CS 526
Advanced Compiler Construction

https://charithm.web.illinois.edu/cs526/sp2022/
(slides adapted from Sasa and Vikram)
POINTER ANALYSIS

The slides adapted from Vikram Adve
Course

So far:
• Dataflow analysis (examples and theory)
• Dependency analysis
• SSA (sparse dataflow analysis via def-use chains)

Coming up next:
• Pointer analysis (generalize the dependence relationship)
• Interprocedural analysis (how to analyze function calls?)
• Vectorization
• ML in compilers
POINTER ANALYSIS

The slides adapted from Vikram Adve
Pointer Analysis

Pointer and Alias Analysis are fundamental to reasoning about heap manipulating programs (pretty much all programs today).

- **Pointer Analysis:**
  - What objects does each pointer points to?
  - Also called points-to analysis

- **Alias Analysis:**
  - Can two pointers point to the same location?
  - Client of pointer analysis
Example

X = 1
P = &X
*P = 2
return X

// What is the value of X?
Aliases

Consider references r1 or r2,
• may be of the form “x” or “*p” “**p”,“(p)->q->i”...
• We assume C notation for dereferencing pointers (*, ->)

**Alias:** r1 are r2 are aliased if the memory locations accessed by r1 and r2 overlap.

**Alias Relation:** A set of ordered pairs {(ri, rj)} denoting aliases that may hold at a particular point in a program.
• Sometimes called a may-alias relation.

**May or Must:** A kind of aliasing if it happens optionally or always
• May: e.g., depending on the control flow: if (b) { p = &q; }
• Must: determined that they always represent aliases
Aliases

We look at the language that extends the simple expressions with the additional pointer-like structures:

\[
p := \&x \\
| p := q \\
| *p := q \\
| p := *q
\]

Consider references \( r1 \) or \( r2 \),

- may be of the form “\( x \)” or “\( *p \)” “\( **p \)” “\( (*p)\rightarrow q\rightarrow i \)” …
- We assume C notation for dereferencing pointers (\( *, \rightarrow \))
Example

\[ X = 1 \]
\[ P = \&X \]
\[ Q = P \]
\[ P = 2 \]
Example

\[ X = 1 \]
\[ P = \& X \]
\[ Q = P \]
\[ \ast P = 2 \]

**Aliasing pairs**

\[ \text{Aliasing pairs: } \{ (\ast P, X), (\ast Q, X), (\ast P, \ast Q) \} \]

**Alias:** \( r_1 \) and \( r_2 \) are aliased if the memory locations accessed by \( r_1 \) and \( r_2 \) overlap.
Points-To Facts

Points-to Pair: pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) may hold the address of one of the memory locations of \(r_2\).

- Also written: \(r_1 \rightarrow r_2\), means “\(r_1\) points to \(r_2\)”.

Points-to Set: \(\{(r_i, r_j)\}\): A set of points-to pairs that may hold at a particular point in a program.

Points-To Graph: A directed graph where
- **Nodes** represents one or more memory objects;
- Each **Edge** \(p \rightarrow q\) means some object in the node \(p\) may hold a pointer to some object in the node \(q\).
**Example**

\[
X = 1 \\
P = \&X \\
Q = P \\
*P = 2
\]

**Points-to Pair:** pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) An ordered may hold the address of one of the memory locations of \(r_2\).

**Points-to pairs**

\[
// (P, X) \\
// \{ (P, X), (Q, X) \}
\]
Example

\[ X = 1 \]
\[ P = \&X \]
\[ Q = P \]
\[ R = Q \]

Points-to Pair: pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) An ordered may hold the address of one of the memory locations of \(r_2\).

Points-to pairs

// \((P, X)\)

// \{(P, X), (Q, X)\}

// \{(P, X), (Q, X), (R, X)\}

“Short notation”: vs the long one that would list all the aliases.
Challenges of Points-To Analysis

- **Pointers to pointers**, which can occur in many ways: take address of pointer; pointer to structure containing pointer; pass a pointer to a procedure by reference
- **Aggregate objects**: structures and arrays containing pointers
- **Recursive data structures** (lists, trees, graphs, etc.) closely related problem: anonymous heap locations
- **Control-flow**: analyzing different data paths
- **Interprocedural**: a location is often accessed from multiple functions; a common pattern (e.g., pass by reference)
- Compile-time cost
  - Number of variables, $|V|$, can be large
  - Number of alias pairs at a point can be $O(|V|^2)$
Common Simplifying Assumptions

**Aggregate objects:** arrays (and perhaps structures) containing pointers

**Simple solution:** Treat as a single big object!

- Commonplace for arrays.
- Not a good choice for structures?
  - See Intel Paper (Ghiya, Lavery & Sehr, PLDI 2001)
- Pointer arithmetic is only legal for traversing an array:
  
  \[ q = p \pm i \] and \[ q = \&p[i] \] are handled the same as \[ q = p \]
Common Simplifying Assumptions

