Field-Sensitive Points-to-Analysis

Eda GÜNGÖR
s6edguen@uni-bonn.de
Outline

- CFL-reachability
- Aim of the PointsTo Analysis
- Field-sensitive Versus Field-based
- Context-Insensitive Formulation
  - Graph representation
  - Analysis Grammar
- Context-Insensitive Points-To Analysis
  - Refinement algorithm
  - Regular Approximation
  - Refinement with Regular Reachability
  - Evaluation
What is CFL-reachability?

- Context-free language reachability
- An extension of traditional graph reachability
- Interesting paths in G are described by the language “L”
- The CFL-reachability problem requires determining for all node pairs, if there is an L-path between these nodes
CFL-reachability with an Input Graph

- Each path \( p \) has a string \( s(p) \), constructed by edge labels
  - \( s(p) = "()" \)

- \( p \) is an L-path iff \( s(p) \in L \)
  - There is an L-path from \( s \) to \( t \)
  - There is no L-path from \( u \) to \( t \)

- \( t \) is L-reachable from \( s \) (simply, \( s \ L \ t \))
  - \( t \) is not L-reachable from \( u \)
The Aim of the PointsTo Analysis

- Computing the best possible (most precise) flow-insensitive points-to information
  - **Flow-insensitive**: treats each method as if its control-flow graph contains all possible edges
  - Requires a subset-based, field-sensitive manner
  - Input program with
    - No arrays
    - No recursive method calls
Field-sensitive versus Field-based

\[
x = \text{new Obj();} \quad \text{// o1}
y = \text{new Obj();} \quad \text{// o2}
x.f = \text{new Obj();} \quad \text{// o3}
y.f = \text{new Obj();} \quad \text{// o4}
z = x.f; \quad \text{// pt(z) = ?} \quad \text{pt(z): points-to set of variable } z
\]

**Field-sensitive analysis:** Reasons separately about the instance fields of each abstract object. Since, \( x \) and \( y \) cannot be aliased, the result is 
\[ \text{pt}(z) = \{o3\} \]

**Field-based analysis:** Treats each instance field as a global variable, yielding 
\[ \text{pt}(z) = \{o3, o4\} \]
Context-Insensitive Formulation

Graph Representation
Analysis Grammar
## Graph Representation

<table>
<thead>
<tr>
<th>Statement</th>
<th>Graph Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>s: x = new T()</td>
<td>( o_s \xrightarrow{\text{new}} x ) (allocation statements)</td>
</tr>
<tr>
<td>x = y</td>
<td>( y \xrightarrow{\text{assign/assignglobal}} x ) (copy statements)</td>
</tr>
<tr>
<td>x = y.f</td>
<td>( y \xrightarrow{\text{getfield}[f]} x ) (heap reads)</td>
</tr>
<tr>
<td>x.f = y</td>
<td>( y \xrightarrow{\text{putfield}[f]} x ) (heap writes)</td>
</tr>
<tr>
<td>s: x = m(a_1,a_2,..,a_k)</td>
<td>( a_i \xrightarrow{\text{param}[s]} f_{m,i} ) (for ( i \in [1,2,..,k] ))</td>
</tr>
<tr>
<td>ret_m</td>
<td>( \xrightarrow{\text{return}[s]} x )</td>
</tr>
</tbody>
</table>

- Value flow of the statement from right-hand side to left-hand side
- The field name f is part of the edge label
1. \texttt{x = new Obj(); //o₁}
2. \texttt{z = new Obj(); //o₂}
3. \texttt{w = x;}
4. \texttt{y = x;}
5. \texttt{y.f = z;}
6. \texttt{v = w.f;}

A Code Example and Its Graph Representation
Grammar of the Analysis

- The language \( L_F \) is defined by the grammar
  \[
  \text{flowsTo} \rightarrow \text{new} \ (\text{assign})^* 
  \]

- \( x \) is \( L_F \)-reachable from \( o \) iff \( o \in \text{pt}(x) \)

- To track value flow through the heap via \text{putfield}[f] and \text{getfield}[f] statements, \( L_F \) is extended

- Extended \( \text{flowsTo} \) grammar
  \[
  \text{flowsTo} \rightarrow \text{new} \ (\text{assign} \mid \text{putfield}[f] \ alias \ \text{getfield}[f])^* 
  \]
May-aliasing

- Aliasing by the means of flowsTo-paths:
  - x and y are **may-aliased** iff
    - o flowsTo x
    - o flowsTo y
    - o ∈ pt(x) ∩ pt(y)
  for an object o

