Static Program Analysis

Reaching Definitions

Winter Term 2014/15

Advanced Lecture (9 CP)

Christian Hammer
Scale

(1) while(TRUE) {
(2)   if (((p_ab[CTRL2] & 0x10) == 0) {
(3)     u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
(4)     u_kg = u * kal_kg;
(5)   if (((p_cd[CTRL2] & 0x01) != 0) {
(6)       for (idx=0; idx<7; idx++) {
(7)         e_puf[idx] = p_cd[PA];
(8)       if (((p_cd[CTRL2] & 0x10) != 0) {
(9)         switch (e_puf[idx]) {
(10)        case '+': kal_kg *= 1.1; break;
(11)        case '-': kal_kg *= 0.9; break; }
(12)       e_puf[idx] = '\0';
(13)       printf("Artikel: %7.7s\n", e_puf);
(14)       printf(" %6.2f kg ", u_kg);
(15)     }}}}
(1) while(TRUE) {
(2)   if (((p_ab[CTRL2] & 0x10) == 0) {
(3)     u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
(4)     u_kg = u * kal_kg;
(5)     if ((p_cd[CTRL2] & 0x01) != 0) {
(6)       for (idx=0; idx<7; idx++) {
(7)         e_puf[idx] = p_cd[PA];
(8)       if (((p_cd[CTRL2] & 0x10) != 0) {
(9)           switch(e_puf[idx]) {
(10)          case '+' : kal_kg *= 1.1; break;
(11)          case '-' : kal_kg *= 0.9; break; } }
(12)       e_puf[idx] = '\0'; } }
(13)     printf("Artikel: %07.7s\n", e_puf);
(14)     printf(" %6.2f kg ", u_kg);
(15)   }
(16) }
(17) }

Calibration factor can be changed!
Dataflow Analysis

- Can a value computed at a certain statement flow to another given statement?
- Usually represents program as a directed graph
- Nodes are statements/predicates
- Edges describe the control flow
- Allows analysis of programs with
  - structured and
  - unstructured control flow
  (break, continue, try ... catch ... finally, goto)
Intraprocedural Control Flow Graph

- Control Flow Graph (CFG) $G = (N, E, n_s, n_e, \nu)$
- Set of nodes $N$ (statements and predicates)
- Distinguished nodes $n_s$ and $n_e$
- Set of control flow edges $E$, $(n, m) \in E$
  
  $n \rightarrow_{cf} m$ iff $m$ may execute directly after $n$
- Total attribute function $\nu : E \rightarrow \{\text{true, false, } \varepsilon\} \cup \mathbb{Z}$
  condition under which control flows along an edge
- Functions $\text{succ}$ and $\text{pred}$ that map the successors and predecessors to each node.
- Reachability is undecidable, so conservatively assumed
Intraprocedural Control Flow Graph

(1) read(n);
(2) i = 1;
(3) sum = 0;
(4) prod = 1;
(5) while (i <= n) {
(6) sum = sum + i;
(7) prod = prod * i;
(8) i++;
(9) }
(10) write(sum);
(11) write(prod);
Monotone Dataflow Analysis Framework

- most important class of dataflow analysis
- compute the most precise solution under certain conditions
- only takes a finite number of steps
  - even if loops have infinitely many paths
- Basic data structure: complete lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
Monotone Function Space

- Set of functions $\mathcal{F}$ on a meet semi-lattice $\mathcal{L} = (L, \sqsubseteq, \sqcap, \bot)$
- $\exists id_L \in \mathcal{F}: \forall x \in L: id_L(x) = x$ (identity function)
- $\forall F \in \mathcal{F}: \forall x, y \in L: x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)$ (monotonicity)
- $\forall F, G \in \mathcal{F}: F \circ G \in \mathcal{F}$ (closed under composition)
- $\forall F, G \in \mathcal{F}: F \sqcap G \in \mathcal{F}$ (pointwise infimum)

Monotone Dataflow Analysis Framework

- Consists of a meet semi-lattice $\mathcal{L}$
- And a monotone function space $\mathcal{F}$
- **distributive**, if all functions are distributive over $\sqcap$:
  $\forall F \in \mathcal{F}: \forall x, y \in L: F(x \sqcap y) = F(x) \sqcap F(y)$
Computing Data Flow Analyses

- **Meet-Over-All-Paths (MOP) solution desired**
  
  \[ F(n_1, \ldots, n_k) : L \rightarrow L : F(n_1, \ldots, n_k)(x) = (F_{n_k} \circ F(n_1, \ldots, n_{k-1}))(x) \]

  MOP is \( \prod_{Path} P_i F_{P_i} \)

- Can be computed for 1, 2, 3, 4, 5, 10, 11; 1, 2, 3, 4, 5, 6, 7, 8, 10, 11; 1, 2, 3, 4, 5, 6, 7, 8, 6, 7, 8, 10, 11; etc.