Recursive data structures (lists, trees, graphs, …)

Solution: Compute aliases, not “shape”

• Don’t prove something is a linked-list or a binary tree (leave that for shape analysis)

• k-limiting: only track k or fewer levels of dereferencing

• Use simplified naming schemes for heap objects (later slide)
Common Simplifying Assumptions

**Control-flow**: analyzing different data paths blows up the analysis time/space

**Solution(?)**: Could ignore the issue and compute a single common result for any path!

**No consensus on this issue!** (Will discuss later)
Naming Schemes for Heap Objects

The Naming Problem: Example 1

```cpp
int main() {
    // Two distinct objects
    T* p = create(n);
    T* q = create(m);
}

T* create(int num) {
    // Many objects allocated here
    return new T(num);
}
```

Q. Should we try to distinguish the objects created in main()?
The Naming Problem: Example 2

T* makelist(int len) {
    T* newObj = new T; // Many distinct objects
         // allocated here
    newObj->next = (--len == 0)? NULL :
                   makelist(len);
}

Q. Can we distinguish the objects created in makelist()?
Possible Naming Abstractions

$\mathbf{H}_0$: One name for the entire heap

$\mathbf{H}_T$: One name per type $T$ (for type-safe languages)

$\mathbf{H}_L$: One name per heap allocation site $L$ (line number)

$\mathbf{H}_C$: One name per (acyclic) call path $C$ ("cloning")

$\mathbf{H}_F$: One name per immediate caller $F$ or call-site ("one-level cloning")
Flow-Sensitivity of Analysis

Def. A flow-sensitive analysis is one that computes a distinct result for each program point. A flow-insensitive analysis generally computes a single result for an entire procedure or an entire program.

A flow-insensitive algorithm effectively ignores the order of statements!

```c
int f(T q, T r){
    T* p;
    ...
    p = &q;
    ...
    p = &r;
}
```
Flow-Sensitivity of Analysis

Def. A flow-sensitive analysis is one that computes a distinct result for each program point. A flow-insensitive analysis generally computes a single result for an entire procedure or an entire program.

A flow-insensitive algorithm effectively ignores the order of statements!

```
int f(T q, T r){
    T* p;
    if (...)
        p = &q;
    else
        p = &r;
}
```
Flow-Sensitivity of Analysis

Pointer Analysis

• **Flow-sensitive**: At program point n, compute alias pairs \(<a, b>\) that may hold at n for some path from program entry to n.

• **Flow-insensitive**: Compute all alias pairs \(<a, b>\) such that a may be aliased to b at some point in a program (or function).

Important special cases

• Local scalar variables: SSA form gives flow-sensitivity
• Malloc or new: Allocates “fresh” memory, i.e., no aliases
• Read-only fields: e.g., array length
Realizable Paths

Definition: Realizable Path
A program path is realizable iff every procedure call on the path returns control to the point where it was called (or to a legal exception handler or program exit)

Whole-program Control Flow Graph?
Conceptually extend CFG to span whole program:
• split a call node in CFG into two nodes: CALL and RETURN
• add edge from CALL to ENTRY node of each callee
• add edge from EXIT node of each callee to RETURN
Problem: This produces many unrealizable paths

Focusing only on realizable paths requires context-sensitive analysis
**Context-Sensitivity of Analysis**

**Def.** A context-sensitive interprocedural analysis computes results that may hold only for realizable paths through the program. Otherwise, the analysis is context-insensitive.

```c
T* identity(T* p) {
    return p;
}

void f1() {
    T* p1 = new T; // Object o1
    T* q1 = identity(p1);
}

void f2() {
    T* p2 = new T; // Object o2
    T* q2 = identity(p2);
}
```

**Diagram:**
- **Context Insensitive**
  - q1 → o1
  - q1 → o2
- **Context Sensitive**
  - q2 → o1
  - q2 → o2
Context-Sensitivity of Analysis

**Pointer Analysis**

Apply the definitions directly using points-to pairs \(<a, b>\). But important variations exist:

- Heap cloning vs. no cloning: Cloning gives greater context-sensitivity
- Bottom-up vs. top-down: Does final result for a procedure include only “realizable” behavior from all contexts?
- Handling of recursive functions: Does analysis retain context-sensitivity within SCCs in the call graph?

**Object Sensitivity:** Context represents each allocation site. Typically offers quite precise context analysis

[Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java; Milanova et al. ISSTA 2002]
Field-Sensitivity of Analysis

**Def.** A field-sensitive analysis is one that tracks distinct behavior for individual fields of a record type. Otherwise, it is field-insensitive.

```c
int f(T q, T r) {
    p.a = &q;
    p.b = &r;
}
```

**Challenges**

- **Complexity:** For certain analysis techniques, converts linear representation to worse (perhaps even exponential).
- **Non-type-safe programs:** May have to track behavior at every byte offset within the structure (not just each field).
Flow Insensitive Algorithms

3 popular algorithms

• Any address
• Andersen, 1994
• Steensgaard, 1996

Acceptable precision in practice for compiler optimization, however perhaps insufficient for static analysis approaches for security, reliability, or bug finding
Any Address Analysis

- **Single points-to set**: contains all variables whose address is taken, passed by reference, etc.

