Interprocedural Data Flow Analysis

Static Program Analysis

Christian Hammer
18. Juni 2014
Interprocedural Reaching Definitions

Global Variables

(1) int a, b, c;

(3) void q () {
(4)   int z=1;
(5)   a=2;
(6)   b=3;
(7)   p(4, z);
(8)   z=a;
(9)   c=5;
(10)  p(6, c);
(11)  }

(12) void p(int x,int &y) {
(13)   static int a = 6;
(14)   a=c;
(15)   if(x) {
(16)     d=7;
(17)     p(8, x);
(18)   } else {
(19)     b=9;
(20)   }
(21)   y =0;
(22)   }

call-by-value

call-by-reference
6.2 Interprocedural Realizable Paths

In the intraprocedural case all paths in the CFG were assumed to be executable and therefore realizable. In the interprocedural case this is more complicated: The individual procedures of a program are represented in control flow graphs $G_p = (N_p, E_p, n_{s_p}, n_{e_p})$ for each procedure $p$. An interprocedural control flow graph (ICFG) is a directed graph $G = (N?, E?, n_{s?}, n_{e?})$, where $N? = \bigcup_{p \in P} N_p$ and $E? = E_C[\bigcup_{p \in P} E_p]$. One procedure $q$ is the program's main procedure, its START and EXIT nodes are the main START and EXIT nodes: $n_{s?} = n_{s_q}$ and $n_{e?} = n_{e_q}$.

The calls are represented by call and return edges in $E_C$: A call edge $e_2 \in E_C$ is going from a call node $n_2 \in N_p$ to the START node $n_{s_q}$ of the called procedure $q$. A return edge $e_2 \in E_C$ is going from the EXIT node $n_{e_q}$ of the called procedure $q$ back to the immediate successor of the call node $n_2$.

Example 6.2: Figure 6.2 shows the ICFG for the reaching definition example. Note that there are control flow edges between call nodes and their immediate successors.

If any path through the ICFG is assumed to be a realizable path, data flow analysis will become imprecise, as clearly unrealizable paths can be traversed: Consider the definition of global variable $c$ in line/node 9, which reaches the called procedure via the call edge at line/node 10. All paths through $p$ are free of definitions for $c$ and the definition gets propagated along the return edges: via $16$, $17$, and $19$.

There are two common variants: First, the immediate successor of a call node is an explicitly defined return node. Second, the return edge is going from the EXIT node to the call node itself.
Analyzing Interprocedural Programs

- \[ \text{RD}_{\text{IMOP}}(n) = \bigcup_{p=\langle n^0, \ldots, n \rangle} [p](\emptyset) \]
- where \( p \) are inter procedurally realizable paths (impossible in general)
- interprocedural minimal-fixed-point (IMFP) solution is computed
- However, impossible to check for interprocedurally realizable paths

- Procedures can be inlined
  - replace calls by the called procedure
  - resulting program can be analyzed like an intraprocedural one
  - not possible in the presence of recursion
  - even without the size of the inlined programs may grow exponentially
  - not feasible in practice
Analyzing Interprocedural Programs (cont.)

- Compute effects of procedures
  - represented in a transfer function
  - maps flow information at a call site from the call to the return
  - call statements are ordinary statements with transfer functions
  - intraprocedural techniques can be applied

- Explicit encoding of calling context of a procedure
  - procedure is analyzed for each calling context separately
  - in the presence of recursion the set of calling contexts may be infinite
  - depending on the encoding of the calling context
Effect Calculation

- functional approach [SP81]
- maps the data flow information at the entry of a procedure to the information that holds at the exit
- computed function can be used in the transfer functions at the call statements
- intraprocedural data flow analysis can then be used in a second pass
- first pass is a data flow analysis where the data flow information are functions and the transfer functions are function compositions
- For some data flow problems the resulting data flow information is infinite function compositions and therefore not computable
- For a large class of data flow problems these computed functions reduce to simple mappings where the composition can be computed instantly
Context Encoding

