Program Analysis in Datalog

Using the tool
bddbbdd System Overview

Java bytecode → Joeq frontend → Input relations → Datalog program → Output relations
Compiler Frontend

- Convert IR into tuples
- Tuples format:
  
  # V0:16 F0:11 V1:16  
  0 0 1               
  0 1 2               
  1470 0 1464
Compiler Frontend

Robust frontends:
- Joeq compiler
- Soot compiler
- SUIF compiler (for C code)

Still experimental:
- Eclipse frontend
- gcc frontend
- ...
Extracting Relations

Idea: Iterate thru compiler IR, numbering and dumping relations of interest.

- Types
- Methods
- Fields
- Variables
- ...

---

cs 510 Software Engineering
joeq.Main.GenRelations

*Generate initial relations for points-to analysis.*
  - Does initial pass to discover call graph.

**Options:**
- `fly`: dump on-the-fly call graph info
- `cs`: dump context-sensitive info
- `ssa`: dump SSA representation
- `partial`: no call graph discovery
- `Dpa.dumppath=`: where to save files
- `Dpa.icallgraph=`: location of initial call graph
- `Dpa.dumpdotgraph`: dump call graph in dot
Format of a Datalog file

Domains

\[
\text{Name Size (map file)}
\]

\[
\begin{align*}
V & : 65536 & \text{var.map} \\
H & : 32768
\end{align*}
\]

Relations

\[
\text{Name (<attribute list>) flags}
\]

\[
\begin{align*}
\text{Store} & (v1 : V, f : F, v2 : V) & \text{input} \\
\text{PointsTo} & (v : V, h : H) & \text{input, output}
\end{align*}
\]

Rules

\[
\text{Head :- Body.}
\]

\[
\text{PointsTo(v1,h) :- Assign(v1,v), PointsTo(v,h).}
\]
Demo
Program Analysis in Datalog

Context Sensitivity
Existing Solution

Call strings based
Cloning-Based Solution

Simple brute force technique.
- Clone every path through the call graph.
- Run context-insensitive algorithm on expanded call graph.

The catch: exponential blowup
Cloning is exponential!
Actually, cloning is unbounded in the presence of recursive cycles.

Technique: We treat all methods within a strongly-connected component as a single node.
Recursion
Top 20 Sourceforge Java Apps

Number of Clones

Size of program (variable nodes)

Number of clones

- $10^0$
- $10^4$
- $10^8$
- $10^{12}$
- $10^{16}$
Cloning is infeasible (?)

- Typical large program has \( \sim 10^{14} \) paths.
- If you need 1 byte to represent a clone:
  - Would require 256 terabytes of storage
  - \( > 12 \) times size of Library of Congress
  - Registered ECC 1GB DIMMs: $41.7 million
    - Power: 96.4 kilowatts = Power for 128 homes
  - 500 GB hard disks: 564 x $195 = $109,980
    - Time to read sequential: 70.8 days
Key Insight

- There are many similarities across contexts.
  - Many copies of nearly-identical results.
- BDDs can represent large sets of redundant data efficiently.
  - Need a BDD encoding that exploits the similarities.
Context-sensitive Pointer Analysis

Algorithm

1. First, do context-insensitive pointer analysis to get call graph.
2. Number clones.
3. Do context-insensitive algorithm on the cloned graph.

- Results explicitly generated for every clone.
- Individual results retrievable with Datalog query.
Counting rule

Set the counts of contexts created, \( c \), to 0.
For each incoming edge,
  If the predecessor of the edge \( p \) has \( k \) contexts,
  create \( k \) clones of node \( n \),
Add tuple \((i, p, i + c, n)\) to \( IE_c\), for \( 0 \leq i \leq k-1 \)
\( c = c + k \).
Expanded Call Graph
Numbering Clones
vP_0(v, h) means there is an invocation site $h$ that assigns a newly allocated object to variable $v$. 

\[
vP_c(c, v, h) := vP_0(v, h), IE_c(c, h, _, _).
\]

\[
vP_c(c_1, v_1, h) := \text{assign}_c(c_1, v_1, c_2, v_2), vP_c(c_2, v_2, h).
\]

\[
hP(h_1, f, h_2) := \text{store}(v_1, f, v_2), vP_c(c, v_1, h_1), vP_c(c, v_2, h_2).
\]

\[
vP_c(c, v_2, h_2) := \text{load}(v_1, f, v_2), vP_c(c, v_1, h_1), hP(h_1, f, h_2),
\]

\[
\text{assign}_c(c_1, v_1, c_2, v_2) := IE_c(c_2, i, c_1, m), \text{formal}(m, z, v_1), \text{actual}(i, z, v_2).
\]