General Data Flow Frameworks

Uday Khedker
(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

September 2017

Part 1

About These Slides

Copyright

These slides constitute the lecture notes for CS618 Program Analysis course at IIT Bombay and have been made available as teaching material accompanying the book:

  (Indian edition published by Ane Books in 2013)

Apart from the above book, some slides are based on the material from the following book


*These slides are being made available under GNU FDL v1.2 or later purely for academic or research use.*

Outline

- Modelling General Flows
- Constant Propagation
- Strongly Live Variables Analysis (after mid-sem)
- Pointer Analyses (after mid-sem)
- Heap Reference Analysis (after mid-sem)
Part 2
Precise Modelling of General Flows

Complexity of Constant Propagation?

\[ a = b + 1 \]
\[ a = b + 1 \]
\[ b = c + 1 \]
\[ c = d + 1 \]
\[ d = 2 \]

Iteration #1

\[ a = b + 1 \]
\[ a = b + 1 \]
\[ b = c + 1 \]
\[ c = d + 1 \]
\[ d = 2 \]

Iteration #2

\[ a = 5 \]
\[ a = 5 \]
\[ b = 3 \]
\[ c = 3 \]
\[ d = 2 \]

Iteration #4

\[ a = 5 \]
\[ a = 5 \]
\[ b = 3 \]
\[ c = 3 \]
\[ d = 2 \]

Path Closures of Flow Functions

<table>
<thead>
<tr>
<th>Paths Terminating at ( p_2 )</th>
<th>Data Flow Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1, p_2 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( p_1, p_2, p_3, p_2 )</td>
<td>( f(x) )</td>
</tr>
<tr>
<td>( p_1, p_2, p_3, p_4, p_2 )</td>
<td>( f(f(x)) = f^2(x) )</td>
</tr>
<tr>
<td>( p_1, p_2, p_3, p_4, p_3, p_2 )</td>
<td>( f(f(f(x))) = f^3(x) )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

- For static analysis we need to summarize the value at \( p_2 \) by a value which is safe after any iteration.

\[ f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots \]

- \( f^* \) is called the loop closure of \( f \).

Loop Closure Boundedness

- Boundedness of \( f \) requires the existence of some \( k \) such that

\[ f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x) \]

- This follows from the descending chain condition

- For efficiency, we need a constant \( k \) that is independent of the size of the lattice
Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill
  \[
  f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
  f^2(x) = f(Gen \cup (x - \text{Kill})) \\
  = Gen \cup ((Gen \cup (x - \text{Kill})) - \text{Kill}) \\
  = Gen \cup ((Gen - \text{Kill}) \cup (x - \text{Kill})) \\
  = Gen \cup (Gen - \text{Kill}) \cup (x - \text{Kill}) \\
  = Gen \cup (x - \text{Kill}) = f(x) \\
  f^*(x) = x \sqcap f(x)
  \]

- Loop Closures of Bit Vector Frameworks are 2-bounded.
- Intuition: Since Gen and Kill are constant, the same things are generated or killed in every application of \(f\).
  Multiple applications of \(f\) are not required unless the input value changes.

Larger Values of Loop Closure Bounds

- Fast Frameworks \(\equiv\) 2-bounded frameworks (eg. bit vector frameworks)
  Both these conditions must be satisfied
    - Separability
      Data flow values of different entities are independent
    - Constant or Identity Flow Functions
      Flow functions for an entity are either constant or identity
- Non-fast frameworks
  At least one of the above conditions is violated

Separability

\(f : L \rightarrow L\) is \(\langle \hat{h}_1, \hat{h}_2, \ldots, \hat{h}_m \rangle\) where \(\hat{h}_i\) computes the value of \(\hat{x}_i\)

- Separable
  \[\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle\]
  \[\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle\]
  Example: All bit vector frameworks

- Non-Separable
  \[\langle \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m \rangle\]
  \[\langle \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m \rangle\]
  Example: Constant Propagation
Separability of Bit Vector Frameworks

- $\hat{L}$ is $\{0, 1\}$, $L$ is $\{0, 1\}^m$
- $\hat{\land}$ is either boolean AND or boolean OR
- $\hat{\lor}$ and $\hat{\bot}$ are 0 or 1 depending on $\hat{\land}$.
- $\hat{h}$ is a bit function and could be one of the following:

<table>
<thead>
<tr>
<th>Raise</th>
<th>Lower</th>
<th>Propagate</th>
<th>Negate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
</tr>
<tr>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
</tr>
<tr>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
<td>$\hat{\bot}$</td>
<td>$\hat{\top}$</td>
</tr>
</tbody>
</table>

Larger Values of Loop Closure Bounds

Composite flow function for the loop is

$$f((v_a, v_b, v_c, v_d)) = (v_b + 1, v_c + 1, v_d + 1, 2)$$

$f$ is not 2-bounded because:

$$f((\hat{\bot}, \hat{\bot}, \hat{\bot}, \hat{\bot})) = (\hat{\bot}, \hat{\bot}, \hat{\bot}, \hat{\bot})$$

$$f^2((\hat{\bot}, \hat{\bot}, \hat{\bot}, \hat{\bot})) = (\hat{\bot}, \hat{\bot}, \hat{\bot}, \hat{\bot})$$

Example of Constant Propagation

<table>
<thead>
<tr>
<th>MoP</th>
<th>MFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$ $a = 1$</td>
<td>$\langle \hat{\top}, \hat{\top}, \hat{\top} \rangle$</td>
</tr>
<tr>
<td>$b = 2$</td>
<td>$(1, 2, 3, \hat{\top})$</td>
</tr>
<tr>
<td>$c = a + b$</td>
<td>$(1, 2, 3, \hat{\top})$</td>
</tr>
</tbody>
</table>

$n_2$

$\langle 1, 1, 3, 3 \rangle$

$\langle 1, 1, 3, 3 \rangle$

$\langle 1, 1, 1, 1 \rangle$

$\langle 1, 1, 1, 1 \rangle$

$\langle 2, 1, 3, 2 \rangle$

$\langle 2, 1, 3, 1 \rangle$

$\langle 2, 1, 3, 2 \rangle$

$\langle 2, 1, 3, 1 \rangle$
Component Lattice for Integer Constant Propagation

Notations for Mappings as Data Flow Values

Accessing and manipulating a mapping $X \subseteq A \rightarrow B$

- $X(a)$ denotes the image of $a \in A$
  
  $X(a) \in B$

- $X[a \mapsto v]$ changes the image of $a$ in $X$ to $v$

  $X[a \mapsto v] = (X - \{(a, u) \mid u \in B\}) \cup \{(a, v)\}$

Overall Lattice for Integer Constant Propagation

- $In_n/Out_n$ values are mappings $\text{Var} \rightarrow \widehat{L}$: $In_n, Out_n \in \text{Var} \rightarrow \widehat{L}$

- Overall lattice $L$ is a set of mappings $\text{Var} \rightarrow \widehat{L}$: $L = \forall \text{Var} \rightarrow \widehat{L}$

- $\sqcap$ and $\sqcap^n$ get defined by $\sqsubseteq$ and $\widehat{\sqsubseteq}$

  ▶ Partial order is restricted to data flow values of the same variable

  Data flow values of different variables are incomparable

  $$(x, v_1) \sqsubseteq (y, v_2) \iff x = y \land v_1 \sqsubseteq v_2$$

  $\text{OR}$

  $x \mapsto v_1 \sqsubseteq y \mapsto v_2 \iff x = y \land v_1 \sqsubseteq v_2$

  ▶ For meet operation, we assume that $X$ is a total function

  Partial functions are made total by using $\widehat{\top}$ value

  $$X \sqcap Y = \{(x, v_1 \sqcap v_2) \mid (x, v_1) \in X, (x, v_2) \in Y\}$$

  $\text{OR}$

  $$X \sqcap Y = \{x \mapsto v_1 \sqcap v_2 \mid x \mapsto v_1 \in X, x \mapsto v_2 \in Y\}$$

Defining Data Flow Equations for Constant Propagation

$$In_n = \begin{cases} B_{\text{I}} = \{(y, ud) \mid y \in \text{Var}\} & n = \text{Start} \\ \prod_{p \in \text{pred}(n)} Out_p & \text{otherwise} \end{cases}$$

$$Out_n = f_n(In_n)$$

$$f_n(X) = \begin{cases} X[y \mapsto c] & n \text{ is } y = c, y \in \text{Var}, c \in \text{Const} \\ X[y \mapsto nc] & n \text{ is } \text{input}(y), y \in \text{var} \\ X[y \mapsto X(z)] & n = y = z, y \in \text{Var}, z \in \text{Var} \\ X[y \mapsto \text{eval}(e, X)] & n = y = e, y \in \text{Var}, e \in \text{Expr} \\ X & \text{otherwise} \end{cases}$$

$$\text{eval}(e, X) = \begin{cases} nc & a \in \text{Opd}(e) \cap \text{Var}, X(a) = nc \\ ud & a \in \text{Opd}(e) \cap \text{Var}, X(a) = ud \\ -X(a) & e \text{ is } -a \\ X(a) \oplus X(b) & e \text{ is } a \oplus b \end{cases}$$
Example Program for Constant Propagation

```
n_1 input (e);

n_2 a = 7; b = 2; f = e;
    if (f > 0) n_3 a = 2;
    if (f >= e + 2)

n_4 b = c + 1;
    if (b >= 7)
    n_5 if (f >= e + 1)

n_6 f = f + 1;

n_7 c = d * a;

n_8 d = a + b;

n_9 e = a + b;

n_10 d = a + 1;
    f = f + 1
```

Result of Constant Propagation

<table>
<thead>
<tr>
<th>Iteration #1</th>
<th>Changes in iteration #2</th>
<th>Changes in iteration #3</th>
<th>Changes in iteration #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>In_1</td>
<td>T, T, T, T, T</td>
<td>T, T, T, T, T</td>
<td>T, T, T, T, T</td>
</tr>
<tr>
<td>Out_1</td>
<td>T, T, T, T, T</td>
<td>T, T, T, T, T</td>
<td>T, T, T, T, T</td>
</tr>
<tr>
<td>In_2</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_2</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_3</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_3</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_4</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_4</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_5</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_5</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_6</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_6</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_7</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_7</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_8</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_8</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>In_9</td>
<td>7, 2, T, T, T, T, T, T</td>
<td>1, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
<tr>
<td>Out_9</td>
<td>2, 2, T, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
<td>1, 2, 6, 3, 1, 1, 1</td>
</tr>
</tbody>
</table>

Example Program for Constant Propagation

```
n_1 input (e);

n_2 a = 7; b = 2; f = e;
    if (f > 0) n_3 a = 2;
    if (f >= e + 2)

n_4 b = c + 1;
    if (b >= 7)
    n_5 if (f >= e + 1)

n_6 f = f + 1;

n_7 c = d * a;

n_8 d = a + b;

n_9 e = a + b;

n_10 d = a + 1;
    f = f + 1
```

For readability, we have combined many statements in a single block. However, constant propagation requires every basic block to contain a single statement because of the presence of dependent parts in flow functions.
Monotonicity of Constant Propagation

Proof obligation: \( X_1 \sqsubseteq X_2 \Rightarrow f_n(X_1) \sqsubseteq f_n(X_2) \)

where,

\[
\begin{align*}
X[y \mapsto c] & \quad n \text{ is } y = c, y \in \text{Var}, c \in \text{Const} \quad (C1) \\
X[y \mapsto nc] & \quad n \text{ is } \text{input}(y), y \in \text{var} \quad (C2) \\
X[y \mapsto X(z)] & \quad n \text{ is } y = z, z \in \text{Var} \quad (C3) \\
X[y \mapsto \text{eval}(e, X)] & \quad n \text{ is } y = e, y \in \text{Var}, e \in \text{Expr} \quad (C4) \\
X & \quad \text{otherwise} \quad (C5)
\end{align*}
\]

- The proof obligation trivially follows for cases C1, C2, C3, and C5
- For case C4, it requires showing

\[
X_1 \sqsubseteq X_2 \Rightarrow \text{eval}(e, X_1) \sqsubseteq \text{eval}(e, X_2)
\]

which follows from the definition of \( \text{eval}(e, X) \)

Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[
\begin{align*}
& a = 1 \\
& b = 2 \\
& a = 2 \\
& b = 1
\end{align*}
\]

\[
\begin{align*}
& c = a + b
\end{align*}
\]

- Correct combination.