- **Any pointer may point** to **any variable** in this set

- Simple, fast, linear-time algorithm

- Common choice for function pointers, and for global variables, e.g., for initial call graph

- Can refine with splitting by types
Example 1

```c
void main() {
    T *p, *q, *r;
    T t;

    o1:p = new T; // {p} -> {o1}
    q = &t; // {p,q} -> {o1,t}
    r = q; // {p,q,r} -> {o1,t}
}
```
Andersen’s Algorithm

• Generally the most precise flow- and context-insensitive algorithm
• Compute a single points-to graph for entire program
• Refinement by Burke: Separate points-to graph for each function
• Cost is $O(n^3)$ for program with n assignments
  • McAlister, On the complexity analysis of static analyses (SAS’99)
  • Sridharan and Fink, The Complexity of Andersen’s Analysis in Practice (SAS’09)
Andersen’s Algorithm: Conceptual

Initialize: Points-to graph with a separate node per variable

Iterate until convergence:
At each statement, evaluate the appropriate rule:

<table>
<thead>
<tr>
<th>Form</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = &amp;x$</td>
<td>Add $p \rightarrow x$</td>
</tr>
<tr>
<td>$p = q$</td>
<td>$\forall x :$ if $q \rightarrow x$, add $p \rightarrow x$</td>
</tr>
<tr>
<td>$*p = q$</td>
<td>$\forall x, r:$ if $q \rightarrow x$ and $p \rightarrow r$, add $r \rightarrow x$</td>
</tr>
<tr>
<td>$p = *q$</td>
<td>$\forall x, r:$ if $q \rightarrow x$ and $x \rightarrow r$, add $p \rightarrow r$</td>
</tr>
</tbody>
</table>
Andersen’s Algorithm: Actual

1. Build initial "inclusion constraint graph" and initial points-to sets
2. Iterate until converged:
   • Update constraint graph for new points-to pairs
   • Update the points-to sets according to new constraints

Inclusion Constraint Graph: Add constraint for pointer assignments (pts is points-to set):

<table>
<thead>
<tr>
<th>Name</th>
<th>Form</th>
<th>Constraint</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to pair</td>
<td>p = &amp;x</td>
<td>p ⊇ {x}</td>
<td>pts(p) U= {x}</td>
</tr>
<tr>
<td>Direct constraint</td>
<td>p = q</td>
<td>p ⊇ q</td>
<td>pts(p) U= pts(q)</td>
</tr>
<tr>
<td>Indirect constraint</td>
<td>*p = q</td>
<td>*p ⊇ q</td>
<td>for v ∈ pts(p):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pts(v) U= pts(q)</td>
</tr>
<tr>
<td>Indirect constraint</td>
<td>p = *q</td>
<td>p ⊇ *q</td>
<td>for v ∈ pts(q):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pts(p) U= pts(v)</td>
</tr>
</tbody>
</table>
Example 1 Revisited

```c
void main() {
    T *p, *q, *r;
    T t;

    o1:p = new T; // {p} -> {o1}
    q = &t;      // {p} -> {o1}, {q} -> {t}
    r = q;       // {r} -> {t}
}
```
Example 2

```c
void f(int i) {
    T *p, *q, *r;

    o1:p = new T;
    // {p} -> {o1}

    o2:q = new T;
    // {q} -> {o2}

    if (i>0)
        r = p;
    else
        r = q;

    // {r} -> {o1}
    // {r} -> {o2}
}
```

<table>
<thead>
<tr>
<th>Form</th>
<th>Constraint</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = &amp;x</td>
<td>p ⊇ {x}</td>
<td>pts(p) U= {x}</td>
</tr>
<tr>
<td>p = q</td>
<td>p ⊇ q</td>
<td>pts(p) U= pts(q)</td>
</tr>
</tbody>
</table>
| *p = q   | *p ⊇ q     | for v ∈ pts(p):
|          |            |     pts(v) U= pts(q) |
| p = *q   | p ⊇ *q     | for v ∈ pts(q):
|          |            |     pts(p) U= pts(v) |
Example 3

\[ \begin{align*}
    p &= &a; &\quad &\text{// } p \to \{a\} &\quad &\text{// } p \to \{a\} \\
    s &= &p; &\quad &\text{// } s \to \{p\} &\quad &\text{// } s \to \{p,q\} \\
    r &= &s; &\quad &\text{// } r \to \{a\} &\quad &\text{// } r \to \{a,b\} \\
    q &= &b; &\quad &\text{// } q \to \{b\} &\quad &\text{// } q \to \{t\} \\
    s &= &q; &\quad &\text{// } s \to \{p,q\} &\quad &\text{// } s \to \{p,q\} \\
\end{align*} \]

Done?  

