Context Sensitivity, IFDS and CFL-reachability

Announcements

- HW2 due Tuesday
- Will go over Soot and framework
- Please ask questions!

Outline of Today’s Class

- Sharir and Pnueli results
  - Functional approach to Interprocedural Analysis
  - Call-string approach to Interprocedural Analysis
- 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

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

Classic Results and Ideas

- Sharir and Pnueli’s “Two approaches to Interprocedural dataflow analysis”, 1981
  - Amir Pnueli, Turing Award in 1996 “For seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification.”
- A finite lattice of dataflow facts
- Distributive transfer functions
- No local variables and no parameter passing

Sharir and Pnueli Example (Available Expressions)

1. read a, b
   t = a * b
2. call p
3. return p
4. t = a * b
   print t
5. entry p
6. if a == 0 then
   a = a - 1
7. call p
8. return p
9. exit p
10. If a == 0 then
    - a = a - 1
11. Else
    - a = a * b
12. Print t
Functional Approach to Interprocedural Dataflow Analysis

- Operates on unchanged property space
- Computes summary transfer functions $\Phi_p$ that summarize effect of procedures $p$
- Reduces problem to intraprocedural case:
  - $in(return \ p) = \Phi_p(in(call \ p))$
  - thus, avoids propagation from callee along the exit $p$ -- $return$ $p$ edge!

Functional Approach

Phase 1:
- Compute a summary transfer function $\Phi_p$ that captures effect of $p$.
  - In our example $\Phi_p$ is the identity function: nothing gets generated and nothing gets killed.

Phase 2:
- Dataflow analysis:
  - At return $p$
    - $in(return \ p) = \Phi_p(in(call \ p))$
    - $out(return \ p) = in(return \ p)$
    - avoids propagation along exit-return edges!
  - At entry $p$
    - $in(entry \ p) = V \ in(call \ p)$
      (propagates facts from all callers to callee)

Computing Summary Transfer Functions

- For certain lattices and function spaces, we can compute summary transfer functions $\Phi$.
- The IFDS framework we discuss today.
- In general, not clear how to compute $\Phi$'s efficiently
- Ad-hoc approaches/approximation when computing $\Phi$'s for specific monotone function spaces (points-to analysis, taint analysis)

Call String Approach to Interprocedural Dataflow Analysis

- A call string records outstanding calls in path
  - E.g., call string $c_1$ denotes that "we got there" on a path with outstanding calls at $c_1$ and at $c_2$
  - Meaning: $a*b$ is available at 6 on paths with outstanding call string $c_1$, but it is not available on paths with outstanding call string $c_1, c_2$

Call String Approach

- Tags solutions per program point with corresponding call string
- Multiple tagged solutions per program point $j$ in $p$:
  - Sharir and Pnueli Example:
    - We have $\langle \{ a*b \}, c_1, c_2 \rangle, \langle \{ \}, c_1, c_2 \rangle$ at 6
    - Meaning: $a*b$ is available at 6 on paths with outstanding call string $c_1$, but it is not available on paths with outstanding call string $c_1, c_2$
Call String Approach
- Apply original transfer functions point-wise
- Apply on original dataflow lattice elements
  - \( \{ a \cdot b \} \), \( \{ a \cdot b, a+b \} \), \( \{ \} \), etc.

At exit nodes, propagate only matching call strings
\( \langle \text{ret} \rightarrow x \rangle \), \( \langle \text{ret} \rightarrow y \rangle \), \( \langle \text{ret} \rightarrow z \rangle \) at 9
Propagate \( \langle \text{ret} \rightarrow y \rangle \) to 6, thus, \( \{ d \rightarrow y \} \), because matches call string \( \langle c_2 \rangle \)

What is \( S_{CS}(8) \)?
Union of \( \langle p \rightarrow x, \langle c_1 \rangle \rangle \) and
\( \langle p \rightarrow y, \langle c_2 \rangle \rangle \) so \( S_{CS}(8) \) is graph \( \{ p \rightarrow x, p \rightarrow y \} \)

What is \( S_{CS}(4) \)?
What is \( S_{CS}(6) \)?
(out(6) more precisely)
Sharir and Pnueli, Key Result

- $S_{FA}(j)$ is the solution at $j$ computed by the functional approach
- $S_{CS}(j)$ is the solution at $j$ computed by the call string approach

For distributive functions and finite lattices

$$S_{FA}(j) = S_{CS}(j) = MORP(j)$$

Caveats?

