Announcements

- Quiz 1-2, HW1 now in Rainbow Grades
  - If you got points off, come by to get your paper
- HW2 due today
- HW3: XTA Analysis
  - Will post the official assignment today
  - Submitty HW3 soon, today or tomorrow

Outline of Today’s Class

- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity
- The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

Reading

- Dragon book, Chapter 12.1-3 Dragon book
- Thomas Reps, Susan Horwitz and Mooly Sagiv, “Precise, Interprocedural Dataflow Analysis via Graph Reachability, POPL’95

Context-Sensitive Analysis In Practice

- Transfer functions are not distributive
- Local variables, flow of values from actual arguments to formal parameters, and from return to left-hand-side
- Procedures have side effects!
- Sometimes there is no call graph!
  - Function pointers, virtual calls, functions as first-class values and higher-order functions
- Parameter passing mechanisms

Context-Sensitive Analysis In Practice

- Context-sensitive analysis in practice: ad-hoc variants of Sharir and Pnueli’s call string and functional approaches
- Call string approach
  - More intuitive than functional approach
  - Virtually universally applicable, widely used
- Functional approach
  - Better approach, whenever applicable
  - More difficult to implement
  - Better precision and better scalability, in general
Call String-Based Context Sensitivity

- **Calling context** is defined as the content of the entire stack
- Call-string-based context-sensitivity uses a call string as abstraction of the stack
- k-CFA: distinguishes context by k most recent call sites that lead to p
  - make a “copy” of procedure p for each static call string of length k
- 1-CFA: “inline” p at each call site of p

Example: 1-CFA

1. a = &x
2. p_c1 = a  
call id_c1
3. return id_c1 
b = ret_c1
4. z = ‘b’ + ‘b’  
c = &y
5. p_c2 = c  
call id_c2
6. return id_c2 
d = ret_c2
7. entry id_c1
8. ret_c1 = p_c1
9. exit id_c1
10. entry id_c2
11. ret_c2 = p_c2
12. exit p_c2

Problems?

- Recursion renders infinite call strings
- 2-CFA and 3-CFA are popular length strings
- One (common) approach with recursion
  - Collapse strongly connected components in call graph into one big blob. Treat blob as single procedure, context-insensitively
  - Analyze acyclic call graph with full call string
  - Exponential growth, even without recursion
  - Efficient data structures (Binary Decision Diagrams) can make full call string practical

Strongly-Connected Components

- p forms a SCC.
  - Treat calls within p context-insensitively, thus concluding that a*b is not available at 4;
  - Analyze p per c1, other calling contexts if any

Cloning-based Context Sensitivity

- Remember, calling context is the content of the entire stack
- Cloning-based context sensitivity uses program state of interest as abstraction of the stack
- Clone (i.e., copy) a procedure for each program state of interest, i.e., “calling context”
- A hybrid of functional and call-string
Aside: Points-to Analysis for Java

- Similar to points-to analysis for C, but easier
- Context-insensitive, flow-insensitive analysis
- Syntax
  - Object allocation: \( a_i: x = \text{new} A \ // o_i \)
  - Assignment: \( x = y \)
  - Field Write: \( x.f = y \)
  - Field Read: \( x = y.f \)
  - Virtual call: \( c_i: x = y.m(z) \)

Next, define the analysis semantics
- Transfer functions (constraints) over syntax
  - E.g., Allocation \( x = \text{new} A \ // o_i \)
    for each reachable method \( R \)
    for each Allocation site \( x = \text{new} A \ // o_i \)
    \( \{ o_i \} \subseteq \text{pts}(x) \)
  - Note: \( \text{pts}(x) \) denotes the points-to set of \( x \)
- Natural progression: RTA => XTA => Points-to (PTA)

Aside: Points-to Analysis for Java

Transfer Functions (Constraints)