Done?
Andersen’s Algorithm: Cycles

Cycle in constraint graph:
\[ \text{pts}(p) \supseteq \text{pts}(q) \supseteq \text{pts}(r) \supseteq \text{pts}(p) \]
\[ \Rightarrow \text{pts}(p) = \text{pts}(q) = \text{pts}(r) = \text{pts}(p) \]
\[ \Rightarrow \text{No need to propagate points-to pairs around such cycles!} \]
Andersen’s Algorithm: **Cycles**

**Cycle in constraint graph:**

\[ \text{pts}(p) \supseteq \text{pts}(q) \supseteq \text{pts}(r) \supseteq \text{pts}(p) \]

\[ \Rightarrow \text{pts}(p) = \text{pts}(q) = \text{pts}(r) = \text{pts}(p) \]

\[ \Rightarrow \text{No need to propagate points-to pairs around such cycles!} \]

**Offline cycle elimination:**

- Find cycles due to direct pointer copies (direct constraints)
- Collapse each cycle into a single node, reduces size of constraint graph
- But many more cycles can be induced by indirect constraint edges: we need cycle elimination during transitive closure (**"online"**) 


**Online cycle elimination:**

- Fähndrich, Foster, Aiken and Su (PLDI ’98): Cycle elimination is essential for scalability.
- Heintze and Tardieu (PLDI 2001): "A million lines of code per second."
- Hardekopf and Lin (PLDI 2007)
Steensgaard’s Algorithm

Unification:

• Conceptually: restrict every node to only one outgoing edge (on the fly)
• If $p \rightarrow x$ and $p \rightarrow y$, merge $x$ and $y$ ("unify")
• All objects “pointed to” by $p$ one equivalence class

\[
\begin{align*}
A &= \&B \\
B &= \&C \\
A &= \&D \\
D &= \&E
\end{align*}
\]
Steensgard’s Algorithm

Unification: Conceptually: restrict every node to only one outgoing edge (on the fly)
• If $p \rightarrow x$ and $p \rightarrow y$, merge $x$ and $y$ (“unify”)
• All objects “pointed to” by $p$ form one equivalence class

Algorithm
1. For each statement, merge points-to sets:
   
   $p = q$: Merge two equivalence classes ($p$’s and $q$’s targets)
   Less expensive than computing points-to iterations
   This may cause other nodes to collapse!

2. Use Tarjan’s “union-find” (disjoint-set) data structure to record equivalence classes
Steensgard’s Algorithm

“Union-find” aka Disjoint Set data structure:
• Splits the set of elements into disjoint partitions
• Maintains the partition with every addition
• Operations:
  • Find(x): follows parent pointers from x until reaching root (i.e. finds the set containing x)
  • Union(x,y): 1) finds the roots of x,y; 2) merges the trees by connecting the root nodes. (i.e. merges the sets)
• Properties: addition and merge of sets in near constant time, i.e. \( \alpha(n) \) – inverse Ackerman func. \( \alpha(n) < 4 \) even for large n.

Consequence for Steensgard’s analysis:
• Non-iterative algorithm, almost-linear running time: \( O(n\alpha(n)) \)
• Like Andersen, single solution for the entire program
Steensgard vs. Andersen

Consider assignment $p = q$, i.e., only $p$ is modified, not $q$.

**Subset-based Algorithms** (Anderson’s algorithm is an example)
- Add a constraint: Targets of $q$ must be subset of targets of $p$
- Graph of such constraints is also called “inclusion constraint graphs”
- Enforces unidirectional flow from $q$ to $p$

**Unification-based Algorithms** (Steensgard is an example)
- Merge equivalence classes: targets of $p$ and $q$ must be identical
- Assumes bidirectional flow from $q$ to $p$ and vice-versa

**In-between solutions:**
- Unification-based Pointer Analysis with Directional Assignment, Das, PLDI 2000 – exploits the semantics of C; uses Andersen for top pointers, Steensgard elsewhere
Alias Analysis

- Alias analysis is a common client of pointer (points-to) analysis
  - **Pointer Analysis**: What objects does each pointer points to?
  - **Alias Analysis**: Can two pointers point to the same location? (i.e., it is possible that *p = *q)

- Once we have performed the pointer analysis, it is trivial to compute alias analysis (but not vice versa)

- Two pointers p and q may alias if \( \text{points-to}(p) \cap \text{points-to}(q) \neq \emptyset \)
Which Pointer Analysis To Use?
Hind & Pioli, ISSTA, Aug. 2000

Compared 5 algorithms (4 flow-insensitive, 1 flow-sensitive):
• Any address
• Steensgard
• Anderson
• Burke (like Anderson, but separate solution per procedure)
• Choi et al. (flow-sensitive)

Metrics
1. Precision: number of alias pairs
2. Precision of important optimizations: MOD/REF, REACH, LIVE, flow dependences, constant prop.
3. Efficiency: analysis time/memory, optimization time/memory

Benchmarks: 23 C programs, including some from SPEC benchmarks
Which Pointer Analysis To Use?

