Analysis of programs with pointers

Simple example

```
x := 5     S1
ptr := &x   S2
*ptr := 9   S3
y := x     S4
```

• What are the defs and uses of x in this program?
• Problem: just looking at variable names will not give you the correct information
  – After statement S2, program names “x” and “*ptr” are both expressions that refer to the same memory location (aliases)
  – We say that ptr points-to x after statement S2.
• In a C-like language that has pointers, we must know the points-to relation to be able to determine defs and uses correctly

Program model

• For now, only types are int, int*, int** etc.
• No heap
  – All pointers point to only stack variables
• No procedure or function calls
• Statements involving pointer variables:
  – address: x := &y
  – copy: x := y
  – load: x := *y
  – store: *x := y
• Arbitrary computations involving ints

Points-to relation

• Directed graph:
  – nodes are program variables
  – edge (a,b): variable a points-to variable b

• Can use a special node to represent NULL
• Points-to relation is different at different program points
Points-to graph

- Out-degree of node may be more than one
  - if points-to graph has edges (a,b) and (a,c), it means that variable a may point to either b or c
  - depending on how we got to that point, one or the other will be true
  - path-sensitive analyses: track how you got to a program point (we will not do this)

```
if (p)
  then x := &y
  else x := &z
```

What does x point to here?

Ordering on points-to relation

- Subset ordering: for a given set of variables
  - Least element is graph with no edges
  - $G_1 \leq G_2$ if $G_2$ has all the edges $G_1$ has and maybe some more
- Given two points-to relations $G_1$ and $G_2$
  - $G_1 \cup G_2$: least graph that contains all the edges in $G_1$ and in $G_2$

Overview

- We will look at three different points-to analyses.
- Flow-sensitive points-to analysis
  - Dataflow analysis
  - Computes a different points-to relation at each point in program
- Flow-insensitive points-to analysis
  - Computes a single points-to graph for entire program
  - Andersen’s algorithm
    - Natural simplification of flow-sensitive algorithm
  - Steensgard’s algorithm
    - Nodes in tree are equivalence classes of variables
    - if x may point-to either y or z, put y and z in the same equivalence class
    - Points-to relation is a tree with edges from children to parents rather than a general graph
    - Less precise than Andersen’s algorithm but faster

Example
Notation

- Suppose \( S \) and \( S_1 \) are set-valued variables.
- \( S \leftarrow S_1 \): strong update
  - set assignment
- \( S \cup S_1 \): weak update
  - set union: this is like \( S \leftarrow S \cup S_1 \)

Flow-sensitive algorithm

Dataflow equations

- Forward flow, any path analysis
- Confluence operator: \( G_1 \cup G_2 \)
- Statements
  - \( x := &y \)
    - \( G' = G \) with \( pt(x) \leftarrow \{y\} \)
  - \( x := y \)
    - \( G' = G \) with \( pt(x) \leftarrow pt(y) \)
  - \( x := *y \)
    - \( G' = G \) with \( pt(x) \leftarrow U pt(a) \) for all \( a \in pt(y) \)
  - \( *x := y \)
    - \( G' = G \) with \( pt(a) \leftarrow pt(y) \) for all \( a \in pt(x) \)

Dataflow equations (contd.)

- \( x := &y \)
  - \( G' = G \) with \( pt(x) \leftarrow \{y\} \)
- \( x := y \)
  - \( G' = G \) with \( pt(x) \leftarrow pt(y) \)
- \( x := *y \)
  - \( G' = G \) with \( pt(x) \leftarrow U pt(a) \) for all \( a \in pt(y) \)
- \( *x := y \)
  - \( G' = G \) with \( pt(a) \leftarrow pt(y) \) for all \( a \in pt(x) \)
Strong vs. weak updates

- **Strong update:**
  - At assignment statement, you know precisely which variable is being written to.
  - Example: \( x := \ldots \).
  - You can remove points-to information about \( x \) coming into the statement in the dataflow analysis.

- **Weak update:**
  - You do not know precisely which variable is being updated; only that it is one among some set of variables.
  - Example: \( \ast x := \ldots \).
  - Problem: at analysis time, you may not know which variable \( x \) points to (see slide on control-flow and out-degree of nodes).
  - Refinement: if out-degree of \( x \) in points-to graph is 1 and \( x \) is known not be nil, we can do a strong update even for \( \ast x := \ldots \).