- But when should we stop calculating?

  Analysis time not linear with program size!
Fixed points

- Fixed point of an operator \( f \) on a semi-lattice \( L \) is an element \( x \in L \) such that \( f(x) = x \).
- Maximal fixed point (MFP) \( x \) if \( \forall y \in L: f(y) = y \Rightarrow y \subseteq x \).
Computing the MFP

```
foreach n ∈ CFG do
    A[n] = ⊥
od

do
    change = false
    foreach n ∈ CFG do
        temp = \prod_{q ∈ pred(n)} F_q(A[q])
        if temp ≠ A[n]
            change = true
            A[n] = temp
        fi
    od
until !change
```

- \( \text{fix}(s) := A[s] \) after loop (Kildall ’77)
- Coincidence Theorem (Kam, Ullman ’77)

\[ \text{fix}(s) = \bigcap_{\text{Path } P_i \text{ ending in } s} F_{P_i}(⊥) \]

- If \( \mathcal{F} \) is not distributive, the equality becomes \( \sqsubseteq \), i.e., the fixed point is only a conservative approximation
Reaching definitions

- Reaching definitions is classical compiler problem
- Also prerequisite for computing dependence graphs
- Does a definition of a variable reach the use at a statement?
- Without being redefined underway?
- $\text{Def}(n)$ set of variables defined at $n$
- $\text{Use}(n)$ set of variables used at $n$
- Definition $d$, variable $v := \text{var}(d)$ where $v \in \text{Def}(n)$
- $d$ reaches node $n'$ if $\exists \text{Path } P = (n=n_0, ..., n_k=n')$ in $G$, $k > 0$
  $\forall i \in 1, ..., k - 1: v \notin \text{Def}(n_i)$
Reaching definitions as a DMDFF

- $\mathcal{L} = (\mathcal{P}(D), \supseteq, \cup, \emptyset)$ (powerset of all definitions)
- Transfer functions derived from abstract semantics of variable assignment
- $F_n(X) = X \setminus \text{kill}(n) \cup \text{gen}(n)$
- Statement $n$ defines variables in $\text{Def}(n)$, so all definitions in $D$ defining the same variable can no longer be visible (killed)
  \[
  \text{kill}(n) := \bigcup_{v \in \text{Def}(n)} D_v, \quad \text{where } D_v := \{d \in D \mid v = \text{var}(d)\}
  \]
- All elements of $\text{Def}(n)$ are generated by $n$, i.e. visible after its execution
  \[
  \text{gen}(n) := \text{Def}(n)
  \]
- This MDFF is distributive (MOP and MFP coincide)

2.1.3 Static Single Assignment Form

A very popular program representation is called static single-assignment (SSA) form \cite{CFR91}. It effectively separates a program’s values from the program’s variables and thus enables several more effective optimizations. A program is in static single-assignment form if every variable is assigned at most once in the source code. It is called static because the variable may very well be assigned to multiple times during program execution, e.g. in a loop. For program analysis, the outstanding property of SSA form is that it makes def-use chains, as computed by reaching definitions in the last section, explicit and allows flow-sensitive analysis for the program’s variables, as each variable is assigned to exactly once. Because of that, SSA form has become a standard intermediate representation in program analysis.

To compute the static single-assignment form \cite{CFR91}, one usually introduces a subscript for each variable that is incremented at each assignment statement. At join points in the control flow graph, new assignments with a so-called $\rightarrow$-operator need to be inserted that represent the choice of the appropriate variable according to the program’s flow. Note that programs in SSA form are equivalent to the original program. As an example, consider Figure 2.2, where a program is depicted alongside its SSA form. In line 5 of the SSA form, $\text{sum}_2$ is defined as either $\text{sum}_1$, if the while loop has not yet been entered, or else as $\text{sum}_3$, the last value defined in the loop body.
Basic Blocks

- Exactly one entry point
  no code within is the destination of a jump instruction

- Exactly one exit point
  only the last instruction can be a predicate

- Whenever the first instruction in a basic block is executed,
  the rest of the instructions are necessarily executed exactly
  once, in order

- Simplifies analysis when performed on basic blocks instead
  of statements
Example