I. Precision: (Table 2)

- Steensgard much better than Any-Address (6x on average)
- Anderson/Burke significantly better than Steensgard (about 2x)
- Choi negligibly better than Anderson/Burke

Table 2: Mod and Ref at pointer dereferences and all CFG nodes. No assignments through a pointer occur in compiler. “AT” = Address Taken, “St” = Steensgaard’s, “A/B” = Andersen/Burke et al., “Ch” = Choi et al.

<table>
<thead>
<tr>
<th>Name</th>
<th>Pointer Mod</th>
<th>Pointer Ref</th>
<th>Mod</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AT</td>
<td>St</td>
<td>A/B</td>
<td>Ch</td>
</tr>
<tr>
<td>allroots</td>
<td>3</td>
<td>2.00</td>
<td>1.00</td>
<td>–</td>
</tr>
<tr>
<td>052.sivinn</td>
<td>14</td>
<td>1.00</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>01.qbsort</td>
<td>12</td>
<td>2.00</td>
<td>1.50</td>
<td>–</td>
</tr>
<tr>
<td>06.matx</td>
<td>15</td>
<td>3.00</td>
<td>2.22</td>
<td>–</td>
</tr>
<tr>
<td>15.trie</td>
<td>10</td>
<td>1.12</td>
<td>–</td>
<td>1.00</td>
</tr>
<tr>
<td>04.bisect</td>
<td>14</td>
<td>1.15</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>fixoutput</td>
<td>9</td>
<td>1.80</td>
<td>–</td>
<td>9</td>
</tr>
<tr>
<td>17.bintr</td>
<td>7</td>
<td>1.00</td>
<td>–</td>
<td>7</td>
</tr>
<tr>
<td>anagram</td>
<td>17</td>
<td>1.00</td>
<td>–</td>
<td>17</td>
</tr>
<tr>
<td>ks</td>
<td>17</td>
<td>1.90</td>
<td>1.86</td>
<td>1.62</td>
</tr>
<tr>
<td>03.eks</td>
<td>12</td>
<td>1.22</td>
<td>–</td>
<td>12</td>
</tr>
<tr>
<td>08.main</td>
<td>13</td>
<td>6.00</td>
<td>3.27</td>
<td>2.61</td>
</tr>
<tr>
<td>09.vor</td>
<td>19</td>
<td>1.85</td>
<td>3.35</td>
<td>1.32</td>
</tr>
<tr>
<td>loader</td>
<td>47</td>
<td>3.77</td>
<td>2.23</td>
<td>–</td>
</tr>
<tr>
<td>129.compress</td>
<td>13</td>
<td>1.40</td>
<td>1.07</td>
<td>–</td>
</tr>
<tr>
<td>ft</td>
<td>10</td>
<td>2.87</td>
<td>1.80</td>
<td>1.72</td>
</tr>
<tr>
<td>football</td>
<td>32</td>
<td>6.00</td>
<td>2.10</td>
<td>–</td>
</tr>
<tr>
<td>compiler</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>assembler</td>
<td>87</td>
<td>1.24</td>
<td>2.21</td>
<td>–</td>
</tr>
<tr>
<td>yacc2</td>
<td>48</td>
<td>1.14</td>
<td>1.11</td>
<td>–</td>
</tr>
<tr>
<td>simulator</td>
<td>87</td>
<td>3.16</td>
<td>2.05</td>
<td>–</td>
</tr>
<tr>
<td>flex</td>
<td>56</td>
<td>5.37</td>
<td>1.78</td>
<td>–</td>
</tr>
<tr>
<td>099.go</td>
<td>154</td>
<td>42.68</td>
<td>13.64</td>
<td>154</td>
</tr>
</tbody>
</table>

Average 30.26 4.03 2.06 2.02 30.70 4.87 2.35 2.29 2.50 1.04 0.871 0.867 4.48 1.75 1.540 1.536
Which Pointer Analysis To Use?

2. MOD/REF precision: (Table 2)

- Steensgard much better than Any-Address (2.5x on average)
- Anderson/Burke significantly better than Steensgard (15%)
- Choi very slightly better than Anderson/Burke (1%)

