Compiler

Static Analysis:

Reaching Definition Analysis

Reaching Definition (RD) Analysis – Motivation

The following code are accepted by phases up to type checking:

```
void baz() {
  int x;
  int y;
  y = x;
}
```

```
int bazz(A a) {
  int x;
  if (a == null) { return 0; }
  return a.x + x;
}
```

We'd like to notify users about the above ill-formed code (similar to Eclipse IDE)!

RD Analysis — Overview

- An assignment (definition) of the form

$$[\texttt{x = e;}]^{l}$$

may reach a point l' if

  - there is an execution of the program that reaches l' where x was last assigned a value at l

RD Analysis — Overview

- Note the emphasis of may reach
- It is very common to distinguish may analyses from must analyses.
- A helpful way to think about this distinction is to think in terms of program paths.
  - *may* X means there exists a path on which X happens
  - *must* X means for all paths X happens
- What if we have an analysis problem that requires X on no paths?

RD Example (1)

```
static int factorial(int n) {
  int result;
  int i;
  [StaticJavaLib.assertTrue(n >= 1);]1
  [result = 1;]2
  [i = 2;]3
  [while (i <= n) {
    [result = result * i;]5
    [i = i + 1;]6
  }]4
  [return result;]7
}
```

Clearly the definition in 2 may reach 5, we describe this more compactly by saying `(result, 2)` reaches the entry 5.

RD Example (2)

Entry Set

```
                                          static int factorial(int n) {
                                            int r;
                                            int i;
                  (n,•),(r,?),(i,?)         [StaticJavaLib.assertTrue(n >= 1);]^1
                  (n,•),(r,?),(i,?)         [r = 1;]^2
                  (n,•),(r,2),(i,?)         [i = 2;]^3
      (n,•),(r,2),(r,5),(i,3),(i,6)         [while (i <= n) {
      (n,•),(r,2),(r,5),(i,3),(i,6)           [r = r * i;]^5
            (n,•),(r,5),(i,3),(i,6)           [i = i + 1;]^6
                                            }]^4
(n,•),(r,2),(r,5),(r,7),(i,3),(i,6)         [return r;]^7
                                          }
```

Dot (•) denotes definition from a formal parameter (or a field)
Question mark (?) denotes unknown definition

RD Example (3)

Exit Set

```
                                           static int factorial(int n) {
                                             int r;
                                             int i;
                     (n,•),(r,?),(i,?)       [StaticJavaLib.assertTrue(n >= 1);]1
                     (n,•),(r,2),(i,?)       [r = 1;]2
                     (n,•),(r,2),(i,3)       [i = 2;]3
         (n,•),(r,2),(r,5),(i,3),(i,6)       [while (i <= n) {
               (n,•),(r,5),(i,3),(i,6)         [r = r * i;]5
                     (n,•),(r,5),(i,6)         [i = i + 1;]6
                                             }]4
   (n,•),(r,2),(r,5),(r,7),(i,3),(i,6)       [return r;]7
                                           }
```

Dot (•) denotes definition from a formal parameter (or a field)
Question mark (?) denotes unknown definition

Denoting RD

- We can describe the set of reaching definitions for the entry and exit of each program point as a pair of functions

$$(\mathrm{RD}_{entry}, \mathrm{RD}_{exit})$$

  where
  - $\mathrm{RD}_i : \mathrm{Lab}_* \to P(\mathrm{Var}_* \times (\{ \bullet, ? \} \cup \mathrm{Lab}_*))$
- Notes
  - Lab_* and Var_* are the subsets of labels and variables occurring in the program under analysis, e.g., { 1, 2, …, 7 } and { n, `result`, `i` }
  - dot (•) denotes definition from a formal parameter (or a field)
  - question mark (?) denotes unknown definition

RD Example (4)

l	$RD_{entry}(l)$	$RD_{exit}(l)$
1	(n,•),(r,?),(i,?)	(n,•),(r,?),(i,?)
2	(n,•),(r,?),(i,?)	(n,•),(r,2),(i,?)
3	(n,•),(r,2),(i,?)	(n,•),(r,2),(i,3)
4	(n,•),(r,2),(r,5),(i,3),(i,6)	(n,•),(r,2),(r,5),(i,3),(i,6)
5	(n,•),(r,2),(r,5),(i,3),(i,6)	(n,•),(r,5),(i,3),(i,6)
6	(n,•),(r,5),(i,3),(i,6)	(n,•),(r,5),(i,6)
7	(n,•),(r,2),(r,5),(r,7),(i,3),(i,6)	(n,•),(r,2),(r,5),(r,7),(i,3),(i,6)