Non-Distributivity of Constant Propagation

\[
\begin{align*}
& x = (1, 2, 3, ?) \quad \text{(Along } \text{Out}_{n_1} \rightarrow \text{In}_{n_2}) \\
& y = (2, 1, 3, 2) \quad \text{(Along } \text{Out}_{n_2} \rightarrow \text{In}_{n_2})
\end{align*}
\]

- Function application before merging

\[
\begin{align*}
f(x) \cap f(y) & = f((1, 2, 3, ?)) \cap f((2, 1, 3, 2)) \\
& = \langle 1, 2, 3, 2 \rangle \\
& = \langle \hat{1}, \hat{1}, 3, 2 \rangle
\end{align*}
\]

- Function application after merging

\[
\begin{align*}
f(x \cap y) & = f((1, 2, 3, ?) \cap (2, 1, 3, 2)) \\
& = f(\langle \hat{1}, \hat{1}, 3, 2 \rangle) \\
& = \langle \hat{1}, \hat{1}, 1, 1 \rangle
\end{align*}
\]

\[
f(x \cap y) \sqsubseteq f(x) \cap f(y)
\]

Possible combinations due to merging

\[
\begin{align*}
& a = 1 \\
& b = 2 \\
& a = 2 \\
& b = 1
\end{align*}
\]

\[
\begin{align*}
& c = a + b
\end{align*}
\]

- Correct combination.
Why is Constant Propagation Non-Distributive?

Possible combinations due to merging

\[ a = 1 \]
\[ b = 2 \]
\[ a = 2 \]
\[ b = 1 \]
\[ a = 2 \]
\[ b = 1 \]
\[ c = a + b = 2 \]

- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

Tutorial Problem on Constant Propagation

How many iterations do we need?

- Every back edge occurs only once in the ifp from \( n_3 \) to \( n_1 \) that goes via \( n_5, n_7, \) and \( n_9 \).
- \( 5 + 1 \) iterations for computing data flow values (+1 iteration to detect convergence)
And now how many iterations do we need?

Back edge $n_{10} \rightarrow n_1$ needs to be traversed once each for back edges $n_9 \rightarrow n_8$, $n_7 \rightarrow n_6$, $n_5 \rightarrow n_4$, and $n_3 \rightarrow n_2$ (in that order).

$\Rightarrow 8 + 1$ iterations.

Summary flow function:
(data flow value at node 7)

\[
f(\langle v_a, v_b, v_c \rangle) = \langle 1 \cap (v_b + 1), \ (v_c + 1), \ (v_a + 1) \rangle
\]

\[
f^0(\top) = \langle \top, \top, \top \rangle
\]

\[
f^1(\top) = \langle 1, \top, \top \rangle
\]

\[
f^2(\top) = \langle 1, \top, 2 \rangle
\]

\[
f^3(\top) = \langle 1, 3, 2 \rangle
\]

\[
f^4(\top) = \langle \bot, 3, 2 \rangle
\]

\[
f^5(\top) = \langle \bot, 3, \bot \rangle
\]

\[
f^6(\top) = \langle \bot, \bot, \bot \rangle
\]

\[
f^7(\top) = \langle \bot, \bot, \bot \rangle
\]

\[
f^*(\top) = \prod_{i=0}^{6} f^i(\top)
\]
The data flow value of every variable could change twice.

- In the worst case, only one change may happen in every step of a function application.
- Maximum number of steps: $2 \times |\text{Var}|$
- Boundedness parameter $k$ is $(2 \times |\text{Var}|) + 1$

The moral of the story:

• The boundedness parameter $k$ is $(2 \times |\text{Var}|) + 1$
• Maximum number of steps: $2 \times |\text{Var}|$
• Boundedness parameter $k$ is $(2 \times |\text{Var}|) + 1$
Conditional Constant Propagation

- The condition in block `evalCond(c)` evaluates the condition in block `c` using the data flow values in `X`

Data Flow Equations for Conditional Constant Propagation

- `In_n = \begin{cases} \langle \text{reachable}, BI \rangle & \text{if } n \text{ is } \text{Start} \\ \bigcap_{p \in \text{pred}(n)} g_p \cdot n(Out_p) & \text{otherwise} \end{cases}`

- `Out_n = \begin{cases} \langle \text{reachable}, f_n(X) \rangle & \text{if } In_n = \langle \text{reachable}, X \rangle \\ \langle \text{notReachable}, T \rangle & \text{otherwise} \end{cases}`

- `g_{m \rightarrow n}(s, X) = \begin{cases} \langle s, X \rangle & \text{if } \text{label}(m \rightarrow n) \in \text{evalCond}(m, X) \\ \langle \text{notReachable}, \top \rangle & \text{otherwise} \end{cases}`

Compile Time Evaluation of Conditions using the Data Flow Values

- `evalCond(m, X)` evaluates the condition in block `m` using the data flow values in `X`

- `\{ T, F \}`: Block `m` does not have a condition, or some variable in the condition is `⊥` in `X`

- `\{ \}`: No variable in the condition in block `m` is `⊥` in `X`, but some variable is `⊤` in `X`

- `\{ T \}`: The condition in block `m` evaluates to `T` with the data flow values in `X`

- `\{ F \}`: The condition in block `m` evaluates to `F` with the data flow values in `X`
### Part 4

#### Strongly Live Variables Analysis

- A variable is strongly live if
  - it is used in a statement other than assignment statement, or (same as simple liveness)
  - it is used in an assignment statement defining a variable that is strongly live (different from simple liveness)
- Killing: An assignment statement, an input statement, or BI
- Generation: A direct use or a use for defining values that are strongly live (this is different from how simple liveness is generated)

---

### Conditional Constant Propagation

<table>
<thead>
<tr>
<th>Iteration #1</th>
<th>Changes in Iteration #2</th>
<th>Changes in Iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_{in} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( O_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( I_{m} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( O_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( I_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( O_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( I_{in} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( O_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( I_{m} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( O_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( I_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
<tr>
<td>( O_{out} )</td>
<td>( R(0, 0, 1, 1) )</td>
<td>( R(0, 0, 1, 1) )</td>
</tr>
</tbody>
</table>

---

### Understanding Strong Liveness

<table>
<thead>
<tr>
<th>Simple Liveness</th>
<th>Strong Liveness</th>
<th>Simple Liveness</th>
<th>Strong Liveness</th>
<th>Simple Liveness</th>
<th>Strong Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ x }</td>
<td>{ x }</td>
<td>{ x }</td>
<td>{ x }</td>
<td>{ z, x }</td>
<td>{ z }</td>
</tr>
<tr>
<td>( y = x )</td>
<td>( y = x )</td>
<td>( y = x )</td>
<td>( y = x )</td>
<td>( z = y )</td>
<td>( z )</td>
</tr>
<tr>
<td>{ print (x) }</td>
<td>{ print (x) }</td>
<td>{ print (y) }</td>
<td>{ print (y) }</td>
<td>{ print (z) }</td>
<td>{ print (z) }</td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

- **Same**
- **Different**
**Live Variables Analysis: Simple and Strong Liveness**

- A variable is live at a program point if its current value is likely to be used later.
- We want to compute the smallest set of variables that are live.
- Simple liveness considers every use of a variable as useful.
- Strong liveness checks the liveness of the result before declaring the operands to be live.
- Strong liveness is more precise than simple liveness.

**Properties of Strongly Live Variable Analysis**

- What is \( \hat{L} \) for strongly live variables analysis?
  - \( \hat{L} = \{0, 1\}, 1 \subseteq 0 \)
- Is strongly live variables analysis a bit vector framework?
  - No because data flow equations cannot be defined only in terms of bit vector operations.
- Is strongly live variables analysis a separable framework?
  - No, because strong liveness of variables occurring in RHS of an assignment may depend on the variable occurring in LHS.
- Is strongly live variables analysis distributive? Monotonic?
  - Distributive, and hence monotonic.

**Data Flow Equations for Strongly Live Variables Analysis**

\[
\begin{align*}
L_n &= f_n(Out_n) \\
Out_n &= \begin{cases} 
  BI & n \text{ is End} \\
  \bigcup_{s \in \text{succ}(n)} L_s & \text{otherwise}
\end{cases}
\]

where,

\[
f_n(X) = \begin{cases} 
  \{X - \{y\}\} \cup (\text{Opd}(e) \cap \text{Var}) & n \text{ is } \text{input}(y) \\
  \{X - \{y\}\} & n \text{ is } \text{use}(y) \\
  \{X\} & \text{otherwise}
\end{cases}
\]

If \( y \) is not strongly live, the assignment is skipped using the "otherwise" clause.

**Distributivity of Strongly Live Variables Analysis (1)**

We need to prove that

\[
\forall X_1, X_2 \in L, \ f_n(X_1 \cup X_2) = f_n(X_1) \cup f_n(X_2)
\]

- Intuitively,
  - There is no dependent component \( X \)
  - Incomparable results cannot be produced (A fixed set of variable are excluded or included)
- Formally,
  - We prove it for \( \text{input}(y), \text{use}(y), y = e \), and empty statements independently.
Distributivity of Strongly Live Variables Analysis (2)

- For input statement: \( f_n(X_1 \cup X_2) = (X_1 \cup X_2) - \{y\} \)
  \( = (X_1 - \{y\}) \cup (X_2 - \{y\}) \)
  \( = f_n(X_1) \cup f_n(X_2) \)

- For use statement: \( f_n(X_1 \cup X_2) = (X_1 \cup X_2) \cup \{y\} \)
  \( = (X_1 \cup \{y\}) \cup (X_2 \cup \{y\}) \)
  \( = f_n(X_1) \cup f_n(X_2) \)

- For empty statement: \( f_n(X_1 \cup X_2) = X_1 \cup X_2 = f_n(X_1) \cup f_n(X_2) \)

Distributivity of Strongly Live Variables Analysis (3)

For \( y = e \) statement: Let \( Y = Opd(e) \cap \text{Var} \). There are three cases:

- \( y \in X_1, y \in X_2 \):
  \( f_n(X_1 \cup X_2) = ((X_1 \cup X_2) - \{y\}) \cup Y \)
  \( = (X_1 - \{y\}) \cup (X_2 - \{y\}) \cup Y \)
  \( = ((X_1 - \{y\}) \cup Y) \cup ((X_2 - \{y\}) \cup Y) \)
  \( = f_n(X_1) \cup f_n(X_2) \)

- \( y \in X_1, y \not\in X_2 \):
  \( f_n(X_1 \cup X_2) = ((X_1 \cup X_2) - \{y\}) \cup Y \)
  \( = ((X_1 - \{y\}) \cup Y) \cup (X_2) \)
  \( = f_n(X_1) \cup f_n(X_2) \)

- \( y \not\in X_1, y \not\in X_2 \):
  \( f_n(X_1 \cup X_2) = X_1 \cup X_2 = f_n(X_1) \cup f_n(X_2) \)

Result of Strongly Live Variables Analysis

<table>
<thead>
<tr>
<th>Node</th>
<th>Iteration #1</th>
<th>Iteration #2</th>
<th>Iteration #3</th>
<th>Iteration #4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Out(_n)</td>
<td>In(_n)</td>
<td>Out(_n)</td>
<td>In(_n)</td>
</tr>
<tr>
<td>(n_6)</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>(n_5)</td>
<td>{}</td>
<td>{}</td>
<td>{a}</td>
<td>{a}</td>
</tr>
<tr>
<td>(n_4)</td>
<td>{}</td>
<td>{}</td>
<td>{a}</td>
<td>{a}</td>
</tr>
<tr>
<td>(n_3)</td>
<td>{}</td>
<td>{}</td>
<td>{a}</td>
<td>{b}</td>
</tr>
<tr>
<td>(n_2)</td>
<td>{a}</td>
<td>{a, b}</td>
<td>{a, b}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>(n_1)</td>
<td>{a}</td>
<td>{a, b}</td>
<td>{b}</td>
<td>{a, c}</td>
</tr>
</tbody>
</table>
Tutorial Problem: Strongly May-Must Liveness Analysis?

- Instead of viewing liveness information as
  - a map $\forall \text{Var} \rightarrow \{0, 1\}$ with the lattice $\{0, 1\}$, view it as
  - a map $\forall \text{Var} \rightarrow \hat{L}$ where $\hat{L}$ is the May-Must Lattice
- Write the data flow equations
- Prove that the flow functions are distributive

Part 5

Pointer Analyses

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer's Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

Code Optimization In Presence of Pointers

Program

1. $q = p$;
2. while (...) { do {
3. $q = q \rightarrow \text{next}$;
4. }while (...) }
5. $p \rightarrow \text{data} = r1$;
6. print $(q \rightarrow \text{data})$;
7. $p \rightarrow \text{data} = r2$;

Memory graph at statement 5

- Is $p \rightarrow \text{data}$ live at the exit of line 5? Can we delete line 5?
  - No, if $p$ and $q$ can be possibly aliased
    (while loop or do-while loop with a circular list)
  - Yes, if $p$ and $q$ are definitely not aliased
    (do-while loop without a circular list)
Code Optimization In Presence of Pointers

\[
\begin{align*}
  a &= 5 \\
  x &= & \&a \\
  b &= & \ast x \\
  a &= 5 \\
  x &= & \&a \\
  b &= = & \ast x \\
  a &= 5 \\
  x &= & \&a \\
  b &= = & \ast x \\
  a &= 5 \\
  x &= & \&a \\
  b &= 5 \\
\end{align*}
\]

Original Program \hspace{1cm} Constant Propagation without aliasing \hspace{1cm} Constant Propagation with aliasing

The World of Pointer Analysis

 Alias Analysis \hspace{1cm} Pointer Analysis

 Alias analysis of reference parameters, fields of unions array indices \hspace{1cm} Points-to analysis of data and function pointers

Pointer Analysis Musings

- Pointer analysis collects information about indirect accesses in programs
  - Enables precise data analysis
  - Enable precise interprocedural control flow analysis
- Needs to scale to large programs
- Pointer Analysis Musings
  - Which Pointer Analysis should I Use?
    Michael Hind and Anthony Pioli. ISTAA 2000
  - Pointer Analysis: Haven’t we solved this problem yet?
    Michael Hind PASTE 2001
  - 2017 . . 😞

The Mathematics of Pointer Analysis

In the most general situation

- Alias analysis is undecidable.