Structures

- **Structure types**
  - `struct cell {int value; struct cell *left, *right;}`
  - `struct cell x,y;`

- **Use a “field-sensitive” model**
  - \( x \) and \( y \) are nodes
  - each node has three internal fields labeled value, left, right

- **This representation permits pointers into fields of structures**
  - If this is not necessary, we can simply have a node for each structure and label outgoing edges with field name.

Example

```c
int main(void)
{
    struct cell {int value; struct cell *next;};
    struct cell x,y,*p;
    int sum;
    x.value = 5;
    x.next = &y;
    y.value = 6;
    y.next = &z;
    z.value = 7;
    p = &x;
    sum = 0;
    while (p != NULL) {
        sum += (p)->value;
        p = (p)->next;
    }
    return sum;
}
```

Flow-insensitive algorithms
**Flow-insensitive analysis**

- Flow-sensitive analysis computes a different graph at each program point.
- This can be quite expensive.
- One alternative: flow-insensitive analysis
  - Intuition: compute a points-to relation which is the least upper bound of all the points-to relations computed by the flow-sensitive analysis
- Approach:
  - Ignore control-flow
  - Consider all assignment statements together
    - replace strong updates in dataflow equations with weak updates
  - Compute a single points-to relation that holds regardless of the order in which assignment statements are actually executed

**Andersen’s algorithm**

- Statements

**Example**

```c
int main(void)
{ 
    struct cell {int value;
    struct cell *next;
};
    struct cell x,y,z,*p;
    int sum;
    x.value = 5;
    x.next = &y;
    y.value = 6;
    y.next = &z;
    z.value = 7;
    z.next = NULL;
    p = &x;
    sum = 0;
    while (p != NULL) {
        sum = sum + (*p).value;
        p = (*p).next;
    }
    return sum;
}
```

**Solution to flow-insensitive equations**

- Compare with points-to graphs for flow-sensitive solution
- Why does p point-to NULL in this graph?
Andersen’s algorithm formulated using set constraints

• Statements

\[ pt : \text{var} \rightarrow 2^{\text{var}} \]

\[
\begin{align*}
x & := & y & \\
y & \in & pt(x) & \\
\forall a \in pt(y), pt(x) \supseteq pt(a) & \\
x & := & *y & \\
* & x & := & y & \\
pt(x) \supseteq pt(y) & \forall a \in pt(x), pt(a) \supseteq pt(y)
\end{align*}
\]

Steensgard’s algorithm

• Flow-insensitive
• Computes a points-to graph in which there is no fan-out
  – In points-to graph produced by Andersen’s algorithm, if x points-to y and z, y and z are collapsed into an equivalence class
  – Less accurate than Andersen’s but faster
• We can exploit this to design an \( O(N^*\alpha(N)) \) algorithm, where N is the number of statements in the program.

Steensgard’s algorithm using set constraints

• Statements

\[ pt : \text{var} \rightarrow 2^{\text{var}} \]

\[
\begin{align*}
x & := & y & \\
y & \in & pt(x) & \\
\forall a \in pt(y), pt(x) = pt(a) & \\
x & := & *y & \\
* & x & := & y & \\
pt(x) = pt(y) & \forall a \in pt(x), pt(a) = pt(y)
\end{align*}
\]

Trick for one-pass processing

• Consider the following equations

\[
\begin{align*}
pt(x) & = pt(y) & dummy \in pt(x) \\
z & \in pt(x) & pt(x) = pt(y) \\
z & \in pt(x) &
\end{align*}
\]

• When first equation on left is processed, x and y are not pointing to anything.
• Once second equation is processed, we need to go back and reprocess first equation.
• Trick to avoid doing this: when processing first equation, if x and y are not pointing to anything, create a dummy node and make x and y point to that.
  – this is like solving the system on the right
• It is easy to show that this avoids the need for revisiting equations.
Algorithm

- Can be implemented in single pass through program
- Algorithm uses union-find to maintain equivalence classes (sets) of nodes
- Points-to relation is implemented as a pointer from a variable to a representative of a set
- Basic operations for union find:
  - rep(v): find the node that is the representative of the set that v is in
  - union(v1, v2): create a set containing elements in sets containing v1 and v2, and return representative of that set