\( a_i: x = \text{new} A \ // o_i \)
\( x = y \)
\( x.f = y \)
\( x = y.f \)
\( c_i: x = y.m(z) \)

for each \( o \) in \( \text{pts}(y) \)
for each \( o \) in \( \text{pts}(o.f) \)
for each \( o \) in \( \text{pts}(y) \)
for each \( o \) in \( \text{pts}(o.f) \)

let \( m'(\text{this},p,ret) = \text{resolve}(o,m) \) in
\( \{ o \} \subseteq \text{pts}(\text{this}) \)
\( \text{pts}(z) \subseteq \text{pts}(p) \)
\( \text{pts}(ret) \subseteq \text{pts}(x) \)

Aside: Points-to Analysis for Java

Example

\( A \ a = \text{new} A(); // o1 \)
\( X x = \text{new} X(); // o2 \)
\( c1: a.set(x); \)
\( A a2 = \text{new} B(); // o3 \)
\( X x2 = \text{new} Y(); // o4 \)
\( c2: a2.set(x2); \)

\( // \text{set}(X p) \{ \text{this}.f = p; \} \)

Cloning-Based Context Sensitivity

- It is more effective if we "cloned" method \( \text{set} \) per receiver object rather than per call site

\( A \ a = \text{new} A(); // o1 \)
\( c1: a.set_{-o1}(\text{new} X()); // o2 \)
\( c2: a.set_{-o1}(\text{new} X()); // o3 \)
\( A a2 = \text{new} A(); // o4 \)
\( c3: a2.set_{-o4}(\text{new} Y()); // o5 \)

- This is flow-insensitive and context-sensitive
Cloning-Based Context Sensitivity

class A { <init>(X p) { this.f = p; } } ...
class B extends A { <init>(X p) { c1: super(p); } }
Note: super calls A.<init>(p)
class C extends B { <init>(X p) { c2: super(p); } }

1. CFA?
c = new C; //o1
2. CFA?
c3: c.<init>(new X()); //o2
3. CFA?
c2 = new C; //o3
4. CFA?
c4: c2.<init>(new X()); //o4

Summary-based Context Sensitivity

Compute summary transfer functions
- \( x = \text{id}(y) \) applies “add edge \( x \rightarrow a \) for each \( y \rightarrow a \)” in the points-to for C example
- \( p() \) applies the “identity function” in Sharir and Pnueli’s Available expressions example
- \( a.\text{set}(x) \) “sets field \( f \) of all objects \( a \) points to to point to the objects \( x \) points to” in the Java example

Phase 1: compute summary transfer functions
- Collapse into SCC on call graph, then compute summaries bottom up
Phase 2: propagate values into callees

Strongly-Connected Components

\( p \) forms a SCC.
- Compute summary of \( p \)
- Summary of \( p \) says \( a^*b \) is NOT available

Outline

Context-sensitive analysis in practice
- Call-string-based context sensitivity
- Cloning-based context sensitivity
- Summary-based context sensitivity

The IFDS framework
- Efficient and precise summary-based analysis
- CFL-reachability

IFDS Context Sensitivity

Interprocedural, Finite, Distributive, Subset (IFDS) problems
- Allows for efficient computation of summary transfer functions. Converts problem into Context-Free-Language (CFL)-Reachability
- Can reduce monotone problem into the IFDS problem, but with loss of precisions
Reading: Thomas Reps, Susan Horwitz and Mooly Sagiv, “Precise, Interprocedural Dataflow Analysis via Graph Reachability, POPL’95
Efficient Encoding of Transfer Functions

- Finite set of dataflow facts $D$
- E.g., all variables $\{x, y, z\}$
- Transfer functions $f: 2^D \to 2^D$
- Edge $\Lambda \to d$ means $d \in f(\emptyset)$
- I.e., $d$ is generated
- Edge $d_1 \to d_2$ means $d_2 \notin f(\emptyset)$ and $d_2 \in f(S)$ if $d_1 \in S$
- I.e., $d_1$ in $S$ leads to $d_2$ in $f(S)$
- Edge $\Lambda \to \Lambda$ always there

E.g., $f$ at $j: x = y + z$

Taint Analysis

What Can Be Encoded.
- 1. $z = 5$
- 2. $y = \text{“tainted” value}$
- 3. $x = y + z$

The paths from top $\Lambda$ to $x$ and $y$ entail that $x$ and $y$ are tainted at exit from 3.