- Flow insensitive alias analysis is NP-hard
  Horwitz [TOPLAS 1997]

- Points-to analysis is undecidable
  Chakravarty [POPL 2003]

Adjust your expectations suitably to avoid disappointments!
The Engineering of Pointer Analysis

So what should we expect? To quote Hind [PASTE 2001]

- “Fortunately many approximations exist”
- “Unfortunately too many approximations exist!”

*Engineering of pointer analysis is much more dominant than its science*

---

Pointer Analysis: Engineering or Science?

- Engineering view.
  - Build quick approximations
  - The tyranny of (exclusive) OR! Precision OR Efficiency?

- Science view.
  - Build clean abstractions
  - Can we harness the Genius of AND? Precision AND Efficiency?

- A distinction between approximation and abstraction is subjective
  - Our working definition
    - Abstractions focus on precision and conciseness of modelling
    - Approximations focus on efficiency and scalability

---

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer’s Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

---

Alias Information Vs. Points-to Information

```
1
a
x
b

2
a
x
b

“x Points-to a”
denoted x → a

“x and b are Aliases”
denoted x ⊤= b
```

- What about transitivity?
  - Points-to: No.
  - Alias: Depends.
Comparing Points-to and Alias Relations (1)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Memory</th>
<th>Points-to</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = &amp; y$</td>
<td>Before (assume)</td>
<td>$x \leftarrow y$</td>
<td>Existing</td>
</tr>
<tr>
<td>After</td>
<td>$x \leftarrow y$</td>
<td>New &amp; $x \leftarrow y$</td>
<td>New Direct</td>
</tr>
<tr>
<td>$x = y$</td>
<td>Before (assume)</td>
<td>$x \leftarrow y$</td>
<td>$y \leftarrow z$</td>
</tr>
<tr>
<td>After</td>
<td>$x \leftarrow y \leftarrow z$</td>
<td>New $x \leftarrow y \leftarrow z$</td>
<td>New Direct</td>
</tr>
</tbody>
</table>

• Indirect aliases. Substitute a name by its aliases for transitivity
• Derived aliases. Apply indirection operator to aliases (ignored here)

$x \leftarrow y \Rightarrow \ast x \leftarrow \ast y$

Comparing Points-to and Alias Relations (2)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Memory</th>
<th>Points-to</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ast x = y$</td>
<td>Before (assume)</td>
<td>$\ast x \leftarrow u$</td>
<td>$\ast y \leftarrow z$</td>
</tr>
<tr>
<td>After</td>
<td>$\ast x \leftarrow u \leftarrow z$</td>
<td>New $\ast x \leftarrow u \leftarrow z$</td>
<td>New Direct</td>
</tr>
<tr>
<td>$x = \ast y$</td>
<td>Before (assume)</td>
<td>$x \leftarrow u$</td>
<td>$z \leftarrow u$</td>
</tr>
<tr>
<td>After</td>
<td>$x \leftarrow u \leftarrow z$</td>
<td>New $x \leftarrow u \leftarrow z$</td>
<td>New Indirect</td>
</tr>
</tbody>
</table>

The resulting memories look similar but are different. In the first case we have $u \leftarrow z$ whereas in the second case the arrow direction is opposite (i.e. $z \leftarrow u$).

Comparing Points-to and Alias Relations (3)

• Points-to information records edges in the memory graph
  ▶ aliases of the kind $x \leftarrow \& y$
  $x$ holds the address of $y$
  ▶ other aliases can be discovered by composing edges
  ▶ since addresses are explicated, it can represent only those memory locations that can be named at compile time

More compact but less general
• Alias information records paths in the memory graph
  ▶ paths incident on the same node
  $x$ and $y$ hold the same address (and the address is left implicit)
  ▶ since addresses are implicit, it can represent unnamed memory locations too
  ▶ if we have $x \leftarrow y$ then $\ast x \leftarrow \ast y$ is redundant and is not recorded

More general and more complex

An Outline of Pointer Analysis Coverage

• The larger perspective
• Comparing Points-to and Alias information
  • Flow Insensitive Points-to Analysis · Next Topic
• Flow Sensitive Points-to Analysis
• Pointer Analyses: An Engineer’s Landscape
• Liveness Based Points-to Analysis
• Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions
Flow Sensitive Vs. Flow Insensitive Pointer Analysis

- Flow insensitive pointer analysis
  - Inclusion based: Andersen’s approach
  - Equality based: Steensgaard’s approach
- Flow sensitive pointer analysis
  - May points-to analysis
  - Must points-to analysis

Flow Insensitivity in Data Flow Analysis

- Assumption: Statements can be executed in any order.
- Instead of computing point-specific data flow information, summary data flow information is computed. The summary information is required to be a safe approximation of point-specific information for each point.
- Kill_\text{n}(X) component is ignored.
  If statement \text{n} kills data flow information, there is an alternate path that excludes \text{n}.

Examples of Flow Insensitive Analyses

- Type checking/inferencing
  (What about interpreted languages?)
- Address taken analysis
  Which variables have their addresses taken?
- Side effects analysis
  Does a procedure modify a global variable? Reference Parameter?
Flow Insensitivity in Data Flow Analysis

Assuming $\text{Gen}_n(X)$ has dependent parts and $\text{Kill}_n(X)$ is ignored

Allows arbitrary compositions of flow functions in any order

⇒ Flow insensitivity

In practice, dependent constraints are collected in a global repository in one pass and then are solved independently

Notation for Andersen’s and Steensgaard’s Points-to Analysis

- $P_x$ denotes the set of pointees of pointer variable $x$
- $\text{Unify}(x, y)$ unifies locations $x$ and $y$
  - $x$ and $y$ are treated as equivalent locations
  - the pointees of the unified locations are also unified transitively
- $\text{UnifyPTS}(x, y)$ unifies the pointees of $x$ and $y$
  - $x$ and $y$ themselves are not unified

Andersen’s and Steensgaard’s Points-to Analysis

<table>
<thead>
<tr>
<th>Statement</th>
<th>Andersen’s Points-to Sets</th>
<th>Steensgaard’s Points-to Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y$</td>
<td>$P_x \supset P_y$</td>
<td>$P_x \supset {y}$ for some $z \in P_x$</td>
</tr>
<tr>
<td>$x = &amp; y$</td>
<td>$P_x \supset {y}$</td>
<td>$\text{UnifyPTS}(y, z)$ for some $z \in P_x$</td>
</tr>
<tr>
<td>$x = \ast y$</td>
<td>$P_x \supset P_y, \forall z \in P_y$</td>
<td>$\forall z \in P_y, \text{UnifyPTS}(x, z)$</td>
</tr>
<tr>
<td>$\ast x = y$</td>
<td>$P_z \supset P_y, \forall z \in P_x$</td>
<td>$\forall z \in P_x, \text{UnifyPTS}(y, z)$</td>
</tr>
</tbody>
</table>

Andersen’s view
- $x$ points to $y$
- Include $y$ in the points-to set of $x$

Steensgaard’s view
- Equivalence between: Pointees of $x$
- Unify $y$ and pointees of $x$
### Andersen's and Steensgaard's Points-to Analysis

<table>
<thead>
<tr>
<th>Statement</th>
<th>Andersen's Points-to Sets</th>
<th>Steensgaard's Points-to Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = &amp; y$</td>
<td>$P_x \supseteq {y}$</td>
<td>$P_x \supseteq {y}$ for some $z \in P_x$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$P_x \supseteq P_y$</td>
<td>$\text{UnifyPTS}(x, y)$</td>
</tr>
<tr>
<td>$x = &amp; y$</td>
<td>$P_x \supseteq P_z, \ \forall z \in P_y$</td>
<td>$\forall z \in P_y, \ \text{UnifyPTS}(x, z)$</td>
</tr>
<tr>
<td>$*x = y$</td>
<td>$P_z \supseteq P_y, \ \forall z \in P_x$</td>
<td>$\forall z \in P_x, \ \text{UnifyPTS}(y, z)$</td>
</tr>
</tbody>
</table>

**Andersen's view**
- $x$ points to pointees of $y$
- Include the pointees of $y$ in the points-to set of $x$

**Steensgaard's view**
- Equivalence between: Pointees of $x$ and pointees of $y$
- Unify points-to sets of $x$ and $y$

<table>
<thead>
<tr>
<th>Statement</th>
<th>Andersen's Points-to Sets</th>
<th>Steensgaard's Points-to Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = &amp; y$</td>
<td>$P_x \supseteq {y}$</td>
<td>$P_x \supseteq {y}$ for some $z \in P_x$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$P_x \supseteq P_y$</td>
<td>$\text{UnifyPTS}(x, y)$</td>
</tr>
<tr>
<td>$x = &amp; y$</td>
<td>$P_x \supseteq P_z, \ \forall z \in P_y$</td>
<td>$\forall z \in P_y, \ \text{UnifyPTS}(x, z)$</td>
</tr>
<tr>
<td>$*x = y$</td>
<td>$P_z \supseteq P_y, \ \forall z \in P_x$</td>
<td>$\forall z \in P_x, \ \text{UnifyPTS}(y, z)$</td>
</tr>
</tbody>
</table>

**Andersen's view**
- $x$ points to pointees of pointees of $y$
- Include the pointees of pointees of $y$ in the points-to set of $x$

**Steensgaard's view**
- Equivalence between: Pointees of $x$ and pointees of pointees of $y$
- Unify points-to sets of $x$ and pointees of $y$
Inclusion Based (aka Andersen's) Points-to Analysis: Example 1

Program

Node | Constraint
---- | -------
1    | $P_a \supset \{b\}$
2    | $P_c \supset P_a$
3    | $P_a \supset \{d\}$
4    | $P_a \supset \{e\}$
5    | $P_b \supset P_a$

Points-to Graph

- Since $P_a$ has changed, $P_c$ needs to be processed again

Equality Based (aka Steensgaard's) Points-to Analysis: Example 1

Program

Node | Constraint
---- | -------
1    | $P_a \supset \{b\}$
2    | $P_c \supset P_a$
3    | $P_a \supset \{d\}$
4    | $P_a \supset \{e\}$
5    | $P_b \supset P_a$

Points-to Graph

- Observe that $P_c$ is processed for the third time
- Order of processing the sets influences efficiency significantly
- A plethora of heuristics have been proposed

Actually:
- c does not point to any location in block 1
- a does not point b in block 5
  (the method ignores the kill due to 3 and 4)
- b does not point to itself at any time
Equality Based (aka Steensgaard’s) Points-to Analysis: Example 1

Program

<table>
<thead>
<tr>
<th>Node</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P_a \supseteq { b } )</td>
</tr>
<tr>
<td>2</td>
<td>UnifyPTS(c, a)</td>
</tr>
<tr>
<td>3</td>
<td>( P_d \supseteq { d } )</td>
</tr>
<tr>
<td>5</td>
<td>UnifyPTS(b, a)</td>
</tr>
</tbody>
</table>

Points-to Graph

- The full blown up points-to graph has far more edges than in the graph created by Andersen’s method
- Far more efficient but far less precise

Comparing Equality and Inclusion Based Analyses (2)

- Andersen’s algorithm is cubic in number of pointers
- Steensgaard’s algorithm is nearly linear in number of pointers
  ▶ How can it be more efficient by an orders of magnitude?