**Problem:**
- May-aliasing is unsuitable for CFL-reachability since it can only check language membership of a single path

**Solution:**
- Extending the graph representation to allow for **inverse paths**
Inverse Paths

- Enable alias paths by introducing reversed flowsTo paths (flowsTo)
- Connect two may-aliased variables and allow for a single path
- The grammar:

  $$\text{alias} \rightarrow \text{flowsTo} \text{flowsTo}$$
  $$\text{flowsTo} \rightarrow (\text{assign} \mid \text{putfield}[f] \text{alias} \text{getfield}[f])^* \text{new}$$

- An (x alias y)-path can be defined as a path

  $$x \text{flowsTo n flowsTo y} \quad (\text{for some node } n)$$
A Context-free Grammar for \( L_F \)

\[
\begin{align*}
\text{flowsTo} & \rightarrow \text{new} \\
\text{flowsTo} & \rightarrow \text{new} \\
\text{flowsTo} & \rightarrow \text{ciAssign} \quad \text{flowsTo} \\
\text{flowsTo} & \rightarrow \text{ciAssign} \quad \text{flowsTo} \\
\text{flowsTo} & \rightarrow \text{flowsTo} \quad \text{putfield}[f] \quad \text{alias} \quad \text{getfield}[f] \\
\text{flowsTo} & \rightarrow \text{getfield}[f] \quad \text{alias} \quad \text{putfield}[f] \quad \text{flowsTo} \\
\text{alias} & \rightarrow \text{flowsTo} \quad \text{flowsTo} \\
\text{ciAssign} & \rightarrow \text{assign} \quad \mid \quad \text{assignglobal} \quad \mid \quad \text{param}[i] \quad \mid \quad \text{return}[i] \\
\text{ciAssign} & \rightarrow \text{assign} \quad \mid \quad \text{assignglobal} \quad \mid \quad \text{param}[i] \quad \mid \quad \text{return}[i] \\
\text{pointsTo} & \rightarrow \text{flowsTo}
\end{align*}
\]
A Context-free Grammar for $L_F$

- Each $\text{flowsTo}$ inverts the edges of $\text{flowsTo}$ production

- The ciAssign and ciAssign non-terminals treat edges

- Determining a points-to set for a variable $x$ requires solving a backwards $L_F$-reachability problem from $x$

- $\text{pointsTo} \rightarrow \text{flowsTo}$ production makes this backwards-reachability correspondence explicit: $o \in \text{pt}(x) \iff x \text{ pointsTo } o \ (x \text{ flowsTo } o)$
Example: flowsTo-path from $o_2$ to $v$

\[
\begin{align*}
&y \text{ assign } x \text{ new } o_1 \text{ new } x \text{ assign } w \\
&\quad \rightarrow y \text{ ciAssign } x \text{ flowsTo } o_1 \text{ flowsTo } x \text{ ciAssign } w \\
&\quad \rightarrow y \text{ flowsTo } o_1 \text{ flowsTo } w \\
&\quad \rightarrow y \text{ alias } w \\
&\quad \rightarrow o_2 \text{ new } z \text{ putfield}[f] y \text{ alias } w \text{ getfield}[f] v \\
&\quad \rightarrow o_2 \text{ flowsTo } z \text{ putfield}[f] y \text{ alias } w \text{ getfield}[f] v \\
&\quad \rightarrow o_2 \text{ flowsTo } v
\end{align*}
\]
Context-Insensitive Points-To Analysis

Refinement Algorithm
Regular Approximation
Refinement with Regular Reachability
Evaluation
Refinement-based Algorithm

- Solves the Lsf-reachability, simplified Lf-reachability, problem
- Quickly proves that a node is not Lsf-reachable from another node
- Focuses on unbalanced parentheses in the graph and skipping over the rest of it
- Improves performance through “approximation” and “refinement”
  - Approximate analysis: must answer correctly when p is an Lsf-path, but can answer incorrectly when it is not
  - Refinement: gradually removes the imprecision of this analysis, eventually yielding the correct answer when p is not an Lsf-path.
- To maintain correctness, the algorithm only skips sub-paths beginning and ending with matched parentheses
Regular Approximation

- Computes only the initial approximation of the refinement algorithm
  - i.e., reachability over a graph with all possible match edges

- Approximation requires regular reachability over a regular language $R_F$

- Asymptotically less expensive than CFL-reachability

- Has a simple and efficient demand-driven algorithm RegularPT to find points-to information based on RF-reachability
  - A demand-driven: determining which nodes are L-reachable from some specific source node
Regular Reachability