<table>
<thead>
<tr>
<th>Name</th>
<th>AT</th>
<th>St</th>
<th>A/B</th>
<th>Ch</th>
<th>AT</th>
<th>St</th>
<th>A/B</th>
<th>Ch</th>
<th>AT</th>
<th>St</th>
<th>A/B</th>
<th>Ch</th>
<th>AT</th>
<th>St</th>
<th>A/B</th>
</tr>
</thead>
<tbody>
<tr>
<td>allroots</td>
<td>3</td>
<td>2.00</td>
<td>1.00</td>
<td></td>
<td>3</td>
<td>2.00</td>
<td>1.38</td>
<td></td>
<td>.88</td>
<td>.85</td>
<td></td>
<td>.83</td>
<td></td>
<td>1.77</td>
<td>1.58</td>
</tr>
<tr>
<td>052.aiwinn</td>
<td>14</td>
<td>1.00</td>
<td></td>
<td></td>
<td>14</td>
<td>1.00</td>
<td></td>
<td></td>
<td>1.17</td>
<td>.51</td>
<td></td>
<td></td>
<td>2.24</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>01.qbt</td>
<td>12</td>
<td>2.00</td>
<td>1.50</td>
<td></td>
<td>12</td>
<td>1.76</td>
<td></td>
<td></td>
<td>.95</td>
<td>.46</td>
<td></td>
<td></td>
<td>3.39</td>
<td>.94</td>
<td></td>
</tr>
<tr>
<td>06.mtx</td>
<td>15</td>
<td>3.00</td>
<td>2.22</td>
<td></td>
<td>15</td>
<td>3.25</td>
<td>3.12</td>
<td></td>
<td>1.09</td>
<td>.39</td>
<td>34</td>
<td></td>
<td>1.48</td>
<td>.89</td>
<td>.88</td>
</tr>
<tr>
<td>15.trie</td>
<td>10</td>
<td>1.12</td>
<td>1.00</td>
<td></td>
<td>10</td>
<td>1.00</td>
<td></td>
<td></td>
<td>1.02</td>
<td>.58</td>
<td></td>
<td>.52</td>
<td></td>
<td>4.42</td>
<td>.88</td>
</tr>
<tr>
<td>04.value</td>
<td>14</td>
<td>1.15</td>
<td></td>
<td></td>
<td>14</td>
<td>1.00</td>
<td></td>
<td></td>
<td>2.57</td>
<td>.58</td>
<td></td>
<td></td>
<td>3.92</td>
<td>1.57</td>
<td></td>
</tr>
<tr>
<td>fixoutput</td>
<td>9</td>
<td>1.80</td>
<td></td>
<td></td>
<td>9</td>
<td>2.00</td>
<td></td>
<td></td>
<td>.74</td>
<td>.37</td>
<td></td>
<td></td>
<td>.78</td>
<td>.56</td>
<td></td>
</tr>
<tr>
<td>17.bintr</td>
<td>7</td>
<td>1.00</td>
<td></td>
<td></td>
<td>7</td>
<td>1.20</td>
<td></td>
<td></td>
<td>.62</td>
<td>.30</td>
<td></td>
<td></td>
<td>2.07</td>
<td>.71</td>
<td></td>
</tr>
<tr>
<td>anagram</td>
<td>17</td>
<td>1.00</td>
<td></td>
<td></td>
<td>17</td>
<td>1.10</td>
<td></td>
<td></td>
<td>.90</td>
<td>.45</td>
<td></td>
<td></td>
<td>2.38</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>ks</td>
<td>17</td>
<td>1.90</td>
<td>1.86</td>
<td>1.62</td>
<td>17</td>
<td>1.79</td>
<td></td>
<td>1.74</td>
<td>1.70</td>
<td>.56</td>
<td>.55</td>
<td>.53</td>
<td></td>
<td>3.76</td>
<td>1.35</td>
</tr>
<tr>
<td>08.eks</td>
<td>12</td>
<td>1.22</td>
<td></td>
<td></td>
<td>12</td>
<td>1.02</td>
<td></td>
<td></td>
<td>1.83</td>
<td>.50</td>
<td></td>
<td></td>
<td>3.54</td>
<td>1.63</td>
<td></td>
</tr>
<tr>
<td>08.main</td>
<td>12</td>
<td>8.00</td>
<td>3.27</td>
<td>2.61</td>
<td>12</td>
<td>5.14</td>
<td>4.61</td>
<td>3.59</td>
<td>1.75</td>
<td>.63</td>
<td></td>
<td></td>
<td>5.35</td>
<td>1.32</td>
<td>1.30</td>
</tr>
<tr>
<td>09.vor</td>
<td>19</td>
<td>1.85</td>
<td>1.35</td>
<td>1.32</td>
<td>19</td>
<td>1.92</td>
<td>1.68</td>
<td>1.60</td>
<td>2.04</td>
<td>.62</td>
<td></td>
<td></td>
<td>7.91</td>
<td>1.40</td>
<td>1.34</td>
</tr>
<tr>
<td>loader</td>
<td>47</td>
<td>3.77</td>
<td>2.23</td>
<td></td>
<td>47</td>
<td>2.09</td>
<td>1.36</td>
<td></td>
<td>5.08</td>
<td>.90</td>
<td>73</td>
<td></td>
<td>9.72</td>
<td>1.78</td>
<td>1.30</td>
</tr>
<tr>
<td>129.compress</td>
<td>13</td>
<td>1.40</td>
<td>1.07</td>
<td></td>
<td>13</td>
<td>2.26</td>
<td>1.11</td>
<td></td>
<td>1.68</td>
<td>.80</td>
<td>78</td>
<td></td>
<td>1.66</td>
<td>1.29</td>
<td>1.28</td>
</tr>
<tr>
<td>ft</td>
<td>10</td>
<td>2.87</td>
<td>1.80</td>
<td>1.72</td>
<td>10</td>
<td>2.65</td>
<td>2.53</td>
<td>2.39</td>
<td>2.14</td>
<td>.90</td>
<td>74</td>
<td>.73</td>
<td></td>
<td>2.58</td>
<td>1.31</td>
</tr>
<tr>
<td>football</td>
<td>32</td>
<td>6.00</td>
<td>2.10</td>
<td></td>
<td>32</td>
<td>3.26</td>
<td>1.54</td>
<td></td>
<td>1.37</td>
<td>.70</td>
<td>61</td>
<td></td>
<td>4.55</td>
<td>1.83</td>
<td>1.65</td>
</tr>
<tr>
<td>compiler</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>1.00</td>
<td></td>
<td></td>
<td>3.38</td>
<td></td>
<td></td>
<td></td>
<td>4.45</td>
<td>4.44</td>
<td></td>
</tr>
<tr>
<td>assembler</td>
<td>87</td>
<td>1.24</td>
<td>2.21</td>
<td></td>
<td>87</td>
<td>15.14</td>
<td>2.11</td>
<td></td>
<td>1.21</td>
<td>1.88</td>
<td>.87</td>
<td></td>
<td>15.07</td>
<td>4.09</td>
<td>1.47</td>
</tr>
<tr>
<td>yecc2</td>
<td>48</td>
<td>1.14</td>
<td>1.11</td>
<td></td>
<td>48</td>
<td>1.08</td>
<td>1.02</td>
<td></td>
<td>5.32</td>
<td>.53</td>
<td>52</td>
<td></td>
<td>7.80</td>
<td>1.65</td>
<td></td>
</tr>
<tr>
<td>simulator</td>
<td>87</td>
<td>3.16</td>
<td>2.05</td>
<td></td>
<td>87</td>
<td>3.95</td>
<td>1.86</td>
<td></td>
<td>6.82</td>
<td>.62</td>
<td>57</td>
<td></td>
<td>8.21</td>
<td>1.21</td>
<td>1.06</td>
</tr>
<tr>
<td>flex</td>
<td>56</td>
<td>5.37</td>
<td>1.78</td>
<td></td>
<td>56</td>
<td>5.09</td>
<td>2.03</td>
<td>2.01</td>
<td>5.97</td>
<td>1.60</td>
<td>1.18</td>
<td></td>
<td>1.55</td>
<td>3.89</td>
<td>3.44</td>
</tr>
<tr>
<td>099.go</td>
<td>154</td>
<td>42.68</td>
<td>13.64</td>
<td></td>
<td>154</td>
<td>51.39</td>
<td>17.03</td>
<td></td>
<td>7.31</td>
<td>5.87</td>
<td>3.94</td>
<td></td>
<td>4.47</td>
<td>3.98</td>
<td>3.13</td>
</tr>
</tbody>
</table>