Efficiency of Equality Based Approach

<table>
<thead>
<tr>
<th>Program</th>
<th>Andersen’s approach</th>
<th>Steensgaard’s approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = &amp;b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = &amp;c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = &amp;d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = &amp;c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Andersen’s inclusion based wisdom:
  ▶ Add edges and let the number of successors increase
- Steensgaard’s equality based wisdom:
  ▶ Merge multiple successors and maintain a single successor of any node
  ▶ Since a larger number of pointers treated are alike and fewer distinctions are maintained, we get much smaller points-to graphs
  ▶ Efficient Union-Find algorithms to merge intersecting subsets

Inclusion Based (aka Andersen’s) Points-to Analysis: Example 2

Constraints on Points-to Sets

\[
\begin{align*}
& P_x \supseteq \{ y \} \\
& P_y \supseteq \{ z \} \\
& P_z \supseteq \{ u \} \\
& \forall w \in P_z, \; P_w \supseteq P_y \\
& P_z \supseteq P_y \\
& P_y \supseteq \{ x \}
\end{align*}
\]
Equality Based (aka Steensgaard’s) Points-to Analysis: Example 2

- Treat all pointees of a pointer as “equivalent” locations
- Transitive closure: Pointees of all equivalent locations become equivalent

### Steensgaard’s Points-to Graph

- Example 2
  - \( x = \& y \)
  - \( y = \& z \)
  - \( z = \& u \)
  - \( \# z = y \) \( n_2 \)
  - \( z = y \) \( n_3 \)
  - \( y = \& x \) use \( u \)
  - use \( x \) \( n_4 \)

Effective additional constraints:

- Unify \( x, y \) /* pointees of \( x \) */
- Unify \( x, z \) /* pointees of \( y \) */
- Unify \( x, u \) /* pointees of \( z \) */

\[ \Rightarrow x, y, z, u \text{ are equivalent} \]
\[ \Rightarrow \text{Complete graph} \]

Tutorial Problem for Flow Insensitive Pointer Analysis (1)

Program

<table>
<thead>
<tr>
<th>Inclusion based</th>
<th>Equality based</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = &amp; q )</td>
<td>( u )</td>
</tr>
<tr>
<td>( r = &amp; s )</td>
<td>( q )</td>
</tr>
<tr>
<td>( t = &amp; p )</td>
<td>( p )</td>
</tr>
<tr>
<td>( u = p )</td>
<td>( s )</td>
</tr>
<tr>
<td>( \ast t = r )</td>
<td>( u )</td>
</tr>
</tbody>
</table>

Tutorial Problem for Flow Insensitive Pointer Analysis (2)

Compute flow insensitive points-to information using inclusion based method as well as equality based method

```
if (...) 
    p = \& x;
else 
    p = \& y;

x = \& a;
y = \& b;
*p = \& c;
*y = \& a;
```

Tutorial Problem for Flow Insensitive Pointer Analysis (3)

Compute flow insensitive points-to information using inclusion based method as well as equality based method

```
if (...) 
    b = \& a;
```

```
c = b;
```

```
a = \& b;
y = \& b;
*p = \& c;
*y = \& a;
```

```
a = \& a;
```

```
\ast b = c;
```
An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer’s Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

Must Points-to Information

\[
\begin{align*}
1 & \quad x = & a \\
2 & \quad x = & b \\
3 & \quad y = & b \\
4 & \end{align*}
\]

Must Alias Information

\[
\begin{align*}
1 & \quad x = & a \\
2 & \quad b = & x \\
3 & \quad y = & b \\
4 & \end{align*}
\]

\[x \equiv b \text{ and } b \equiv y \Rightarrow x \equiv y\]
May Alias Information

\[ x = \& a \]
\[ b = \& z \]
\[ b = x \]
\[ y = b \]
\[ z = \& x \]
\[ z = \& y \]

\( x \equiv b \) and \( b \equiv y \) \( \not\equiv x \equiv y \)

Strong and Weak Updates

Weak update: Modification of \( x \) or \( y \) due to \( *z \) in block 5

Strong update: Modification of \( c \) due to \( *w \) in block 5

How is this concept related to May/Must nature of information?

What About Heap Data?

- Compile time entities, abstract entities, or summarized entities
- Three options:
  - Represent all heap locations by a single abstract heap location
  - Represent all heap locations of a particular type by a single abstract heap location
  - Represent all heap locations allocated at a given memory allocation site by a single abstract heap location
- Summarization: Usually based on the length of pointer expression
- Initially, we will restrict ourselves to stack and static data
  We will later introduce heap using the allocation site based abstraction

Lattice for May Points-to Analysis

Let \( P \subseteq \text{Var} \) be the set of pointers. Assume \( \text{Var} = \{p, q\} \) and \( P = \{p\} \)

Product View

| \emptyset | \{p, \} |
| \{p, p\} | \{p, q\} |
| \{p, \{p\}\} | \{p, \{q\}\} |
| \{p, \{p, q\}\} |

Mapping view

| \{p, \{p\}\} |
| \{p, \{q\}\} |

Data flow values \( \subseteq P \times \text{Var} \)

Lattice = \( 2P \times \text{Var}, \subseteq \)

Points-to graph as a list of directed edges

Data flow values = \( P \rightarrow 2\text{Var} \)

Lattice = \( 2P \rightarrow 2\text{Var}, \subseteq_{\text{map}} \)

Points-to graph as a list of adjacency lists
**Lattice for Must Points-to Analysis**

Let $P \subseteq \mathbb{Var}$ be the set of pointers. Assume $\mathbb{Var} = \{p, q, r\}$ and $P = \{p\}$

**Mapping View**

Set View

$$\{(p, \top)\}$$

$$\{(p, p)\} \quad \{(p, q)\} \quad \{(p, r)\}$$

$$\{(p, \bot)\}$$

Data flow values $= P \rightarrow \mathbb{Var} \cup \{\top, \bot\}$

Lattice $= (2^P \rightarrow \mathbb{Var} \cup \{\top, \bot\}, \sqsubseteq_{map})$

Restricted subset of $P \times \mathbb{Var}$

∩ can be used for $\sqcap$

A pointer can point to at most one location

**Lattice for Combined May-Must Points-to Analysis (1)**

- Consider the following abbreviation of the May-Must lattice $\hat{L}$

  \[
  \begin{array}{c}
  \text{Unknown} \\
  \text{No} \quad \text{Must} \\
  \text{May}
  \end{array}
  \]

  abbreviated as

  \[
  \begin{array}{c}
  \text{un} \\
  \text{no} \quad \text{mt}
  \end{array}
  \]

- For $\mathbb{Var} = \{p, q\}$, $P = \{p\}$, the May-Must points-to lattice is the product $P \times \mathbb{Var} \times \hat{L}$

  - Some elements are prohibited because of the semantics of **Must**
  
    - If we have $(p, p, mt) \in P \times \mathbb{Var} \times \hat{L}$, then
      
        - we cannot have $(p, q, un)$, $(p, q, mt)$, or $(p, q, my)$ in $P \times \mathbb{Var} \times \hat{L}$
        
        - we can only have $(p, q, no)$ in $P \times \mathbb{Var} \times \hat{L}$

**Lattice for Combined May-Must Points-to Analysis (2)**

For $\mathbb{Var} = \{p, q\}$, $P = \{p\}$, the May-Must points-to lattice is

**Prohibited**

**Allowed**
Lattice for Combined May-Must Points-to Analysis (2)

For $\forall \text{Var} = \{p, q\}$, $\mathbf{P} = \{p\}$, the May-Must points-to lattice is:

$\{(p.p.un), (p.q.un)\}$

$\{(p.p,un), (p.q.un)\}$

$\{(p.p,un), (p.q.no)\}$

$\{(p.p,no), (p.q.un)\}$

$\{(p.p,no), (p.q.no)\}$

$\{(p.p.my), (p.q.my)\}$

$\{(p.p.my), (p.q.no)\}$

$\{(p.p.my), (p.q.no)\}$

$\{(p.p.my), (p.q.my)\}$

$\{(p.p.no), (p.q.no)\}$

$\{(p.p.no), (p.q.un)\}$

$\{(p.p.no), (p.q.un)\}$

$\{(p.p.no), (p.q.my)\}$

$\{(p.p.no), (p.q.my)\}$

$\{(p.p,no), (p.q.no)\}$

$\{(p.p,no), (p.q.no)\}$

$\{(p.p.no), (p.q.my)\}$

$\{(p.p.no), (p.q.my)\}$

May and Must Analysis for Killing Points-to Information (1)

**May Points-to Analysis**
- $(a, b)$ should be in $\text{MayIn}_4$
  - Holds along path 1-3-4
- Block 4 should not kill $(a, b)$
- Possible if pointee set of $c$ is $\emptyset$
- However, $\text{MayIn}_4$ contains $(c, a)$

For killing points-to information through indirection,
- **Must** points-to analysis should identify pointees of $c$ using $\text{MayIn}_4$
- **May** points-to analysis should identify pointees of $c$ using $\text{MustIn}_4$

May and Must Analysis for Killing Points-to Information (2)

**May Points-to Analysis**
- $(a, b)$ should not be in $\text{MustIn}_4$
  - Does not hold along path 1-2-4
- Block 4 should kill $(a, b)$
- Possible if pointee set of $c$ is $\{a\}$
- However, $\text{MustIn}_4$ contains $\{(a, b)\}$

**Must Points-to Analysis**
- $(a, b)$ should remove a May points-to pair
  - only if it must be removed along all paths
- Kill should remove only strong updates
  - should use Must Points-to information
- $(a, b)$ should remove a Must points-to pair
  - if it can be removed along any path
- Kill should remove all weak updates
  - should use May Points-to information
Discovering Must Points-to Information from May Points-to Information

- **Bl.** every pointer points to "?"
- Perform usual may points-to analysis
- Since c has multiple pointees, it is a MAY relation
- Since a has a single pointee, it is a MUST relation

### Relevant Algebraic Operations on Relations (1)

- Let \( P \subseteq \text{Var} \) be the set of pointer variables
- May-points-to information: \( A = (2^{P \times \text{Var}}, \supseteq) \)
- Standard algebraic operations on points-to relations
  - **Relation application** \( R \times X = \{ v \mid u \in X \land (u, v) \in R \} \)
    (Find out the pointees of the pointers contained in \( X \))
  - **Relation restriction** \( R|_X = \{ (u, v) \in R \mid u \in X \} \)
    (Restrict the relation only to the pointers contained in \( X \) by removing points-to information of other pointers)

### Relevant Algebraic Operations on Relations (2)

Let
\[
\begin{align*}
\text{Var} &= \{ a, b, c, d, e, f, g, ? \} \\
P &= \{ a, b, c, d, e \} \\
R &= \{(a, b), (a, c), (b, d), (c, e), (c, g), (d, a), (e, ?)\} \\
X &= \{a, c\}
\end{align*}
\]

Then,
\[
\begin{align*}
R \times X &= \{ v \mid u \in X \land (u, v) \in R \} \\
&= \{b, c, e, g\} \\
R|_X &= \{ (u, v) \in R \mid u \in X \} \\
&= \{ (a, b), (a, c), (c, e), (c, g) \}
\end{align*}
\]

### Points-to Analysis Data Flow Equations

- \( \text{Ain}_n = \begin{cases} 
\text{Var} \times \{?\} & n \text{ is } \text{Start}_p \\
\bigcup_{p \in \text{pred}(n)} \text{Aout}_p & \text{otherwise}
\end{cases} \)
- \( \text{Aout}_n = (\text{Ain}_n - (\text{Kill}_n \times \text{Var})) \cup (\text{Def}_n \times \text{Pointee}_n) \)

- \( \text{Ain}/\text{Aout} \): sets of mAy points-to pairs
- \( \text{Kill}_n, \text{Def}_n, \) and \( \text{Pointee}_n \) are defined in terms of \( \text{Ain}_n \)
Points-to Analysis Data Flow Equations

\[ \text{Ain}_n = \begin{cases} \forall \text{Var} \times \{?\} & \text{n is Start}_p \\ \bigcup_{p \in \text{pred}(n)} \text{Aout}_p & \text{otherwise} \end{cases} \]

\[ \text{Aout}_n = (\text{Ain}_n - (\text{Kill}_n \times \text{Var})) \cup (\text{Def}_n \times \text{Pointee}_n) \]

- \text{Ain/Aout}: sets of mAy points-to pairs
- \text{Kill}_n, \text{Def}_n, \text{and Pointee}_n are defined in terms of \text{Ain}_n

Extractor Functions for Points-to Analysis

Values defined in terms of \text{Ain}_n (denoted A)

<table>
<thead>
<tr>
<th></th>
<th>Def(_n)</th>
<th>Kill(_n)</th>
<th>Pointee(_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>use x</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>x = &amp;a</td>
<td>{x}</td>
<td>{x}</td>
<td>{a}</td>
</tr>
<tr>
<td>x = y</td>
<td>{x}</td>
<td>{x}</td>
<td>A{y}</td>
</tr>
<tr>
<td>x = *y</td>
<td>{x}</td>
<td>{x}</td>
<td>A({y} \cap P)</td>
</tr>
<tr>
<td>*x = y</td>
<td>A{x} \cap P</td>
<td>Must(A){x} \cap P</td>
<td>A{y}</td>
</tr>
<tr>
<td>other</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Pointees of y in \text{Ain}_n are the targets of defined pointers

Pointees of those pointees of y in \text{Ain}_n which are pointers
## Extractor Functions for Points-to Analysis

Values defined in terms of $A_{in}$ (denoted $A$)

<table>
<thead>
<tr>
<th></th>
<th>$Def_{in}$</th>
<th>$Kill_{in}$</th>
<th>$Pointee_{in}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$use x$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x = &amp;a$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>$x = *y$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>$A({y} \cap P)$</td>
</tr>
<tr>
<td>$*x = y$</td>
<td>$A({x} \cap P)$</td>
<td>$Must(A){x} \cap P$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Pointees of $x$ in $A_{in}$ receive new addresses.