Efficient Computation of Function Composition!

What Can Be Encoded.
- All Bit-Vector Problems!
  - 1. $x = a*b$
  - 2. $a = a - 1$

  - Add edges from $\Lambda$ to facts being generated (e.g., $a*b$)
  - Add in-out edges to facts being preserved (e.g., $a+b$)

  - $f_1:\Lambda \to x+y$, $a*b$, $a+b$

Big Picture, Why Does It Matter

- We can compose transfer functions within a procedure $p$ and compute the summary transfer function $\Phi_p$!

- Precisely: Computes the MORP solution!

- Efficiently: $O(ED^3)$
  - $E$ is the number of intraprocedural edges across all procedures in ICFG

What Cannot Be Encoded

- Monotone functions cannot be encoded
  - E.g., constant propagation, points-to analysis

- Points-to analysis, distributive subset?
  - $f_{\text{pt}}: p \to x$ in $f_{\text{pt}}(S)$ if $q \to y$ in $S$ AND $y \to \Lambda$ in $S$
  - Can encode disjunctions but not conjunctions

- Large class of problems falls under IFDS

- Monotone problems can be reduced into IFDS with loss of precision
Exploded Supergraph G#

- Let $G^*$ be the ICFG, which they call the supergraph.
- First, define the nodes of $G^*$.
- For each node $j \in G^*$ there is node $<j, \Lambda> \in G^*$.
- For each node $j \in G^*$ and $d \in \Lambda$ there is node $<j, d> \in G^*$.

\[
\Lambda \quad d_1 \quad d_2 \quad d_3
\]

- Represents the $\text{in}(j)$.

Exploded Supergraph G#

- Next, add edges to $G^*$.
- For each $k$ in successors of $j$.
- Add edge $<j, \Lambda> \rightarrow <k, \Lambda>$ to $G^*$.
- Add edge $<j, \Lambda> \rightarrow <k, d>$ if $d \in f_j(\emptyset)$.
- Add edge $<d_1, p> \rightarrow <d_2, k>$ if $d_2 \in f_j(\emptyset)$ and $d_2 \in f_j(\text{in}(j))$ if $d_1 \in \text{in}(j)$.

\[
\text{in}(j): \quad o \quad o \quad o \\
\text{in}(k): \quad o \quad o \quad o
\]

- Represent (encode) transfer function $f_j$.

Exploded Supergraph

- One can think about IFDS in terms of Sharir and Pnueli's functional approach.
- ...or in terms of graph reachability: IFDS reduces standard dataflow problem to a reachability problem in $G^*$.
- Path from $<1, \Lambda>$ to $<j, d>$ means that $d$ reaches $j$.
- More precisely, it is a CFL-reachability problem: "Is there a path from $<1, \Lambda>$ to $<j, d>$ whose edges form a string in the language of balanced parentheses?"
- Gives rise to on-demand approaches.

IFDS Conclusion

- Key idea is encoding of transfer functions $f_j$.
- A large class of distributive functions.
- Allows for efficient computation of summary transfer functions $\Phi_p$.
- Reduces problem to CFL-reachability problem.
- Gives rise to on-demand approaches.
- Can use on non-distributive problems but with loss of precision.
- Real-world analysis problems.
- Soot has a built-in IFDS framework.
- Some taint analyses for Android use IFDS.
**IFDS Conclusion**

- My conjecture: framing a monotone analysis in terms of IFDS (with loss of precision) gives better result than using SCC-based summary-based analysis
  - Recursion entails very large SCC’s and loss of precision
- IFDS is defined for **may**-problems. **Must**-problems can be expressed as complement

**Next class**

- Abstract Interpretation (AI)
- After Abstract Interpretation, we will be moving to the functional languages world