Static Program Analysis
Part 7 – interprocedural analysis

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University
Interprocedural analysis

• Analyzing the body of a single function:
  – *intra*procedural analysis

• Analyzing the whole program with function calls:
  – *inter*procedural analysis

• For now, we consider TIP without function pointers and indirect calls

• A naive approach:
  – analyze each function in isolation
  – be maximally pessimistic about results of function calls
  – rarely sufficient precision...
CFG for whole programs

The idea:
• construct a CFG for each function
• then glue them together to reflect function calls and returns

We need to take care of:
• parameter passing
• return values
• values of local variables across calls (including recursive functions, so not enough to assume unique variable names)
A simplifying assumption

• Assume that all function calls are of the form

\[ X = f(E_1, \ldots, E_n); \]

• This can always be obtained by normalization
Interprocedural CFGs (1/3)

Split each original call node into two nodes:

\[ X = f(E_1, \ldots, E_n) \]
\[ \square = f(E_1, \ldots, E_n) \]
\[ X = \square \]

A special edge that connects the call node with its after-call node.

the “call node”
the “after-call node”
Interprocedural CFGs (2/3)

Change each return node

\[
\text{return } E
\]

into an assignment:

\[
\text{result } = E
\]

(where \text{result} is a fresh variable)
Add call edges and return edges:

\[
\text{function } g(a_1, \ldots, a_m)
\]

\[
X = f(E_1, \ldots, E_n)
\]

\[
\text{function } f(b_1, \ldots, b_n)
\]

\[
\text{result } = E
\]
Constraints

• For call/entry nodes:
  – be careful to model evaluation of all the actual parameters before binding them to the formal parameter names (otherwise, it may fail for recursive functions)

• For after-call/exit nodes:
  – like an assignment: \( X = \text{result} \)
  – but also restore local variables from before the call using the call\(\sim\)after-call edge

• The details depend on the specific analysis...
Example: interprocedural sign analysis