---

Strong update using must-points-to information computed from $A_{in}$

<table>
<thead>
<tr>
<th></th>
<th>$Def_{in}$</th>
<th>$Kill_{in}$</th>
<th>$Pointee_{in}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$use x$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x = &amp;a$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>$x = *y$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>$A({y} \cap P)$</td>
</tr>
<tr>
<td>$*x = y$</td>
<td>$A({x} \cap P)$</td>
<td>$Must(A){x} \cap P$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Must($R$) = $\bigcup_{z \in P} \{w\} \times \{w\}$ \text{ if } R(z) = \{w\} \wedge w \neq ?  
otherwise

$z$ has a single pointee $w$ in must-points-to relation.

---

Strong update using must-points-to information computed from $A_{in}$

<table>
<thead>
<tr>
<th></th>
<th>$Def_{in}$</th>
<th>$Kill_{in}$</th>
<th>$Pointee_{in}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$use x$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x = &amp;a$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>$x = *y$</td>
<td>${x}$</td>
<td>${x}$</td>
<td>$A({y} \cap P)$</td>
</tr>
<tr>
<td>$*x = y$</td>
<td>$A({x} \cap P)$</td>
<td>$Must(A){x} \cap P$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Must($R$) = $\bigcup_{z \in P} \{w\} \times \{w\}$ \text{ if } R(z) = \{w\} \wedge w \neq ?  
otherwise

$z$ has no pointee in must-points-to relation.
Extractor Functions for Points-to Analysis

Values defined in terms of $Ain_n$ (denoted $A$)

<table>
<thead>
<tr>
<th>Def$_n$</th>
<th>Kill$_n$</th>
<th>Pointee$_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{use } x$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x = &amp;a$</td>
<td>${x}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>${x}$</td>
<td>$A(y)$</td>
</tr>
<tr>
<td>$x = \ast y$</td>
<td>${x}$</td>
<td>$A({y} \cap P)$</td>
</tr>
<tr>
<td>$\ast x = y$</td>
<td>$A{x} \cap P$</td>
<td>$\text{Must}(A){x} \cap P$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

$\text{Must}(R) = \bigcup_{p} \{z\} \times \left\{ \begin{array}{ll} \{w\} & w \neq \text{?} \\ \emptyset & \text{otherwise} \end{array} \right.$

Pointees of $y$ in $Ain_n$ are the targets of defined pointers

An Example of Flow Sensitive May Points-to Analysis

Assume that the program is type correct

Weak Update $n_4$ $\ast u = \& x$

Strong Update $n_5$ $\ast y = \& y$

Non-Distributivity of Points-to Analysis

May Points-to

Must Points-to

$z \rightarrow w$ is spurious

$a \rightarrow d$ is missing

Sep 2017 IIT Bombay
An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer’s Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

An Example of Flow Insensitive May Points-to Analysis

For simplicity, we ignore the BI with “?”

An Example of Flow Sensitive May Points-to Analysis

Context Sensitivity in Interprocedural Analysis
We will revisit this concept and study it in details in the fourth module (interprocedural data flow analysis) of the course.

Context Sensitivity in Interprocedural Analysis

- Paths from Start to End should constitute a context free language $c^*sr^n$
- Many interprocedural analyses treat cycle of recursion as an SCC and approximate paths by a regular language $c^*sr^*$
- We do not know any practical points-to analysis that is fully context sensitive

Most context sensitive approaches
- either do not consider recursion, or
- do not consider recursive pointer manipulation (e.g. $p = p \rightarrow n$), or
- are context insensitive in recursion

Context Sensitivity in the Presence of Recursion

- Paths from Start to End should constitute a context free language $c^*sr^n$
- Many interprocedural analyses treat cycle of recursion as an SCC and approximate paths by a regular language $c^*sr^*$

We will revisit this concept and study it in details in the fourth module (interprocedural data flow analysis) of the course.
**Pointer Analysis: An Engineer’s Landscape**

- Flow Sensitivity Increases
- Context Sensitivity Increases
- \( FI = FI \subseteq FI_{SSA} \)
- \( FS = FS_{NoKill} \)
- \( CL, CI, CI_{ObjSens}, CS_{Reclns}, CS \)
- Over Crowed Area
- Still Vacant

**An Outline of Pointer Analysis Coverage**

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer’s Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions
Is All This Information Useful?

For simplicity, we ignore the BI with "?"

\[
\begin{align*}
 x &= &y \\
 y &= &z \\
 z &= &u \\
 \emptyset \\
 x \rightarrow &y \rightarrow &z \rightarrow &u \\
 x \rightarrow &y \rightarrow &z \rightarrow &u \\
 z &= &y \\
 z &= &y \\
 n_1 &\rightarrow &n_2 &\rightarrow &n_3 &\rightarrow &n_4 &\rightarrow &n_4 \\
 y &= &x &use &u &use &x \\
\end{align*}
\]

The L and P of LFCPA

Mutual dependence of liveness and points-to information
- Define points-to information only for live pointers
- For pointer indirections, define liveness information using points-to information

The F and C of LFCPA

- Use call strings method for full flow and context sensitivity
- Use value contexts for efficient interprocedural analysis

Use of Strong Liveness

- Simple liveness considers every use of a variable as useful
- Strong liveness checks the liveness of the result before declaring the operands to be live
- Strong liveness is more precise than simple liveness
### Extractor Functions for LFCPA

<table>
<thead>
<tr>
<th>Def(n)</th>
<th>Kill(n)</th>
<th>Pointee(n)</th>
<th>Ref(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>use (x)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>({x})</td>
</tr>
<tr>
<td>(x = &amp;a)</td>
<td>({x})</td>
<td>({a})</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(x = y)</td>
<td>({x})</td>
<td>(A(y))</td>
<td>({y})</td>
</tr>
<tr>
<td>(x = *y)</td>
<td>({x})</td>
<td>(A(A(y) \cap P) \cup A(y) \cap P)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\ast x = y)</td>
<td>(A(x) \cap P)</td>
<td>(\text{Must}(A(x) \cap P))</td>
<td>({x, y})</td>
</tr>
<tr>
<td>other</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

- Lin/Lout: set of Live pointers, Ain/Aout: sets of mAy points-to pairs
- Ref\(n\), Kill\(n\), Def\(n\), and Pointee\(n\) are defined in terms of Ain\(n\)

### Unchanged from earlier points-to analysis

- Defined pointers must be live at the exit for the read pointers to become live
- Some pointers are unconditionally live

### Generation of strong liveness

- Points that become live
- Defined pointers must be live at the exit for the read pointers to become live
- Some pointers are unconditionally live

---

### Extractor Functions for LFCPA

<table>
<thead>
<tr>
<th>Def(n)</th>
<th>Kill(n)</th>
<th>Pointee(n)</th>
<th>Ref(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>use (x)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>({x})</td>
</tr>
<tr>
<td>(x = &amp;a)</td>
<td>({x})</td>
<td>({a})</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(x = y)</td>
<td>({x})</td>
<td>(A(y))</td>
<td>({y})</td>
</tr>
<tr>
<td>(x = *y)</td>
<td>({x})</td>
<td>(A(A(y) \cap P) \cup A(y) \cap P)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\ast x = y)</td>
<td>(A(x) \cap P)</td>
<td>(\text{Must}(A(x) \cap P))</td>
<td>({x, y})</td>
</tr>
<tr>
<td>other</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

- \(x\) is unconditionally live
- \(y\) is live if defined pointers are live
## Extractor Functions for LFPCA

### Unchanged from earlier points-to analysis

<table>
<thead>
<tr>
<th>Def&lt;sub&gt;n&lt;/sub&gt;</th>
<th>Kill&lt;sub&gt;n&lt;/sub&gt;</th>
<th>Pointee&lt;sub&gt;n&lt;/sub&gt;</th>
<th>Ref&lt;sub&gt;n&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>use x</td>
<td>∅</td>
<td>∅</td>
<td>{x}</td>
</tr>
<tr>
<td>x = &amp;a</td>
<td>{x}</td>
<td>{a}</td>
<td>∅</td>
</tr>
<tr>
<td>x = y</td>
<td>{x}</td>
<td>A(y)</td>
<td>{y}</td>
</tr>
<tr>
<td>x = *y</td>
<td>{x}</td>
<td>A(A(y) ∪ P)</td>
<td>∅</td>
</tr>
<tr>
<td>*x = y A(x) ∩ P</td>
<td>Must(A(x) ∩ P)</td>
<td>A(y)</td>
<td>{x}</td>
</tr>
<tr>
<td>other</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

### Generation of strong liveness

- y is live if defined pointers are live
- x is unconditionally live

---

## Deriving Must Points-to for LFPCA

For *x = y*, unless the pointees of x are known:

- points-to propagation should be blocked
- liveness propagation should be blocked

To ensure monotonicity:

\[
\text{Must}(R) = \bigcup_{x \in P} \begin{cases} \forall \text{var} & R\{x\} = \emptyset \vee R\{x\} = \{?\} \\ \{y\} & R\{x\} = \{y\} \land y \neq ? \\ \emptyset & \text{otherwise} \end{cases}
\]

---

## LFPCA Data Flow Equations

1. **Lout<sub>n</sub>**
   
   \[
   \text{Lout}_n = \begin{cases} \emptyset & n = \text{End}_p \\ \bigcup_{s \in \text{succ}(n)} \text{Lin}_s & \text{otherwise} \end{cases}
   \]

2. **Lin<sub>n</sub>**
   
   \[
   \text{Lin}_n = (\text{Lout}_n - \text{Kill}_n) \cup \text{Ref}_n
   \]

3. **Ain<sub>n</sub>**
   
   \[
   \text{Ain}_n = \begin{cases} \bigcup_{p \in \text{pred}(n)} \text{Aout}_p \big| \text{Lin}_n & n = \text{Start}_p \\ \text{Lin}_n \times \{?\} & \text{otherwise} \end{cases}
   \]

4. **Aout<sub>n</sub>**
   
   \[
   \text{Aout}_n = (\text{Ain}_n - (\text{Kill}_n \times \forall \text{var})) \cup (\text{Def}_n \times \text{Pointee}_n) \big| \text{Lout}_n
   \]

- Lin/Lout: set of Live pointers
- Ain/Aout: definitions remain unchanged except for restriction to liveness
Ref
For convenience, we show complete sweeps of liveness and points-to
This is not required by the computation
The data flow equations define a single fixed point computation

LFCPA Data Flow Equations

\[ L_{out} = \begin{cases} \emptyset & \text{n is } End_p \\ \bigcup_{s \in \text{succ}(n)} L_{in_s} & \text{otherwise} \end{cases} \]

\[ L_{in} = \left( L_{out} - \text{Kill}_n \right) \cup \text{Ref}_n \]

\[ A_{in} = \begin{cases} L_{in} \times \{?\} & \text{n is } Start_p \\ \left( \bigcup_{p \in \text{pred}(n)} A_{out_p} \right) \big| L_{in} & \text{otherwise} \end{cases} \]

\[ A_{out} = \left( \left( A_{in} - \text{Kill}_n \times \text{Var} \right) \cup \left( \text{Def}_n \times \text{Pointee}_n \right) \right) \big| L_{out} \]

Motivating Example Revisited

- For convenience, we show complete sweeps of liveness and points-to-analysis repeatedly
- This is not required by the computation
- The data flow equations define a single fixed point computation

\[ A_{in} \text{ and } A_{out} \text{ are restricted to } L_{in} \text{ and } L_{out} \]

\[ L_{out} = \begin{cases} \emptyset & \text{n is } End_p \\ \bigcup_{s \in \text{succ}(n)} L_{in_s} & \text{otherwise} \end{cases} \]

\[ L_{in} = \left( L_{out} - \text{Kill}_n \right) \cup \text{Ref}_n \]

\[ A_{in} = \begin{cases} L_{in} \times \{?\} & \text{n is } Start_p \\ \left( \bigcup_{p \in \text{pred}(n)} A_{out_p} \right) \big| L_{in} & \text{otherwise} \end{cases} \]

\[ A_{out} = \left( \left( A_{in} - \text{Kill}_n \times \text{Var} \right) \cup \left( \text{Def}_n \times \text{Pointee}_n \right) \right) \big| L_{out} \]
First Round of Liveness Analysis and Points-to Analysis