Average: 30.26 4.03 2.06 2.02 30.70 4.87 2.35 2.29 2.50 1.04 0.871 0.867 4.48 1.75 1.540 1.536
Which Pointer Analysis To Use?

3. Analysis cost: (Table 5)
- Any-Address, Steensgaard extremely fast
- Anderson/Burke about 30x slower
- Choi about 2.5x slower than Anderson/Burke

<table>
<thead>
<tr>
<th>Table 5: Analysis Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>allroots</td>
</tr>
<tr>
<td>052 alvinn</td>
</tr>
<tr>
<td>01 qsort</td>
</tr>
<tr>
<td>06 matx</td>
</tr>
<tr>
<td>15 trie</td>
</tr>
<tr>
<td>04 bisect</td>
</tr>
<tr>
<td>fmxoutput</td>
</tr>
<tr>
<td>17 bintr</td>
</tr>
<tr>
<td>anagram</td>
</tr>
<tr>
<td>05 eks</td>
</tr>
<tr>
<td>08 main</td>
</tr>
<tr>
<td>09 vor</td>
</tr>
<tr>
<td>loader</td>
</tr>
<tr>
<td>129 compress</td>
</tr>
<tr>
<td>ft</td>
</tr>
<tr>
<td>football</td>
</tr>
<tr>
<td>compiler</td>
</tr>
<tr>
<td>assembler</td>
</tr>
<tr>
<td>yacc2</td>
</tr>
<tr>
<td>simulator</td>
</tr>
<tr>
<td>flex</td>
</tr>
<tr>
<td>099 go</td>
</tr>
<tr>
<td>Ratio to AT</td>
</tr>
</tbody>
</table>
Which Pointer Analysis To Use?