- Recall the intraprocedural sign analysis...
- Lattice for abstract values:

```
   ┌──┐
   │ T│
   └──┘
    │   
    └──┘
      │
      └──┘
Sign = + - 0

- Lattice for abstract states:

\[ Vars \rightarrow \text{Sign} \]
Example: interprocedural sign analysis

• Constraint for entry node v of function \( f(b_1, \ldots, b_n) \):

\[
\llbracket v \rrbracket = \bigsqcup \bot \llbracket b_1 \mapsto \text{eval}(\llbracket w \rrbracket, E_1^w), \ldots, b_n \mapsto \text{eval}(\llbracket w \rrbracket, E_n^w) \rrbracket
\]

where \( E_i^w \) is i’th argument at \( w \)

• Constraint for after-call node v labeled \( X = \square \), with call node \( v' \):

\[
\llbracket v \rrbracket = \llbracket v' \rrbracket [X \mapsto \llbracket w \rrbracket(\text{result})]
\]

where \( w \in \text{pred}(v) \)

(Recall: no global variables, no heap, and no higher-order functions)
1)  \( [v] = t_v(\bigcup_{w \in \text{pred}(v)} [w]) \)

2)  \( \forall w \in \text{succ}(v): t_v([v]) \subseteq [w] \)

– recall “solving inequations”
– may require fewer join operations
  if there are many CFG edges
– more suitable for interprocedural flow
The worklist algorithm (original version)

\[
x_1 = \perp; \ldots \; x_n = \perp
\]
\[
W = \{v_1, \ldots, v_n\}
\]
\[
\text{while } (W \neq \emptyset) \{ \\
\quad v_i = W.\text{removeNext}() \\
\quad y = f_i(x_1, \ldots, x_n) \\
\quad \text{if } (y \neq x_i) \{ \\
\quad\quad \text{for } (v_j \in \text{dep}(v_i)) \{ \\
\quad\quad\quad W.\text{add}(v_j) \\
\quad\quad\} \\
\quad\quad\} \\
\quad x_i = y \\
\quad\} \\
\}
\]
The worklist algorithm
(alternative version)

\[ x_1 = \perp; \ldots x_n = \perp \]
\[ W = \{v_1, \ldots, v_n\} \]

\[
\text{while } (W \neq \emptyset) \{ \\
\quad v_i = W.\text{removeNext}() \\\n\quad y = t_i(x_i) \\\n\quad \text{for } (v_j \in \text{dep}(v_i)) \{ \\
\quad\quad \text{propagate}(y, v_j) \\\n\quad\} \\\n\text{\}}
\]

\[
\text{propagate}(y, v_j) \{ \\
\quad z = x_j \cup y \\\n\quad \text{if } (z \neq x_j) \{ \\
\quad\quad x_j = z \\\n\quad\quad W.\text{add}(v_j) \\\n\quad\} \\
\}
\]

Implementation: worklistFixpointPropagationSolver
Agenda

• Interprocedural analysis
• Context-sensitive interprocedural analysis
Interprocedurally invalid paths
Example

What is the sign of the return value of \( g \)?

\[
\begin{align*}
\text{f(z)} & \{ \\
& \text{return } z*42; \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{g()} & \{ \\
& \text{var } x,y; \\
& x = f(0); \\
& y = f(87); \\
& \text{return } x + y; \\
& \}
\end{align*}
\]

Our current analysis says “T”
Function cloning
(alternatively, function inlining)

- Clone functions such that each function has only one callee

- Can avoid interprocedurally invalid paths 😊
- For high nesting depths, gives exponential blow-up 😞
- Doesn’t work on (mutually) recursive functions 😞

- Use heuristics to determine when to apply (trade-off between CFG size and precision)
Example, with cloning

What is the sign of the return value of \( g \)?

\[
f_1(z_1) \{
    \text{return } z_1 \times 42;
\}
\]

\[
f_2(z_2) \{
    \text{return } z_2 \times 42;
\}
\]

\[
g() \{
    \text{var } x,y;
    x = f_1(0);
    y = f_2(87);
    \text{return } x + y;
\}
\]
Context sensitive analysis

• Function cloning provides a kind of context sensitivity (also called polyvariant analysis)

• Instead of physically copying the function CFGs, do it *logically*

• Replace the lattice for abstract states, States, by

  \[\text{Contexts} \rightarrow \text{lift(States)}\]

where Contexts is a set of *call contexts*
  – the contexts are abstractions of the state at function entry
  – Contexts must be finite to ensure finite height of the lattice
  – the bottom element of lift(States) represents “unreachable” contexts

• Different strategies for choosing the set Contexts...
One-level cloning

• Let $c_1, \ldots, c_n$ be the call nodes in the program
• Define $\text{Contexts} = \{c_1, \ldots, c_n\} \cup \{\epsilon\}$
  – each call node now defines its own “call context”
    (using $\epsilon$ to represent the call context at the main function)
  – the context is then like the return address of the top-most stack frame in the call stack
• Same effect as one-level cloning, but without actually copying the function CFGs
• Usually straightforward to generalize the constraints for a context insensitive analysis to this lattice
• (Example: context-sensitive sign analysis – later...)
The call string approach

• Let $c_1,\ldots,c_n$ be the call nodes in the program
• Define Contexts as the set of strings over $\{c_1,\ldots,c_n\}$ of length $\leq k$
  – such a string represents the top-most $k$ call locations on the call stack
  – the empty string $\varepsilon$ again represents the call context at the main function
• For $k=1$ this amounts to one-level cloning

Implementation: CallStringSignAnalysis
Example: interprocedural sign analysis with call strings (k=1)

Lattice for abstract states: Contexts → lift(Vars → Sign)
where Contexts=\{\varepsilon,c_1,c_2\}

f(z) {
  var t1, t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}

...[\varepsilon \mapsto \text{unreachable},
c1 \mapsto \bot [z\mapsto 0, t1\mapsto 0, t2\mapsto 0],
c2 \mapsto \bot [z\mapsto +, t1\mapsto +, t2\mapsto +]]

...x = f(0);  // c1
y = f(87);  // c2
...
Context sensitivity with call strings
function entry nodes, for k=1

Constraint for entry node $v$ of function $f(b_1, \ldots, b_n)$:
(if not ‘main’)

$$\llbracket v \rrbracket(c) = \bigcup_{w \in \text{pred}(v)} s_w^{c'}$$

$$w \in \text{pred}(v) \land c = w \land c' \in \text{Contexts}$$

$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } \llbracket w \rrbracket(c') = \text{unreachable} \\ \bot \left[ b_1 \rightarrow \text{eval}(\llbracket w \rrbracket(c'), E_1^w), \ldots, b_n \rightarrow \text{eval}(\llbracket w \rrbracket(c'), E_n^w) \right] & \text{otherwise} \end{cases}$$
Context sensitivity with call strings
after-call nodes, for k=1

Constraint for after-call node $v$ labeled $X = \square$, with call node $v'$ and exit node $w \in \text{pred}(v)$:

$$\llbracket v \rrbracket(c) = \begin{cases} \text{unreachable} & \text{if } \llbracket v' \rrbracket(c) = \text{unreachable} \\ \llbracket v' \rrbracket(c)[X \mapsto \llbracket w \rrbracket(v')(\text{result})] & \text{otherwise} \end{cases}$$
The functional approach

• The call string approach considers *control flow*
  – but why distinguish between two different call sites if their abstract states are the same?
• The functional approach instead considers *data*
• In the most general form, choose
  Contexts = States
  (requires States to be finite)
• Each element of the lattice \( \text{States} \rightarrow \text{lift(States)} \)
  is now a map \( m \) that provides an element \( m(x) \) from States (or “unreachable”) for each possible \( x \) where \( x \) describes the state at function entry
Example: Interprocedural sign analysis with the functional approach

Lattice for abstract states: Contexts → lift(Vars → Sign)
where Contexts = Vars → Sign