Strong liveness: y is not made live because z is not live

Second Round of Liveness Analysis and Points-to Analysis
## Second Round of Liveness Analysis and Points-to Analysis

Consider the following expressions:

- \( x = \& y \)
- \( y = \& z \)
- \( z = \& u \)

Let's analyze the points-to and liveness with some examples:

1. **Points-to Analysis**
   - \( x = \& y \)
   - \( y = \& z \)
   - \( z = \& u \)

   The points-to analysis can be represented as:
   - \( \{u\} \rightarrow u \)
   - \( \{x, y, z\} \rightarrow x \rightarrow y \rightarrow z \rightarrow u \)
   - \( \{u, x\} \rightarrow x \rightarrow y \rightarrow z \rightarrow u \)
   - \( \{u\} \rightarrow u \)
   - \( \{x, y, z\} \rightarrow x \rightarrow y \rightarrow z \rightarrow u \)
   - \( \{u\} \rightarrow u \)
   - \( \{x, y, z\} \rightarrow x \rightarrow y \rightarrow z \rightarrow u \)
   - \( \{u\} \rightarrow u \)

2. **Liveness Analysis**
   - \( \{u\} \rightarrow u \)
   - \( \{x, y, z\} \rightarrow x \rightarrow y \rightarrow z \rightarrow u \)
   - \( \{u\} \rightarrow u \)

### LFCPA Implementation

- LTO framework of GCC 4.6.0
- Naive prototype implementation (Points-to sets implemented using linked lists)
- Implemented FCPA without liveness for comparison
- Comparison with GCC's flow and context insensitive method
- SPEC 2006 benchmarks

### Analysis Time

<table>
<thead>
<tr>
<th>Program</th>
<th>kLoC</th>
<th>Call Sites</th>
<th>L-FCPA Liveness</th>
<th>Points-to L-FCPA</th>
<th>FCPA</th>
<th>GPTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>0.9</td>
<td>33</td>
<td>0.55</td>
<td>0.52</td>
<td>1.9</td>
<td>5.2</td>
</tr>
<tr>
<td>mcf</td>
<td>1.6</td>
<td>29</td>
<td>1.04</td>
<td>0.62</td>
<td>9.5</td>
<td>3.4</td>
</tr>
<tr>
<td>libquantum</td>
<td>2.6</td>
<td>258</td>
<td>2.0</td>
<td>1.8</td>
<td>5.6</td>
<td>4.8</td>
</tr>
<tr>
<td>bzip2</td>
<td>3.7</td>
<td>233</td>
<td>4.5</td>
<td>4.8</td>
<td>28.1</td>
<td>30.2</td>
</tr>
<tr>
<td>parser</td>
<td>7.7</td>
<td>1123</td>
<td>(1.2 \times 10^4)</td>
<td>(145.6)</td>
<td>(4.3 \times 10^5)</td>
<td>422.12</td>
</tr>
<tr>
<td>sjeng</td>
<td>10.5</td>
<td>678</td>
<td>858.2</td>
<td>99.0</td>
<td>(3.2 \times 10^4)</td>
<td>38.1</td>
</tr>
<tr>
<td>hmmer</td>
<td>20.6</td>
<td>1292</td>
<td>90.0</td>
<td>(62.9)</td>
<td>(2.9 \times 10^3)</td>
<td>246.3</td>
</tr>
<tr>
<td>h264ref</td>
<td>36.0</td>
<td>1992</td>
<td>(2.2 \times 10^8)</td>
<td>(2.0 \times 10^8)</td>
<td>?</td>
<td>(4.3 \times 10^8)</td>
</tr>
</tbody>
</table>

### Unique Points-to Pairs

<table>
<thead>
<tr>
<th>Program</th>
<th>kLoC</th>
<th>Call Sites</th>
<th>Unique points-to pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>0.9</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>mcf</td>
<td>1.6</td>
<td>29</td>
<td>41</td>
</tr>
<tr>
<td>libquantum</td>
<td>2.6</td>
<td>258</td>
<td>49</td>
</tr>
<tr>
<td>bzip2</td>
<td>3.7</td>
<td>233</td>
<td>60</td>
</tr>
<tr>
<td>parser</td>
<td>7.7</td>
<td>1123</td>
<td>531</td>
</tr>
<tr>
<td>sjeng</td>
<td>10.5</td>
<td>678</td>
<td>267</td>
</tr>
<tr>
<td>hmmer</td>
<td>20.6</td>
<td>1292</td>
<td>232</td>
</tr>
<tr>
<td>h264ref</td>
<td>36.0</td>
<td>1992</td>
<td>1683</td>
</tr>
</tbody>
</table>
Points-to Information is Small and Sparse

LFCPA Observations

- Usable pointer information is very small and sparse
- Data flow propagation in real programs seems to involve only a small subset of all possible data flow values
- Earlier approaches reported inefficiency and non-scalability because they computed far more information than the actual usable information

LFCPA Conclusions

- Building quick approximations and compromising on precision may not be necessary for efficiency
- Building clean abstractions to separate the necessary information from redundant information is much more significant
- Our experience of points-to analysis shows that
  - Use of liveness reduced the pointer information . . .
  - which reduced the number of contexts required . . .
  - which reduced the liveness and pointer information . . .
- Approximations should come after building abstractions rather than before

LFCPA Lessons: The Larger Perspective

exhaustive computation
computation restricted to usable information
incremental computation
demand driven computation

What should be computed?
Minimum Computation

When should it be computed?
Early Computation
Late Computation

Do not compute what you don’t need!

Who defines what is needed?
Client
LFCPA Lessons: The Larger Perspective

- exhaustive computation
- computation restricted to usable information
- incremental computation
- demand driven computation

What should be computed?  
Maximum Computation  
Early Computation

When should it be computed?  
Minimum Computation  
Late Computation

Do not compute what you don’t need!

Who defines what is needed?  
Algorithm, Data Structure

Definition of Analysis

Avoid computing some values because
- they have been computed before, or
- they can just be “adjusted”, or
- they are equivalent to some other values

E.g. Value based termination of call strings, Work list based methods, BDDs

Do not compute what you don’t need!

Who defines what is needed?  
Algorithm, Data Structure

No One!
LFCPA Lessons: The Larger Perspective

- **exhaustive computation**
- **computation restricted to usable information**
- **incremental computation**
- **demand driven computation**

What should be computed?

- **Maximum Computation**
- **Minimum Computation**
- **Early Computation**
- **Late Computation**

When should it be computed?

- **Do not compute what you don’t need!**

Who defines what is needed?

These seem orthogonal and may be used together

---

Tutorial Problems for FCPA and LFCPA

- Perform may points-to analysis by deriving must info using “?” in BI
- Perform liveness based points-to analysis

![Diagram of computation stages]

---

An Outline of Pointer Analysis Coverage

- The larger perspective
- Comparing Points-to and Alias information
- Flow Insensitive Points-to Analysis
- Flow Sensitive Points-to Analysis
- Pointer Analyses: An Engineer’s Landscape
- Liveness Based Points-to Analysis
- Generalizations to Heap, Arrays, Pointer Arithmetic, and Unions

---

Original LFCPA Formulation

- **Data flow equations**
  - \( L_{in}, L_{out} \)
  - \( A_{in}, A_{out} \)

- **Extractors for statements**
  - \( Def, Kill, Ref, Pointee \)

- **Lattices**
  - \( 2^P \times Var, 2^P \)

- **Named locations**
  - \( Var, Pointers P, \)
Formulating Generalizations in LFCPA

Data flow equations
\(Lin/Lout, Ain/Aout\)

Extractors for statements
\(Def, Kill, Ref, Pointee\)

Extractors for pointer expressions
\(lval, rval, deref, ref\)

Lattices
\(2^S \times 2^T\)

Named locations
\(\text{Variables } Var, \text{ Pointers } P, \text{ Allocation Sites } H, \text{ Fields } F, pF, npF, \text{ Offsets } C\)

Generalization for Heap and Structures
- Grammar.
  \[
  \alpha := \text{malloc} \mid \& \beta \mid \beta \\
  \beta := x \mid \beta.f \mid \beta \rightarrow f \mid \ast \beta
  \]
  where \(\alpha\) is a pointer expression, \(x\) is a variable, and \(f\) is a field

- Memory model: Named memory locations. No numeric addresses

\[
S = P \cup H \cup S_p \quad \text{(source locations)}
\]
\[
T = Var \cup H \cup S_m \cup \{?\} \quad \text{(target locations)}
\]
\[
S_p = R \times npF^* \times pF \quad \text{(pointers in structures)}
\]
\[
S_m = R \times npF^* \times (pF \cup npF) \quad \text{(other locations in structures)}
\]

Named Locations for Pointer Expressions

```c
typedef struct B {
    ...
    struct B *f;
} sB;

typedef struct A {
    ...;
    struct B g;
} sA;
sA *a;
sB *x, *y, b;

1. a = (sA*) malloc(sizeof(sA));
2. y = &a->g;
3. b = x;
4. x.f = &y;
5. return x->f->f;
```

L- and R-values of Pointer Expressions

**lval**
\[
lval(\alpha, A) = \begin{cases}
\{\sigma\} & (\alpha \equiv \sigma, \sigma \in Var) \\
\{\sigma.f \mid \sigma \in lval(\beta, A)\} & \alpha \equiv \beta.f \\
\{\sigma.f \mid \sigma \in lval(\beta, A), \sigma \neq \sigma.f\} & \alpha \equiv \beta \rightarrow f \\
\{\sigma \mid \sigma \in lval(\beta, A), \sigma \neq \sigma.f\} & \alpha \equiv \ast \beta \\
\emptyset & \text{otherwise}
\end{cases}
\]

**rval**
\[
rval(\alpha, A) = \begin{cases}
lval(\beta, A) \quad & \alpha \equiv \& \beta \\
\{lval(\beta, A)\} \quad & \alpha \equiv \text{malloc} \land oi = \text{get_heap_loc}() \\
A(lval(\alpha, A) \cap S) \quad & \text{otherwise}
\end{cases}
\]
Defining Extractor Functions

- Pointer assignment statement $lhs_n = rhs_n$
  
  $Def_n = lval(lhs_n, Ain_n)$
  $Kill_n = lval(lhs_n, Must(Ain_n))$
  $Ref_n = \begin{cases} 
  \text{deref}(lhs_n, Ain_n) & \text{if } \text{Def}_n \cap \text{Lout}_n = \emptyset \\
  \text{deref}(lhs_n, Ain_n) \cup \text{ref}(\text{rhs}_n, Ain_n) & \text{otherwise}
  \end{cases}$
  $\text{Pointee}_n = rval(rhs_n, Ain_n)$

- Use $\alpha$ statement

  $Def_n = Kill_n = \text{Pointee}_n = \emptyset$
  $Ref_n = \text{ref} (\alpha, Ain_n)$

- Any other statement

  $Def_n = Kill_n = Ref_n = \text{Pointee}_n = \emptyset$

Extensions for Handling Arrays and Pointer Arithmetic

- Grammar.

  $\alpha ::= \text{malloc} \mid \& \beta \mid \beta \mid \& \beta + e$
  $\beta ::= x \mid \beta.f \mid \beta \rightarrow f \mid *\beta \mid \beta[e] \mid \beta + e$

- Memory model: Named memory locations. No numeric addresses

  - No address calculation
  - R-values of index expressions retained for each dimension
  - If $rval(x) = 10$, then $rval(a.f[5][2 + x].g) = a.f.5.12.g$
  - Sizes of the array elements ignored

  $S = P \cup H \cup G_p$ (source locations)
  $T = \text{Var} \cup H \cup G_m \cup \{?\}$ (target locations)
  $G_p = R \times (C \cup npF)^* \times (C \cup pF)$ (pointers in aggregates)
  $G_m = R \times (C \cup npF)^* \times (C \cup pF \cup npF)$ (locations in aggregates)

Extending L-Value Computation to Arrays and Pointer Arithmetic

- Pointer arithmetic does not have an l-value
- For handling arrays

  - evaluate index expressions using $\text{eval}$ and accumulate offsets
  - if $e$ cannot be evaluated at compile time, $\text{eval} = \bot$ (i.e. array accesses in that dimension are treated as index-insensitive)

\[
\begin{align*}
\text{lval}(\alpha, A) & = \left\{ \sigma \mid \sigma \in \text{lval}(\beta, A), \sigma \neq \bot \right\} \quad \alpha \equiv \beta \rightarrow f \\
\text{lval}(\alpha, A) & = \left\{ \{\sigma \mid \sigma \in \text{lval}(\beta, A), \sigma \neq \bot\} \cup \{\sigma.\text{eval} \mid \sigma \in \text{lval}(\beta, A)\} \mid \alpha \equiv \beta[e] \right\} \quad \alpha \equiv *\beta \\
\text{lval}(\alpha, A) & = \emptyset \quad \alpha \equiv \beta \& \beta
\end{align*}
\]

 Extending R-Value Computation to Arrays and Pointer Arithmetic

For handling pointer arithmetic

- If the r-value of the pointer is an array location, add $\text{eval}$ to the offset
- Otherwise, over-approximate the pointees to all possible locations

\[
\begin{align*}
\text{rval}(\alpha, A) = \left\{ \begin{array}{ll}
\text{lval}(\beta, A) & \alpha \equiv \& \beta \\
\{\alpha\} & \alpha \equiv \text{malloc} \land \alpha = \text{get_heap_loc}() \\
T & (\alpha \equiv \beta + e) \\
(\exists \sigma \in \text{rval}(\beta, A), \sigma \neq \sigma'.c, \sigma' \in T, c \in C) & (\alpha \equiv \beta + e) \\
(\sigma.c + \text{eval}) & (\alpha \equiv \beta + e) \\
A(\text{lval}(\alpha, A) \cap S) & \text{otherwise}
\end{array} \right.
\end{align*}
\]
Motivating Example for Heap Liveness Analysis

If the while loop is not executed even once.