- A regular approximation of $L_F$-reachability via match edges
- $f_{\text{flowsTo}}$ path for $L_F$
  
  \[
  f_{\text{flowsTo}} \rightarrow \text{new} \ (\text{assign} \ | \ \text{putfield}[f] \ alias \ \text{getfield}[f])^*
  \]
- $L_F$-reachability is checking putfield[f] and getfield[f] edges (the balanced parentheses of $L_F$) for the field-sensitivity conditions
- The most expensive part of field-sensitivity
  - Checking for an alias-path between the base variables of putfield[f] and getfield[f] edges
- **Match edges**: are used instead of “putfield[f] alias getfield[f]” path
- $R_F$ is defined by the grammar
  
  \[
  f_{\text{flowsTo}} \rightarrow \text{new} \ (\text{assign} \ | \ \text{match})^*
  \]
A Graph with All Possible Match Edges

- Match edges:  
  from the source of putfield[f] edge to the target of getfield[f] edge
- v is $R_F$-reachable from $o_2$ but not $L_F$-reachable,  
  - Since, there is no alias-path from $q$ to $p$
- Match edges are added without checking the base variables are connected by an alias-path or not

Results:
- $R_F$-reachability over-approximates $L_F$-reachability
- Precision is lost in cases where an $R_F$-path includes an invalid match edge
Refinement with Regular Reachability

- Lost of precision can be recovered by Refinement technique
- Has an algorithm RefinedRegularPT, an extension of the RegularPT
- Removes match edges by checking the existence of an alias-path between the base variables of the corresponding putfield[f] and getfield[f] edges
- Instead of searching for alias-paths, refines a match edge by looking for aliasReg-paths, defined as

\[
\begin{align*}
\text{aliasReg} & \rightarrow \overline{\text{flowsToReg}} \overline{\text{flowsToReg}} \\
\overline{\text{flowsToReg}} & \rightarrow (\overline{\text{assign}} \mid \overline{\text{match}})^* \overline{\text{new}}
\end{align*}
\]
Example: Removing invalid match edges

Refinement by removing invalid match edges

1. Search for an aliasReg-path from q to p
2. No such path exists
3. Match edge w → v can be safely removed from the graph
4. This removal eliminates the aliasReg-path from w to v
5. Match edge z → y can also be safely removed
## Evaluation: Benchmark Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Intra (Live)</th>
<th>Reg (Live)</th>
<th>RefReg (Live)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soot</td>
<td>18.4 (16.0)</td>
<td>94.1 (98.5)</td>
<td>96.9 (99.8)</td>
</tr>
<tr>
<td>compress</td>
<td>26.0 (23.1)</td>
<td>89.1 (96.4)</td>
<td>93.7 (98.9)</td>
</tr>
<tr>
<td>jess</td>
<td>25.4 (22.5)</td>
<td>89.4 (96.6)</td>
<td>93.9 (99.0)</td>
</tr>
<tr>
<td>raytrace</td>
<td>26.1 (23.1)</td>
<td>89.1 (96.4)</td>
<td>93.7 (98.9)</td>
</tr>
<tr>
<td>db</td>
<td>25.7 (22.7)</td>
<td>89.3 (96.5)</td>
<td>93.7 (98.9)</td>
</tr>
<tr>
<td>javac</td>
<td>25.3 (22.3)</td>
<td>89.9 (96.7)</td>
<td>94.1 (98.8)</td>
</tr>
<tr>
<td>mpeg</td>
<td>26.0 (23.1)</td>
<td>89.1 (96.4)</td>
<td>94.4 (98.9)</td>
</tr>
<tr>
<td>jack</td>
<td>27.0 (25.1)</td>
<td>91.8 (97.5)</td>
<td>95.2 (99.2)</td>
</tr>
</tbody>
</table>

- **Percentage results of measuring the precision of the algorithms**
  - Intra: An intraprocedural field-based analysis
  - Reg: RegularPT
  - Refreg: RefinedRegularPT with a 5 second time limit per query
Conclusions

- The algorithms are precise
  - RegularPT and RefinedRegularPT have nearly precision of field-sensitive Andersen’s.
  - RefinedRegularPT provides more precision than RegularPT
- Precision is retained under early termination
  - RegularPT and RefinedRegularPT retain almost all their precision when they run with small time budgets
- Good performance
References

- Refinement-Based Program Analysis Tools, by Sridharan, 2007, University of California, Berkeley
  
  http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-125.html

- Refinement-Based Context-Sensitive Points-To Analysis for Java, by Sridharan and Bodík, 2006, University of California, Berkeley
Thank you for your attention!