4. Total cost of analysis + optimizations: (Table 5)
- The client analyses improved in efficiency as the pointer information more precise
- Steensgard, Burke are 15% faster than Any-Address!
- Anderson is as fast as Any-Address!
- Choi only about 9% slower

<table>
<thead>
<tr>
<th>Pointer Analysis</th>
<th>Clients</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>AT</td>
<td>ST</td>
</tr>
<tr>
<td>allroots</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>052 alvinn</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>01 qsort</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>06 matrix</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>15 trie</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>04 bisect</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>fixoutput</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>17 bintr</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>anagram</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>ks</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>05 eks</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>08 main</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>09 vor</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>loader</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>129 compression</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>fit</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>football</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>compiler</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>assembler</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>yacc2</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>simulator</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>flex</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>099 go</td>
<td>0.73</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Ratio to AT: 1.00 0.90 29.60 32.92 79.49 1.00 0.81 0.84 0.71 0.69 1.00 0.82 0.98 0.87 1.09
# Analysis Scalability

<table>
<thead>
<tr>
<th>Context-insensitive</th>
<th>Equality-based</th>
<th>Subset-based</th>
<th>Flow-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1980: &lt; 1 KLOC</td>
<td>1994: 5 KLOC</td>
<td>1993: 30 KLOC</td>
</tr>
<tr>
<td></td>
<td>first paper on pointer analysis</td>
<td>1998: 60 KLOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Steensgaard [31]</td>
<td>• Fähndrich et al. [7]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1996: 1+ MLOC</td>
<td>1998: 60 KLOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>first scalable pointer analysis</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context-sensitive</th>
<th>Equality-based</th>
<th>Subset-based</th>
<th>Flow-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Fähndrich et al. [8]</td>
<td>• Whaley and Lam [35]</td>
<td>• Landi and Ryder [19]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cloning-based BDDs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Wilson and Lam [37]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1995: 30 KLOC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Whaley and Rinard [36]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1999: 80 KLOC</td>
</tr>
</tbody>
</table>

Derek Rayside, Points-To Analysis (Summary), 2005  

More recent: Flow-Sensitive Pointer Analysis for Millions of Lines of Code  
Hardekopf and Lin (CGO'11)
Advanced Techniques

- **Shape Analysis**: discovers and reasons about dynamically allocated data structures (e.g., lists, trees, heaps)

- **Escape Analysis**: computes which program locations can access a pointer (across function boundaries)

- **Datalog**: Declarative, constraint-based approach to specify analysis, offers pretty good scalability

  Pointer Analysis; Yannis Smaragdakis; George Balatsouras, Now Publishing, 2015
Datalog

Datalog: declarative language with Prolog-like notation

Elements: *atoms* of the form $p(X_1, X_2, \ldots X_n)$
- $p$ is a predicate
- $X_1, X_2, \ldots X_n$ are variables or constants

**Ground atoms**: predicate with only constant arguments
- Its value is either true or false

Rules: $H : - B_1 & B_2 & \ldots & B_n$
- $H$ is an *atom*, $B_1 \ldots B_n$ are *atoms* or *negations* of atoms
- $:-$ is “if” --- so $H$ is valid if all $B_1 \ldots B_n$ are valid

Datalog program is a collection of rules. The program is applied to a set of ground atoms. The result is the set of ground atoms inferred by applying the rules until fixpoint
Datalog Example

Simple Datalog program (from Dragon book):

\[
\begin{align*}
\text{path}(X,Y) & :\ - \ \text{edge} \ (X,Y) \\
\text{path}(X,Y) & :\ - \ \text{path} \ (X,Z) \ & \text{&} \ \text{path} \ (Z,Y)
\end{align*}
\]

The meaning of the program: A single edge is a path; a path also exist if there is a path between the start point and some other point, and that other point and the end point.

Consider this example:

- True ground atoms: edge(1,2), edge(2,3), edge(3,4)
- Infer path(1,2), path(2,3), path(3,4) using rule #1
- Infer composite paths using successive application of rule #2
Flow-Insensitive Pointer Analysis

(Dragonbook) Compute:

- **Pts(V, H)** – the variable V can point to heap object H
- **Hpts(H, F, G)** – field F of heap object H points to heap object G

Rules constructed by traversing the program:

1. **Pts(V, H)** : “H: V = malloc”
   
   V points to heap loc H if it is allocated at H (say we use line number calling)

   
   V points to H if V points to W and W points to H

   
   In stmt V.F=W, field F of object H points to object G if ptr W points to G and ptr V points to H

   
   In stmt V=W.F, V points to H if W points to G and field F of G points to H
Context-Sensitive Pointer Analysis

First compute:

- **Pts(V, C, H)** – the variable V in context C can point to heap object H
- **Hpts(H, F, G)** – field F of heap object H points to heap object G
- **CSinvokes(S, C, M, D)** – the calls site S in context C calls the D context of M

Rules constructed by traversing the program:


If the call site S in context C calls method M of context D, then the formal parameters in method M of context D can point to the objects pointed to by the actual params in C.