1. \( w = x \) // \( x \) points to \( m_a \)
2. while \( (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)

Motivating Example for Heap Liveness Analysis

If the while loop is executed once.

1. \( w = x \) // \( x \) points to \( m_a \)
2. while \( (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)

Motivating Example for Heap Liveness Analysis

If the while loop is executed twice.

1. \( w = x \) // \( x \) points to \( m_a \)
2. while \( (x.data < \text{max}) \)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)
The Moral of the Story

- Mappings between access expressions and l-values keep changing
- This is a *rule* for heap data
  For stack and static data, it is an *exception*!
- Static analysis of programs has made significant progress for stack and static data.

What about heap data?
- Given two access expressions at a program point, do they have the same l-value?
- Given the same access expression at two program points, does it have the same l-value?

Our Solution

```
y = z = null
1 w = x
w = null
2 while (x.data < max)
{ x.lptr = null
  x.rptr = x.lptr.rptr = null
  x.lptr.lptr.rptr = null
  x = x.rptr } 
3 x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x = y = null
4 y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
5 z = New class of z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
6 y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = z = null
```

HeapStack
```
While loop is not executed even once
```
```
y = z = null
1 w = x
w = null
2 while (x.data < max)
{ x.lptr = null
  x.rptr = x.lptr.rptr = null
  x.lptr.lptr.rptr = null
  x = x.rptr } 
3 x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
4 y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
5 z = New class of z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
6 y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = z = null
```

```
While loop is executed once
```
```
Our Solution

1. w = x
2. while (x.data < max)
    { x.lptr = null
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
    }
3. x = x.rptr
4. y = x.lptr
5. z = New class of z
6. y = y.lptr
7. z.sum = x.data + y.data
x = y = z = null

Some Observations

• The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
• Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution
• A static analysis can discover only some invariants
Some Observations

y = z = null
1. w = x
   w = null
2. while (x.data < max)
   { x.lptr = null
     x.rptr = x.lptr.rptr = null
     x.lptr.lptr.lptr = null
     x.lptr.lptr.rptr = null
   }
3. x = x.rptr
4. y = x.lptr
   y.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
5. z = New class
   z.lptr = z.rptr = null
6. y = y.lptr
7. y.lptr = y.rptr = null
8. z.sum = x.data + y.data
   x = y = z = null

Heap Stack
x z w y a
b c d e f g h i j k l m n o
Ap
rptr lptr

An Overview of Heap Reference Analysis

- A reference (called a link) can be represented by an access path.
  Eg. “x → lptr → rptr”
- A link may be accessed in multiple ways
- Setting links to null
  - Alias Analysis. Identify all possible ways of accessing a link
  - Liveness Analysis. For each program point, identify “dead” links (i.e. links which are not accessed after that program point)
  - Availability and Anticipability Analyses. Dead links should be reachable for making null assignment.
  - Code Transformation. Set “dead” links to null

Assumptions

For simplicity of exposition

- Java model of heap access
  ▶ Root variables are on stack and represent references to memory in heap.
  ▶ Root variables cannot be pointed to by any reference.
- Simple extensions for C++
  ▶ Root variables can be pointed to by other pointers.
  ▶ Pointer arithmetic is not handled.
Key Idea #1: Access Paths Denote Links

- Root variables: x, y, z
- Field names: rptr, lptr
- Access path: \( x \rightarrow rptr \rightarrow lptr \)
  Semantically, sequence of "links"
- Frontier: name of the last link
- Live access path: If the link corresponding to its frontier is used in future

What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for accessing the contents of the corresponding target object:

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum = x.rptr.data</td>
<td>x, O₁, O₂</td>
<td>x, x → rptr</td>
</tr>
<tr>
<td>if (x.rptr.data &lt; sum)</td>
<td>x, O₁, O₂</td>
<td>x, x → rptr</td>
</tr>
</tbody>
</table>

Reading a link for copying the contents of the corresponding target object:

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = x.rptr</td>
<td>x, O₁</td>
<td>x</td>
</tr>
<tr>
<td>x.lptr = y</td>
<td>x, O₁, y</td>
<td>x</td>
</tr>
</tbody>
</table>

Stack

Heap

What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for accessing the contents of the corresponding target object:
What Makes a Link Live?

Assuming that a statement is the last statement in the program, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

**Example**

<table>
<thead>
<tr>
<th>Objects read</th>
<th>Live access paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x.lptr == null)</td>
<td>x, O₁</td>
</tr>
</tbody>
</table>

Reading a link for comparing the address of the corresponding target object:

### Liveness: Assignment Vs. Conditions

- "x = y.l" makes only y live
- "x == y.l" makes all x, y, and y.l live
- The text message forwarding analogy
  - If no one were to read a forwarded message, contents do not matter
  - If people are going to read the message, it needs to be forwarded selectively, and then contents matter 😊

---

Liveness Analysis

Statement involving memory references

Live Access Paths

Effect of the statement on the access paths

Program

Semantic Information
Key Idea #2: Transfer of Access Paths

$\text{x} = \text{x.n}$

$x$ after the assignment is same as $x\rightarrow n$ before the assignment

... = $x.r.d$

Analysis

Generated

$\{x, x\rightarrow n, x\rightarrow n\rightarrow r\}$

Killed

$\{x, x\rightarrow r\}$

$x$ after the assignment is same as $x\rightarrow n$ before the assignment

Nullifying $y\rightarrow n$ will have the side effect of nullifying $x\rightarrow n$

Key Idea #3: Liveness Closure Under Link Aliasing

$x = y$

$x.n$ and $y.n$ are link aliases

$x\rightarrow n$ is live $\Rightarrow$ $y\rightarrow n$ is live

Nullifying $y\rightarrow n$ will have the side effect of nullifying $x\rightarrow n$

Explicit and Implicit Liveness

$x = y$

$x\rightarrow n$ is live $\Rightarrow$ $y\rightarrow n$ is live

$\ldots = x.n.d$

$y\rightarrow n$ is implicitly live

$x\rightarrow n$ is explicitly live

Key Idea #4: Aliasing is Required with Explicit Liveness

1 $x = y$

Spurious

$\{x, y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

Effect of Aliasing

$y\rightarrow p \equiv x\rightarrow p$

$y\rightarrow p\rightarrow q \equiv x\rightarrow p\rightarrow q$

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

Required Liveness

$\{x, y, t\rightarrow t\rightarrow q\}$

Missing

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

Spurious

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$

use $y.q.d$
Key Idea #4: Aliasing is Required with Explicit Liveness

1. \( x = y \)
   - **Explicit Liveness**: \( \{x, y, y \rightarrow p, y \rightarrow p \rightarrow q\} \)
   - **Required Liveness**: \( \{x, y, t, t \rightarrow q\} \)

2. \( x.p = t \)
   - Spurious
   - Effect of Aliasing
     \( y \rightarrow p \equiv x \rightarrow p \)
     \( y \rightarrow p \rightarrow q \equiv x \rightarrow p \rightarrow q \)

3. \( y = y.p \)
   - Missing

4. \( \text{use } y.q.d \)

The need of link alias closure of LHS
- Transferring liveness to RHS (soundness)
- Killing liveness (precision)

Link alias closure of RHS can be computed later for implicit liveness

Notation for Defining Flow Functions for Explicit Liveness

- **Basic entities**
  - Variables \( u, v \in \text{Var} \)
  - Pointer variables \( w, x, y, z \in P \subseteq \text{Var} \)
  - Pointer fields \( f, g, h \in pF \)
  - Non-pointer fields \( a, b, c, d \in npF \)

- **Additional notation**
  - Sequence of pointer fields \( \sigma \in pF^* \) (could be \( \epsilon \))
  - Access paths \( \rho \in P \times pF^* \)
  - Example: \( \{x, x \rightarrow f, x \rightarrow f \rightarrow g\} \)
  - Summarized access paths rooted at \( x \) or \( x \rightarrow \sigma \) for a given \( x \) and \( \sigma \)
     - \( x \rightarrow * = \{x \rightarrow \sigma \mid \sigma \in pF^*\} \)
     - \( x \rightarrow \sigma \rightarrow * = \{x \rightarrow \sigma \rightarrow \sigma' \mid \sigma' \in pF^*\} \)

Data Flow Equations for Explicit Liveness Analysis

\[
\begin{align*}
\text{In}_n &= (\text{Out}_n - \text{Kill}_n(\text{Out}_n)) \cup \text{Gen}_n(\text{Out}_n) \\
\text{Out}_n &= \begin{cases} 
\text{BL} & \text{if } n \text{ is End} \\
\bigcup_{s \in \text{succ}(n)} \text{In}_s & \text{otherwise}
\end{cases}
\end{align*}
\]

Flow Functions for Explicit Liveness Analysis

Let \( A \) denote May Aliases at the exit of node \( n \)

<table>
<thead>
<tr>
<th>Statement ( n )</th>
<th>( \text{Gen}_n(X) )</th>
<th>( \text{Kill}_n(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y )</td>
<td>{y \rightarrow \sigma \mid x \rightarrow \sigma \in X}</td>
<td>( x \rightarrow * )</td>
</tr>
<tr>
<td>( x = y.f )</td>
<td>{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X}</td>
<td>( x \rightarrow * )</td>
</tr>
<tr>
<td>( x.f = y )</td>
<td>{y \rightarrow \sigma \mid z \rightarrow f \rightarrow \sigma \in X, z \in A(x)} \cup {z \rightarrow f \rightarrow * }</td>
<td>\text{Must}(A)(x) }</td>
</tr>
<tr>
<td>( x = \text{new} )</td>
<td>( \emptyset )</td>
<td>( x \rightarrow * )</td>
</tr>
<tr>
<td>( x = \text{null} )</td>
<td>( \emptyset )</td>
<td>( x \rightarrow * )</td>
</tr>
<tr>
<td>other</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
Flow Functions for Explicit Liveness Analysis

Let $A$ denote May Aliases at the exit of node $n$

<table>
<thead>
<tr>
<th>Statement $n$</th>
<th>$\text{Gen}_n(X)$</th>
<th>$\text{Kill}_n(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y$</td>
<td>${y \rightarrow \sigma \mid x \rightarrow \sigma \in X}$</td>
<td>$x \rightarrow \ast$</td>
</tr>
<tr>
<td>$x = y.f$</td>
<td>${y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X}$</td>
<td>$x \rightarrow \ast$</td>
</tr>
<tr>
<td>$x.f = y$</td>
<td>${y \rightarrow \sigma \mid z \rightarrow f \rightarrow \sigma \in X, z \in A(x)}$</td>
<td>$\bigcup_{z \in \text{Must}(A)(x)} z \rightarrow f \rightarrow \ast$</td>
</tr>
<tr>
<td>$x = \text{new}$</td>
<td>$\emptyset$</td>
<td>$x \rightarrow \ast$</td>
</tr>
<tr>
<td>$x = \text{null}$</td>
<td>$\emptyset$</td>
<td>$x \rightarrow \ast$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

May link aliasing for soundness
Must link aliasing for precision

Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An All Paths problem
Liveness of Heap References: An *Any Path* problem

\[ \{x, x \to n, x \to n \to n, x \to n \to n \to r, x \to n \to \ldots \to n \to r\} \]

\[ \{x, x \to r\} \cup \{x, x \to n, x \to n \to r\} \]

\[ \{x, x \to r\} \]

\[ \ldots = x \cdot r \cdot d \]

*Infinite Number of Unbounded Access Paths*

*Finite Number of Bounded Structures*
Key Idea #6: Include Program Point in Graphs

Different occurrences of \( n \)'s in an access path are indistinguishable (pattern of subsequent dereferences remains same)

Access Graph:

\[
\begin{align*}
1 & \quad x = x.n \\
2 & \quad \ldots = x.n.r.d
\end{align*}
\]

Different occurrences of \( n \)'s in an access path are distinct (pattern of subsequent dereferences could be distinct)

Access Graph:

\[
\begin{align*}
1 & \quad x = x.n \\
2 & \quad \ldots = x.n.r.d
\end{align*}
\]
Inclusion of Program Point Facilitates Summarization

Iteration #3

\[ x = x \cdot n \]

\[ \ldots = x \cdot r \cdot d \]

Access Graph and Memory Graph

Program Fragment

\[
\begin{align*}
&x.l = y.r \quad 1 \\
&\text{if} (x.l.n == y.r.n) \quad 2
\end{align*}
\]

Memory Graph

Access Graphs

- Memory Graph: Nodes represent locations and edges represent links (i.e. pointers).
- Access Graphs: Nodes represent dereference of links at particular statements. Memory locations are implicit.

Lattice of Access Graphs

- Finite number of nodes in an access graph for a variable
- \( \sqcup \) induces a partial order on access graphs
  \[ \Rightarrow \] a finite (and hence complete) lattice
  \[ \Rightarrow \] All standard results of classical data flow analysis can be extended to this analysis.

Termination and boundedness, convergence on MFP, complexity etc.

Access Graph Operations

- Union. \( G \sqcup G' \)
- Path Removal
  \( G \ominus R \) removes those access paths in \( G \) which have \( \rho \in R \) as a prefix
- Factorization (\( / \))
- Extension
Defining Factorization

Given statement $x \cdot n = y$, what should be the result of transfer?

<table>
<thead>
<tr>
<th>Live AP</th>
<th>Memory Graph</th>
<th>Transfer</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \rightarrow n \rightarrow r$</td>
<td>$x \rightarrow n \rightarrow r$</td>
<td>$y \rightarrow r$</td>
<td>$r$ (LHS is contained in the live access path)</td>
</tr>
<tr>
<td>$x \rightarrow n$</td>
<td>$x \rightarrow n$</td>
<td>$y$</td>
<td>$\epsilon$ (LHS is contained in the live access path)</td>
</tr>
<tr>
<td>$x$</td>
<td>$x$</td>
<td>$y$</td>
<td>?? (LHS is not contained in the live access path)</td>
</tr>
<tr>
<td>No transfer</td>
<td>No transfer</td>
<td>No transfer</td>
<td>So no remainder</td>
</tr>
</tbody>
</table>

Semantics of Access Graph Operations

- $P(G)$ is the set of all paths in graph $G$
- $P(G, M)$ is the set of paths in $G$ terminating on nodes in $M$
- $S$ is the set of remainder graphs
- $P(S)$ is the set of all paths in all remainder graphs in $S$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Access Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union $G_3 = G_1 \sqcup G_2$</td>
<td>$P(G_3) \supseteq P(G_1) \cup P(G_2)$</td>
</tr>
<tr>
<td>Path Removal $G_2 = G_1 \setminus X$</td>
<td>$P(G_2) \supseteq P(G_1) - {\rho \rightarrow \sigma</td>
</tr>
<tr>
<td>Factorization $S = G_1/\rho$</td>
<td>$P(S) = {\sigma</td>
</tr>
<tr>
<td>Extension $G_2 = (G_1, M) # \emptyset$</td>
<td>$P(G_2) \supseteq P(G_1) \cup {\rho \rightarrow \sigma</td>
</tr>
</tbody>
</table>

Semantics of Access Graph Operations: Examples

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $x = x, l$</td>
<td>$g_1 \Rightarrow x$</td>
<td>$rg_1 \Rightarrow \emptyset$</td>
</tr>
<tr>
<td>2 $y = x \cdot r, d$</td>
<td>$g_4 \Rightarrow x, y \Rightarrow \emptyset$</td>
<td>$g_5 \Rightarrow x, y \Rightarrow \emptyset$</td>
</tr>
</tbody>
</table>

Extension $\sigma$ represents remainder.

Semantics of Access Graph Operations

- $P(G)$ is the set of all paths in graph $G$
- $P(G, M)$ is the set of paths in $G$ terminating on nodes in $M$
- $S$ is the set of remainder graphs
- $P(S)$ is the set of all paths in all remainder graphs in $S$
### Access Graph Operations: Examples

<table>
<thead>
<tr>
<th>Program</th>
<th>Access Graphs</th>
<th>Remainder Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$g_1 \to x$</td>
<td>$r_{g_1} \to r_1$</td>
</tr>
<tr>
<td>2</td>
<td>$y = x.r.d$</td>
<td>$g_4 \to x \to l \circ h_1 \circ f_2$</td>
</tr>
</tbody>
</table>

- **Union**: $g_3 \cup g_4 = g_6$  
  $g_6 \cup \{x \to l\} = g_3$  
  $g_6 \cup \{x \to l\} = g_4$  
  $g_6 \cup \{x \to l\} = g_6$  
  $g_6 \cup \{x \to l\} = g_5$

- **Path Removal**: $g_6 / x = \{rg_1\}$  
  $(g_3, \{h_1\}) \# \{rg_1\} = g_4$  
  $(g_3, \{h_1\}) \# \{rg_2\} = g_6$  
  $(g_3, \{h_1\}) \# \{rg_3\} = g_5$

- **Factorisation**: $g_5 / x = s_1 g_2$  
  $(g_2, \{r_2\}) \# \{s_1\} = g_5$  
  $(g_2, \{r_2\}) \# \{s_1 \cup \{r_1\}\} = g_6$

- **Extension**: $g_4 / x = r = \emptyset$  
  $(g_2, \{r_2\}) \# \emptyset = \emptyset r_5$

- **Remainder is empty**
- **Quotient is empty**

### Data Flow Equations for Explicit Liveness Analysis: Access Graphs Version

- $In_n = (Out_n \oplus \text{Kill}_n(Out_n)) \cup \text{Gen}_n(Out_n)$
- $Out_n = \begin{cases} Bi & n \text{ is End} \\ \bigcup_{s \in \text{succ}(n)} In_s & \text{otherwise} \end{cases}$

- $In_n$, $Out_n$, and $\text{Gen}_n$ are access graphs
- $\text{Kill}_n$ is a set of access paths

### Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let $A$ denote May Aliases at the exit of node $n$

<table>
<thead>
<tr>
<th>Statement $n$</th>
<th>$\text{Gen}_n(X)$</th>
<th>$\text{Kill}_n(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y$</td>
<td>${y \to \sigma \mid x \to \sigma \in X}$</td>
<td>$x \to \ast$</td>
</tr>
<tr>
<td>$x = y.f$</td>
<td>${y \to f \to \sigma \mid x \to \sigma \in X}$</td>
<td>$x \to \ast$</td>
</tr>
<tr>
<td>$x.f = y$</td>
<td>${y \to \sigma \mid z \to f \to \sigma \in X, z \in A(x)}$</td>
<td>$\bigcup_{z \in \text{Must}(A(x))} z \to f \to \ast$</td>
</tr>
<tr>
<td>$x = \text{new}$</td>
<td>$\emptyset$</td>
<td>$x \to \ast$</td>
</tr>
<tr>
<td>$x = \text{null}$</td>
<td>$\emptyset$</td>
<td>$x \to \ast$</td>
</tr>
<tr>
<td>other</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

- May link aliasing for soundness
- Must link aliasing for precision
Flow Functions for Explicit Liveness Analysis: Access Graphs Version

- \( A \) denotes May Aliases at the exit of node \( n \)
- \( \text{mkGraph}(\rho) \) creates an access graph for access path \( \rho \)

<table>
<thead>
<tr>
<th>Statement ( n )</th>
<th>( \text{Gen}_n(X) )</th>
<th>( \text{Kill}_n(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y )</td>
<td>( \text{mkGraph}(y)#(X/x) )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>( x = y.f )</td>
<td>( \text{mkGraph}(y\rightarrow f)#(X/x) )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>( x.f = y )</td>
<td>( \text{mkGraph}(y)#(\bigcup_{z \in A(x)}(X/(z\rightarrow f))) )</td>
<td>( {z \rightarrow f \mid z \in \text{Must}(A)(x)} )</td>
</tr>
<tr>
<td>( x = \text{new} )</td>
<td>( \emptyset )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>( x = \text{null} )</td>
<td>( \emptyset )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>other</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Liveness Analysis of Example Program: 1st Iteration

1. \( w = x \)
2. \( \text{while} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New\ class\ of} \ z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

Liveness Analysis of Example Program: 2nd Iteration

1. \( w = x \)
2. \( \text{while} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New\ class\ of} \ z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

Liveness Analysis of Example Program: 3rd Iteration

1. \( w = x \)
2. \( \text{while} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New\ class\ of} \ z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Liveness Analysis of Example Program: 4th Iteration

Construct access graphs at the entry of block 1 for the following programs

A

1. w = x
2. while (x.data < max)

3. Use x.r.d

4. y = y.lptr

5. z = New class of z

6. y = y.lptr

7. z.sum = x.data + y.data

Why are the access graphs for programs B and D identical?

D

A

The final magic!!

Rotate each picture anti-clockwise by 90° and compare it with its access graph

The structure of access graph of variable x is identical to the control flow structure between pointer assignments of x

Sep 2017 IIT Bombay
Tutorial Problem for Explicit Liveness (2)

- Unfortunately the student who constructed these access graphs forgot to attach statement numbers as subscripts to node labels and has misplaced the programs which gave rise to these graphs.
- Please help her by constructing CFGs for which these access graphs represent explicit liveness at some program point in the CFGs.

![Access Graphs]

Tutorial Problem for Explicit Liveness (3)

- Compute explicit liveness for the program.
- Are the following access paths live at node 1? Show the corresponding execution sequence of statements:

  - P1: y → m → l
  - P2: y → l → n → m
  - P3: y → l → n → l
  - P4: y → n → l → n

  x = z₁
  x = y₁
  xₜ = yₜ mₜ

Availability and Anticipability Analyses

- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.
- Finiteness.
  - An anticipable (available) access path must be anticipable (available) along every paths. Thus unbounded paths arising out of loops cannot be anticipable (available).
  - Due to “every control flow path nature”, computation of anticipable and available access paths uses ∩ as the confluence. Thus the sets are bounded.
  - ⇒ No need of access graphs.
### Availability Analysis of Example Program

