];
S5  r = q;
S6  q = p + i;
S7  p = 0;
S8  *p = x;
S9  *pp = r;
S10 if (q == r)
S11  x = i + 5;
```
Defining Transfer Functions (1)

- **S1**: allocation to dynamic memory \((p=\text{malloc}())\)
  \[ f_{S1}(I) = (I - \{(p,t) \mid \text{any variable } t\}) \cup \{(p, \text{anon}_{S1})\} \]

- **S2**: \&nonpointer-variable to a pointer \((q=&x)\)
  \[ f_{S2}(I) = (I - \{(q,t) \mid \text{any variable } t\}) \cup \{(q,x)\} \]

- **S3**: \&pointer-variable to a pointer \((pp = &p)\)
  \[ f_{S3}(I) = (I - \{(pp,t) \mid \text{any pointer variable } t\}) \cup \{(pp,p)\} \]

- **S4**: \&array-element to a pointer \((q=&b[i])\)
  \[ f_{S4}(I) = (I - \{(q,t) \mid \text{any variable } t\}) \cup \{(q,b[ ])\} \]
Defining Transfer Functions (2)

- **S5**: pointer to pointer \((r=q)\)
  \[ f_{S5}(I) = (I \setminus \{(r,t) \mid \text{any variable } t\}) \cup \{(r,s) \mid (q,s) \in I\} \]

- **S6**: pointer with non-trivial offset to pointer \((q=p+i)\)
  \[ f_{S6}(I) = (I \setminus \{(q,t) \mid \text{any variable } t\}) \]
  \[ \cup \{(q,s) \mid (p,s) \in I \text{ for any array } s\} \]

- **S7**: null or meaningless value to a pointer \((p=null)\)
  \[ f_{S7}(I) = (I \setminus \{(p,t) \mid \text{any variable } t\}) \]

- **S8**: value to *pointer (*p is not a pointer) \((*p=x)\)
  \[ f_{S8}(I) = I \]
Defining Transfer Functions (3)

- S9: value to *pointer (*pp is another pointer) (*pp = r)
  \[ f_{S9}(I) = (I - \{(\star pp, t) \mid \text{any variable } t\}) \cup \{(\star pp, s) \mid (r, s) \in I\} \]

- S10: all other statements which have no effect on pointers
  \[ f_{S10}(I) = I \]

- S11: no effect on pointers, but
  \[ \text{then-part of a condition} \ 'q == r' \ (x = i + 5) \]
  \[ f_{S11}(I) = I - \{(q, t) \mid t \not\in \{t \mid (r, t) \in I\}\} - \{(r, t) \mid t \not\in \{t \mid (q, t) \in I\}\} \]
[ Iterative algorithm for alias analysis ]

for each block B do
  out[B] = \( f_B(\emptyset) \);  // \( \emptyset \) is top of \( \cup \) op
end

while changes to any out[B] occur do
  for each segment B in DFS order do
    in[B] = \( \wedge \) out[P_{B}];
    all predecessors \( P_B \) of \( B \)
    out[B] = \( f_{S_n}(...f_{S_2}(f_{S_1}(in[B])))...); 
  end
end
Solve Example Using DFA

After initialization

1. in[Bi] = ∅ for all blocks Bi
2. out[B2] = \( f_{s3}(f_{s2}(f_{s1}(\emptyset))) = f_{s3}(f_{s2}((q,n))) = \{(q,n)\} \)
3. out[B3] = \( f_{s1}(\emptyset) = \{(p,u)\} \)
4. out[B4] = \( f_{s2}(f_{s1}(\emptyset)) = f_{s2}((p,n)) = \{(p,n),(q,u)\} \)
5. out[B5] = \( f_{s2}(f_{s1}(\emptyset)) = \emptyset \)
6. out[B6] = \( f_{s1}(\emptyset) = \emptyset \)
7. out[B7] = \( f_{s3}(f_{s2}(f_{s1}(\emptyset))) = f_{s3}(f_{s2}(\emptyset)) = f_{s3}(\emptyset) = \emptyset \)

DFS order: B1, B2, B3, B5, B6, B7, B4
After 1st iteration

1. \( \text{in}[B3] = \text{out}[B2] = \{(q,n)\} \)
2. \( \text{out}[B3] = f_{S1}(\{(q,n)\}) = \{(q,n),(p,u)\} \)
3. \( \text{in}[B5] = \text{out}[B3] \cup \text{out}[B4] \cup \text{out}[B7] = \{(q,n),(p,u),(p,n),(q,u)\} \)
4. \( \text{out}[B5] = f_{S2}(f_{S1}(\{(q,n),(p,u),(p,n),(q,u)\})) = \{(q,n),(q,u),(p,u)\} \)
5. \( \text{in}[B7] = \text{out}[B5] = \{(q,n),(q,u),(p,u)\} \)
6. \( \text{out}[B7] = f_{S3}(f_{S2}(f_{S1}(\emptyset))) = f_{S3}(f_{S2}(\{(q,n),(q,u),(p,u)\})) = \{(q,n),(p,u),(p,n),(q,u)\} \)
7. \( \text{in}[B4] = \text{out}[B2] = \{(q,n)\} \)
8. \( \text{out}[B4] = f_{S2}(f_{S1}(\{(q,n)\})) = f_{S2}(\{(p,n),(q,n)\}) = \{(p,n),(q,u)\} \)
Flow-Insensitive Points-to Analysis

- Subset-based algorithm
  - Inclusion-based
    - For $e_1 = e_2$, $\text{points-to}(e_1) \supseteq \text{points-to}(e_2)$
    - Precise but expensive

- Equality-based algorithm
  - Unification-based
    - For $e_1 = e_2$, $\text{points-to}(e_1) = \text{points-to}(e_2)$
    - Imprecise but fast