- call strings capture the “history” of calls that lead to a node $n$
- abstraction of the call stack
- lattice elements combine calling context and intraprocedural data flow facts
- transfer functions extended to handle the additional calling context
- length of the call strings can be limited to a certain length $k$
- call string longer than $k$ are shortened such that the “oldest” elements are removed first
- overcomes limitations of recursion
- maybe imprecise
Call Strings

- calling context $c \in C$ encoded through data flow facts that hold at the entry to procedure $p \in P$
- data flow facts $c'$ at the exit of the procedure stored in mapping $C \times P \rightarrow C$
- At every call node $n$ of a procedure $p$ the data flow facts $c$ are then bound to data flow facts $c' = \text{bind}(c)$ that hold at the entry node of $p$
- If the effect of $p$ for $c'$ has already been computed, it can be reused from the mapping which contains the data flow facts $c''$ holding at the exit of $p$
- After back-binding the effect to the call site, the effect $c''' = \text{bind}^{-1}(c'')$ holds at the exit of the call node $n$
Let $G = (N^*, E^*, n^s_0, n^e_0)$ be an ICFG. A node $m \in N^*$ is data dependent on node $n \in N^*$, if

- there is an interprocedurally matched path $p$ from $n$ to $m$ in the ICFG,
- there is a variable $v$, with $v \in \text{def}(n)$ and $v \in \text{ref}(m)$, and
- for all nodes $k \neq n$ of path $p$, $v \notin \text{def}(k)$ holds.

At call sites the global variables are modeled as call-by-value-result parameters, which is correct without call-by-reference parameters and aliasing.

- $\text{GMOD}(p)$: the set of all variables that might be modified if procedure $p$ is called.
- $\text{GREF}(p)$: the set of all variables that might be referenced if procedure $p$ is called.
Effect Calculation

- \( \text{bind}^{-1}(S, p) = S - \text{locals}(p) \)
- \( \text{GMOD}(n) = \text{bind}^{-1}(\text{GMOD}(p)) \)
- \( \text{GREF}(n) = \text{bind}^{-1}(\text{GREF}(p)) \)

- \( \text{GMOD}(q) = \text{I MOD}(q) \cup \bigcup_{p \in \text{calls}(q)} \text{bind}^{-1}(\text{GMOD}(p), p) \)
- \( \text{GREF}(q) = \text{I REF}(q) \cup \bigcup_{p \in \text{calls}(q)} \text{bind}^{-1}(\text{GREF}(p), p) \)

- \( \text{def}(n) = \text{GMOD}(n) \)
- \( \text{ref}(n) = \text{GMOD}(n) \cup \text{GREF}(n) \)
Example Interprocedural Data Dependences

Figure 6.3: ICFG with data dependence

Procedure. Therefore control dependence is computed only intraprocedural, where the edges between call nodes and their successors are assumed to be normal control flow edges.

6.4.2 Data Dependence

For representation in the PDGs of the procedures, data dependence is computed only intraprocedural:

Definition 6.4

Let $G = (N, E, n_{s0}, n_{e0})$ be an ICFG. A node $m \in N$ is data dependent on node $n \in N$, if

1. there is an interprocedurally matched path $p$ from $n$ to $m$ in the ICFG,
2. there is a variable $v$, with $v \in \text{def}(n)$ and $v \in \text{ref}(m)$, and
3. for all nodes $k \neq n$ of path $p$, $v \not\in \text{def}(k)$ holds.

The difference to interprocedural data dependence is the restriction on interprocedurally matched paths. This variant of data dependence can be computed with a slightly modified version of interprocedural reaching definitions $\text{RD}_\text{IMFP}$.

Without global variables (and call-by-reference and aliasing) the analysis would be even simpler, as called procedures would have no effects in the calling procedure and the intraprocedural computation of $\text{RD}_\text{MFP}$ would be sufficient. Therefore an approach that eliminates global variables is used, where...