```plaintext
1 \ w = x
∅
∅

2 while (x.data < max)
```

```plaintext
3 \ x = x.rptr
```

```plaintext
4 \ y = x.lptr
```

```plaintext
5 \ z = New class of z
```

```plaintext
6 \ y = y.lptr
```

```plaintext
7 \ z.sum = x.data + y.data
```
The Resulting Program

```plaintext
y = z = null
1 w = x
w = null
2 while (x.data < max)
   { x = x.rptr
     x.lptr = null
     x.rptr = x.lptr.rptr = null
     x.lptr.lptr.lptr = null
     x.lptr.lptr.rptr = null
   }
3 x.rptr = x.lptr.rptr = null
4 y = x.lptr
y.lptr = y.rptr = null
5 z = New class of x
z.lptr = z.rptr = null
6 y = y.lptr
y.lptr = y.rptr = null
7 z.sum = x.data + y.data
x = y = z = null
```

Overapproximation Caused by Our Summarization

- The program allocates \( x \rightarrow p \) in one iteration and uses it in the next
- Only \( x \rightarrow p \rightarrow p \) is live at Out
- Third \( p \) is reallocated
Overapproximation Caused by Our Summarization

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- Only $x \rightarrow p \rightarrow p$ is live at $Out_2$
- $x \rightarrow p \rightarrow p$ is live at $Out_2$
- $x \rightarrow p \rightarrow p$ is dead at $Out_2$
- First $p$ used in statement 3
- Second $p$ used in statement 4
- Third $p$ is reallocated

Non-Distributivity of Explicit Liveness Analysis

- The assignment in node 1 removes $x \rightarrow n$ due to the assignment in node 1
- $x \rightarrow n$ is removed from $Out_1$ due to the assignment in node 1
Non-Distributivity of Explicit Liveness Analysis

Non-distributivity of the liveness analysis can be illustrated with the following example:

1. $x.n = \text{null}$
2. $x = x.n$
3. $x.n.n = \text{null}$

Access path $x \rightarrow n \rightarrow r$ (shown in blue color) is a spurious access path that arises due to $\cup$ and is not removed by the assignment in node 1.

Issues Not Covered

- Precision of information
  - Cyclic Data Structures
  - Eliminating Redundant null Assignments
- Properties of Data Flow Analysis:
  - Monotonicity, Boundedness, Complexity
- Interprocedural Analysis
- Extensions for C/C++
- Formulation for functional languages
- Issues that need to be researched: Good alias analysis of heap

BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Conclusions

- Unbounded information can be summarized using interesting insights
  - Contrary to popular perception, heap structure is not arbitrary
    - Heap manipulations consist of repeating patterns which bear a close resemblance to program structure
  - Analysis of heap data is possible despite the fact that the mappings between access expressions and l-values keep changing