```
a = &b;
b = &c;
a = &d;
d = &e;
```
Andersen’s Analysis (Subset-based)

\[
\begin{align*}
\text{a} &= \&\text{b} \\
\text{b} &= \&\text{c} \\
\text{x} &= \&\text{y} \\
\text{y} &= \&\text{z} \\
\text{a} &= \text{x} \\
\end{align*}
\]

\[
\begin{align*}
\text{pt}(\text{a}) &\supseteq \{ \text{b} \} \\
\text{pt}(\text{b}) &\supseteq \{ \text{c} \} \\
\text{pt}(\text{x}) &\supseteq \{ \text{y} \} \\
\text{pt}(\text{y}) &\supseteq \{ \text{z} \} \\
\text{pt}(\text{a}) &\supseteq \text{pt}(\text{x})
\end{align*}
\]

\[
\begin{align*}
\text{pt}(\text{a}) &= \{ \text{b}, \text{y} \} \\
\text{pt}(\text{b}) &= \{ \text{c} \} \\
\text{pt}(\text{x}) &= \{ \text{y} \} \\
\text{pt}(\text{y}) &= \{ \text{z} \}
\end{align*}
\]

[Anderson’94]
Steensgaard’s Analysis (Equality-based)

\[
\begin{array}{c}
\text{a = &b} \\
\text{b = &c} \\
\text{x = &y} \\
\text{y = &z} \\
\text{a = x}
\end{array}
\]

\[
\begin{array}{c}
\text{pt(a) = \{ b \}} \\
\text{pt(b) = \{ c \}} \\
\text{pt(x) = \{ y \}} \\
\text{pt(y) = \{ z \}} \\
\text{pt(a) = pt(x)}
\end{array}
\]

\[
\begin{array}{c}
\text{pt(a) = \{ b, y \}} \\
\text{pt(b) = \{ c \}} \\
\text{pt(x) = \{ b, y \}} \\
\text{pt(y) = \{ z \}}
\end{array}
\]

\[
\begin{array}{c}
\text{pt(a) = \{ b, y \}} \\
\text{pt(b) = \{ c, z \}} \\
\text{pt(x) = \{ b, y \}} \\
\text{pt(y) = \{ c, z \}}
\end{array}
\]

[Steensgaard’96]
B. Steensgaard. Points-to analysis in almost linear time, POPL 1996.
Pointer Analysis Dimension

- Flow sensitivity
- Context sensitivity
- Heap modeling
- Aggregate modeling
- Alias representation
- Whole program
Heap Modeling

- Allocation site
- Connection analysis
- Shape analysis
Aggregate Modeling

- **Array vs. element**

- **Structure/Object vs. field**
Alias Representation

- Points-to relations
  - \(<a, b>\) \(<b, c>\)

- Explicit alias representations
  - \(<\ast a, b>\) \(<\ast\ast a, c>\) \(<\ast b, c>\) \(<\ast\ast a, \ast b>\)

- Precision/efficiency trade-offs exist
Flow Sensitivity

- **Flow-sensitive (FS)**
  - Information at each instruction
  - More precise, less efficient (time and space)

- **Flow-insensitive (FI)**
  - Information about whole program or function (merged information from all instructions)
  - Less precise, more efficient

```c
1: p = malloc();
2: q = malloc();
3: fp = &p;
4: fp = &q;
5: p = malloc();
6: ... = *p
```

- `flow sensitive` (points-to info. at 6)
- `flow insensitive`

- `subset-based`
- `equality-based`
Context Sensitivity

- Context-sensitive (CS)
  - Different information for different call sites and parameters

- Context-insensitive (CI)
  - Merged information from all call sites with all possible parameters

```c
main() {
  1: f();
  2: p = malloc();
  3: g();
}

f() {
  4: p = malloc();
  5: g();
}
g() {
  ...
}
```

*p \rightarrow \{\text{heap2}\}, \{\text{heap4}\}, \text{or} \{\text{heap2, heap4}\}?
Precision vs. Scalability

- Equality-based can analyze 1 Million Lines of Code
  - Getting more precise
- Subset-based more precise, but haven’t scaled well
  - But getting more efficient
- More precise/expensive pointer analysis can make clients more efficient  [ISSTA’00, Which pointer analysis should I use?]
  - Need to consider efficiency including clients!
Satisfying Clients

- Precision/efficiency required depends on client
- Flow-sensitive analysis does NOT provide significant precision improvement over subset-based flow-insensitive
- Only a few programs out of 23 show differences
Satisfying Clients (cont’d)

- For optimizations
  - Current FICI may be sufficient
Summary

- **Alias/Pointer analysis**
  - Effective information for later optimizations

- **Precision and efficiency trade-off**
  - Flow-insensitive vs. flow-sensitive
  - Context-insensitive vs. context-sensitive

- **Flow-insensitive, context-insensitive analysis**
  - Subset-based analysis, equality-based analysis
  - Both provide sufficient precision for most optimizations