```haskell
f(z) {  
  var t1, t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
```

```haskell
x = f(0);
y = f(87);
```

Lattice:  
- \( \bot \) \( \mapsto \) \( 0 \) \( 0 \) \( \bot \) \( 0, 0, 0 \) \( \bot \) \( +, +, + \) unreachable
The functional approach

- The lattice element for a function exit node is thus a **function summary** that maps abstract function input to abstract function output
- This can be exploited at call nodes!
- When entering a function with abstract state \( x \):
  - consider the function summary \( s \) for that function
  - if \( s(x) \) already has been computed, use that to model the entire function body, then proceed directly to the after-call node
- Avoids the problem with interprocedurally invalid paths!
- ...but may be expensive if States is large

Implementation: FunctionalSignAnalysis
Context sensitivity with the functional approach function entry nodes

Constraint for entry node v of function f(b₁, ..., bₙ):
(if not ‘main’)

\[
\llbracket v \rrbracket(c) = \bigsqcup_{w \in \text{pred}(v)} s_w^{c'}
\]

\[
w = s_w^{c'} \land c = s_w^{c'} \land c' \in \text{Contexts}
\]

where \( s_w^{c'} \) is defined as before
Context sensitivity with the functional approach after-call nodes

Constraint for after-call node $v$ labeled $X = □$, with call node $v'$ and exit node $w ∈ \text{pred}(v)$:

$$\llbracket v \rrbracket(c) = \begin{cases} \text{unreachable} & \text{if } \llbracket v' \rrbracket(c) = \text{unreachable } \land \llbracket w \rrbracket(s^c_{v'}) = \text{unreachable} \\ \llbracket v' \rrbracket(c)[X → \llbracket w \rrbracket(s^c_{v'})(\text{result})] & \text{otherwise} \end{cases}$$