(1) read(n);
(2) i = 1;
(3) sum = 0;
(4) prod = 1;
(5) while (i <= n) {
(6)   sum = sum + i;
(7)   prod = prod * i;
(8)   i++;
(9) }
(10) write(sum);
(11) write(prod);

- gen(B1) = {d1, d2, d3, d4}
- kill(B1) = {d6, d7, d8}
- use(B1) = Ø //maybe {n}
- gen(B2) = Ø = kill(B3);
- use(B2) = {i,n}
- gen(B3) = {d6, d7, d8}
- kill(B3) = {d2, d3, d4}
- use(B3) = {i, sum, prod}
- gen(B4) = Ø = kill(B4)
- use(B4) = {sum, prod}
Example

1. \texttt{read(n);}
2. \texttt{i = 1;}
3. \texttt{sum = 0;}
4. \texttt{prod = 1;}
5. \texttt{while (i <= n) {}
6. \texttt{\quad sum = sum + i;}
7. \texttt{\quad prod = prod * i;}
8. \texttt{\quad i++;}
9. \texttt{}}
10. \texttt{write(sum);}
11. \texttt{write(prod);}

- Initialization: \texttt{in} = [\emptyset, \emptyset, \emptyset, \emptyset]
- \texttt{in(B1)} = \emptyset, \texttt{change} = \text{F}
- \texttt{in(B2)} = \texttt{out(B1)} \cup \texttt{out(B3)} = \text{F}_{B1}(\text{in(B1)}) \cup \text{F}_{B3}(\text{in(B3)}) = (\emptyset \setminus \{d6, d7, d8\} \cup \{d1, d2, d3, d4\}) \cup (\emptyset \setminus \{d2, d3, d4\} \cup \{d6, d7, d8\}) = \{d1, d2, d3, d4, d6, d7, d8\}
- \texttt{change} = \text{T}
- \texttt{in(B3)} = \texttt{out(B2)} = \text{F}_{B2}(\text{in(B2)}) = \{d1, d2, d3, d4, d6, d7, d8\} \setminus \emptyset \cup \emptyset = \{d1, d2, d3, d4, d6, d7, d8\}, \texttt{change} = \text{T}
Secure Information Flow

Static Program Analysis

Example

(1) read(n);
(2) i = 1;
(3) sum = 0;
(4) prod = 1;

(5) while (i <= n) {
    (6) sum = sum + i;
    (7) prod = prod * i;
    (8) i++;

(9) }

(10) write(sum);
(11) write(prod);

- in(B1) = ∅
- in(B2) = {d1, d2, d3, d4, d6, d7, d8}
- in(B3) = {d1, d2, d3, d4, d6, d7, d8}
- in(B4) = out(B2) = F_{B2}(in(B2)) = \{d1, d2, d3, d4, d6, d7, d8\} \setminus ∅ \cup ∅ = \{d1, d2, d3, d4, d6, d7, d8\}
  change = T

- ...

Static Program Analysis

Example

(1) read(n);
(2) i = 1;
(3) sum = 0;
(4) prod = 1;

(5) while (i <= n) {
(6) sum = sum + i;
(7) prod = prod * i;
(8) i++;
(9) }

(10) write(sum);
(11) write(prod);

- in(B1) = ∅, change = F
- in(B2) = out(B1) ∪ out(B2) = {d1, d2, d3, d4, d6, d7, d8}, change = F
- in(B3) = out(B2) = {d1, d2, d3, d4, d6, d7, d8}, change = F
- in(B4) = out(B2) = {d1, d2, d3, d4, d6, d7, d8}, change = F
- Algorithm terminates here

need to use transfer functions in the block to determine reaching definitions for statements
**Static Program Analysis**

**Example**

1. `read(n);`
2. `i = 1;`
3. `sum = 0;`
4. `prod = 1;`
5. `while (i <= n) {`
   6. `sum = sum + i;`
   7. `prod = prod * i;`
   8. `i++;`
9. `}`
10. `write(sum);`
11. `write(prod);`

- `in(B1) = ∅`
- `in(B2) = {d1, d2, d3, d4, d6, d7, d8}`
- `in(B3) = {d1, d2, d3, d4, d6, d7, d8}`
- `in(B4) = {d1, d2, d3, d4, d6, d7, d8}`
- at beginning of B2 all definitions are visible, in particular both definitions of `i`, `sum`, and `prod`

- But:
  - `in(7) = F6(in(B3)) = {d1, d2, d4, d6, d7, d8}`