Auxiliary methods

class var {
  // instance variables
  points_to: var;
  name: string;

  // constructor; also creates singleton set in union-find data structure
  var(string);

  // class method; also creates singleton set in union-find data structure
  make-dummy-var(): var;

  // instance methods
  get_pt(): var;
  set_pt(var); // updates rep
}

Initialization: make each program variable into an object of type var and enter object into union-find data structure

for each statement S in the program do
  S is x := &y: {if (pt(x) == null)
    x.set_pt(rep(y));
    else rec_union(pt(x), y);
  }

  S is x := y: {if (pt(x) == null and pt(y) == null)
    x.set_pt(var.make-dummy-var());
    y.set_pt(rec_union(pt(x), pt(y)));
  }

  S is x := *y: {if (pt(y) == null)
    y.set_pt(var.make-dummy-var());
    var a := pt(y);
    if(pt(a) == null)
      a.set_pt(var.make-dummy-var());
    x.set_pt(rec_union(pt(x), pt(a)));
  }

  S is *x := y: {if (pt(x) == null)
    x.set_pt(var.make-dummy-var());
    var a := pt(x);
    if(pt(a) == null)
      a.set_pt(var.make-dummy-var());
    y.set_pt(rec_union(pt(y), pt(a)));
  }

Inter-procedural analysis

- What do we do if there are function calls?

```plaintext
x1 = 6a
y1 = 4b
swap(x1, y1)
```

```plaintext
x2 = 6a
y2 = 4b
swap(x2, y2)
```

```
swap(p1, p2) {
  t1 = *p1;
  t2 = *p2;
  *p1 = t2;
  *p2 = t1;
}
```
Two approaches

• Context-sensitive approach:
  – treat each function call separately just like real program execution would
  – problem: what do we do for recursive functions?
    • need to approximate

• Context-insensitive approach:
  – merge information from all call sites of a particular function
  – in effect, inter-procedural analysis problem is reduced to intra-procedural analysis problem
  • Context-sensitive approach is obviously more accurate but also more expensive to compute

Context-sensitive approach

\[
\begin{align*}
x_1 &= \&a \\
y_1 &= \&b \\
\text{swap}(x_1, y_1)
\end{align*}
\]

\[
\begin{align*}
x_2 &= \&a \\
y_2 &= \&b \\
\text{swap}(x_2, y_2)
\end{align*}
\]

\[
\begin{align*}
\text{swap}(p_1, p_2) &\{ \\
t_1 &= *p_1; \\
t_2 &= *p_2; \\
*p_1 &= t_2; \\
*p_2 &= t_1; \\
\}
\end{align*}
\]

Context-insensitive/Flow-insensitive Analysis

• For now, assume we do not have function parameters
  – this means we know all the call sites for a given function

• Set up equations for binding of actual and formal parameters at each call site for that function
  – use same variables for formal parameters for all call sites

• Intuition: each invocation provides a new set of constraints to formal parameters
Swap example

\[
\begin{align*}
\text{x1} &= & \&a \\
\text{y1} &= & \&b \\
\text{p1} &= & \text{x1} \\
\text{p2} &= & \text{y1} \\
\text{t1} &= & *\text{p1} \\
\text{t2} &= & *\text{p2} \\
*\text{p1} &= & \text{t2} \\
*\text{p2} &= & \text{t1} \\
\end{align*}
\]

Heap allocation

- **Simplest solution:**
  - use one node in points-to graph to represent all heap cells
- **More elaborate solution:**
  - use a different node for each malloc site in the program
- **Even more elaborate solution: shape analysis**
  - goal: summarize potentially infinite data structures
  - but keep around enough information so we can disambiguate pointers from stack into the heap, if possible

Summary

<table>
<thead>
<tr>
<th>Less precise</th>
<th>More precise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality-based</td>
<td>Subset-based</td>
</tr>
<tr>
<td>Flow-insensitive</td>
<td>Flow-sensitive</td>
</tr>
<tr>
<td>Context-insensitive</td>
<td>Context-sensitive</td>
</tr>
</tbody>
</table>

No consensus about which technique to use
Experience: if you are context-insensitive, you might as well be flow-insensitive

History of points-to analysis