For distributive functions and finite lattices, $S_{FA}$ and $S_{CS}$ cannot be computed (efficiently)

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

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, i.e., calling context
- $k$-CFA: make a “copy” of procedure $p$ for each static call string of length $k$ --- string captures $k$ most recent call sites that lead to $p$
- 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 $id_{c1}$
4. $z = *b + *b$
call $id_c$
5. $p_{c2} = c$
call $id_{c2}$
6. return $id_{c2}$
7. entry $id_{c1}$
8. ret$_{c1} = p_{c1}$
9. exit $p_{c1}$
10. entry $id_{c2}$
11. ret$_{c2} = p_{c2}$
12. exit $p_{c2}$
Problems?

```
main:      id:
...       ...
  a = &x;   int* id(int* p) {
  c1: b = id(a);  c3: return id_impl(p);
  z = *b + *b;
  c = &y;
  c2: d = id(c);  int* id_impl(int p) {
  ...            return p;
  }
```

Problems?

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

Strongly-Connected Components

```
1. read a, b
2. call p
3. return p
4. t = a*b
5. entry p
6. if a == 0 then
   7. call p
   8. return p
   9. exit p
```

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 state of interest, i.e., "calling context"
- A hybrid of functional and call-string approaches

Cloning-based Context Sensitivity

```
A a = new A(); // o1
  c1: a.set(new X()); // o2
  c2: a.set(new X()); // o3

A a2 = new B(); // o4
  c2: a2.set(new Y()); // o5

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

Cloning-based Context Sensitivity

- It is more effective if we “cloned” method set per receiver object rather than per call site

```
A a = new A(); // o1
  c1: a.set_o1(new X()); // o2
  c2: a.set_o1(new X()); // o3

A a2 = new A(); // o4
  c2: a2.set_o4(new Y()); // o5
```
Summary-based Context Sensitivity

- Compute summary transfer functions
  - \( x = \text{id}(y) \) applies “add edge \( x \rightarrow a \) for each \( y \rightarrow a \)” for the points-to example
  - \( \text{p()} \) applies the “identity function” for Sharir and Pnueli’s Available expressions 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

1. read \( a, b \)
2. \( t = a \cdot b \)
3. call \( \text{p}() \)
4. \( t = a \cdot b \)
5. entry \( \text{p}() \)
6. if \( a == 0 \) then
   - \( a = a - 1 \)
7. call \( \text{p}() \)
8. return \( \text{p}() \)
9. return \( t = a \cdot b \)

Outline

- Sharir and Pnueli results
  - Functional approach to Interprocedural Analysis
  - Call-string approach to Interprocedural Analysis
- 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

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 \( f: 2^D \rightarrow 2^D \)
  - Edge \( \Lambda \rightarrow d \) means \( d \in f(\Lambda) \)
    - I.e., \( d \) is generated
  - Edge \( d_1 \rightarrow d_2 \) means \( d_2 \notin f(d_1) \) 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 \( \Lambda \rightarrow \Lambda \) always there

Efficient Encoding of Transfer Functions

E.g., \( f(j): x = y + z \)

Taint Analysis
What Can Be Encoded. Taint Analysis

1. \( z = 5 \)
2. \( y = \) “tainted” value
3. \( x = y + z \)

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

What Cannot Be Encoded

- Monotone functions cannot be encoded
  - E.g., constant propagation, points-to analysis
- Points-to analysis, distributive subset?
- We can represent disjunctions but not conjunctions
- Large class of problems falls under IFDS
- Monotone problems can be reduced into IFDS with loss of precision

Big Picture, Why Does It Matter

- We can compose transfer functions within a procedure \( p \) and compute the summary transfer function \( \Phi_p \).

Efficient Computation of Function Composition!

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 \( \langle j, \Lambda \rangle \in G\# \)
- For each node \( j \in G^* \) and \( d \in D \) there is node \( \langle j, d \rangle \in G\# \)
- 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 $d \in f(j)$
    - Add edge $<d1,j> \rightarrow <d2,k>$ if $d2 \notin f(j)$ and $d2 \in f(j)$ if $d1 \in \text{in}(j)$
- Represents transfer function $f_j$

IFDS Conclusion

- Key idea is encoding of transfer functions $f_j$
  - A large class of distributed functions
  - Allows for efficient computation of summary transfer functions $\Phi_p$
  - Reduces problem to CFL-reachability problem
  - Can use on non-distributive problems but with loss of precision
  - Can use on real-world analysis problems
    - Soot has a built-in IFDS framework
    - Some taint analyses for Android use (essentially) IFDS

Exploded Supergraph

1. read $a, b$
2. call $p$
3. return $p$
4. print $t$
5. entry $p$
6. if $a == 0$ then
    7. $a = a - 1$
6. $t = 5$
8. evil $p$

IFDS Conclusion

- Key idea is encoding of transfer functions $f_j$
- A large class of distributed functions
- Allows for efficient computation of summary transfer functions $\Phi_p$
- Reduces problem to CFL-reachability problem
- Can use on non-distributive problems but with loss of precision
- Can use on real-world analysis problems
  - Soot has a built-in IFDS framework
  - Some taint analyses for Android use (essentially) IFDS