Observation

- For blocks that are assignments
  - i.e., 2, 3, 5, and 6
  - entry and exit values change for the defined variable
- For blocks that are not assignments
  - i.e., 1, 4, and 7
  - entry and exit values are the same
- The effect of statements on the values are localized

Uses for RD — Compiler Optimization

- An occurrence of a variable x at statement l is a constant c, if for all reaching definitions $(x, l') \in RD_{entry}(l)$, the value assigned to x at l' is c.
- If all reaching definitions for an expression are outside the loop you can move the expression outside the loop (loop-invariant code motion).
- Construct program dependence graph (PDG). Edges in this graph for a statement l are built as follows: for each reaching definition $(x, l') \in RD_{entry}(l)$, we introduce an edge (l', l).

Uses for RD — Software Engineering Tools

- If for a statement *l*, that uses *x* there is a reaching definition $(x, ?) \in RD_{entry}(l)$ then there is a potential uninitialized use of *x*.
- Used in debugging: if a value at a breakpoint has a bad value then you set breakpoints at the reaching definitions and rerun.
- Program slicers are built using the PDG.

Safety of RD

- Every reachable definition that the program can execute should be detected (may detect more definitions)
  - identify fewer constants
  - fail to move some loop-invariant code
  - issue uninitialization warnings for initialized variables
- Never miss a reaching definition
  - transform an expression based on mistaken impression it is constant
  - move expressions out of loop that change in loop
  - fail to issue a warning when variable may be uninitialized

Is It Safe?

- Are the reaching definitions in the table safe for the example?
  - do they contain all of the reaching definitions that can be executed?
  - are there any extra reaching definitions?

l	$RD_{entry}(l)$	$RD_{exit}(l)$
1	(n,•),(r,?),(i,?)	(n,•),(r,?),(i,?)
1'	(n,•),(r,?),(i,?)	(n,•),(r,?),(r,2),(i,?)
5	(n,•),(r,2),(r,5),(i,3),(i,6)	(n,•),(r,5),(i,3),(i,6)
5'	(n,•),(r,2),(i,3),(i,6)	(n,•),(r,5),(i,3),(i,6)

Data Flow Analysis

- Traditionally, Data Flow Analysis exploit the results of Control Flow Analysis
  - information about the sequencing of execution of parts of a program's syntax.
- We have discussed CFG in the previous lecture
  - we assume that we have build CFG for the method under analysis for the next set of slides

Data Flow Analysis

- We will formulate a RD flow analysis to calculate
  - sets of facts, i.e., sets of pairs from $\mathrm{Var}_* \times \mathrm{Lab}_*$
  - at entry and exit of each labeled statement
- Note that we could choose to compute this information for other points in the program, e.g., inside the RHS of an assignment statement

Two Approaches to Formulating Data Flow Analysis

- Equational approach
  - define a system of simultanous equations
  - 2 | Lab_* | variables (to record sets of facts)
  - 2 | Lab_* | equations (to capture effects of stmts)
- Constraint approach
  - define a system of inclusions
  - 2 | Lab_* | variables (to record sets of facts)
  - inclusion constraints expressing relationships between the sets of facts associated with different statements

The Equational Approach (1)

- For each labeled statement
  - $RD_{entry}(l)$: set of reaching definitions data flow facts before execution of l
  - $RD_{exit}(l)$: set of facts after execution of l
- Statement effects on data flow facts captured by equations
  - $RD_{exit}(l) = f\,(RD_{entry}(l))$
  - The definition of f depends on the kind of statement
    - $[x = e\;;]^l$: $RD_{exit}(l) = RD_{entry}(l) \setminus \{ (x, l') \mid l' \in \mathrm{Lab}_* \} \cup \{ (x, l) \}$
    - $[...]^l$: $RD_{exit}(l) = RD_{\mathrm{entry}}(l)$

Equations for Statement Effects: Factorial Example

- $RD_{exit}(1) = RD_{entry}(1)$
```
[StaticJavaLib.assertTrue(n >= 1);]1
```
- $RD_{exit}(2) = RD_{entry}(2) \,/\, \{ (r, l') \mid l' \in Lab_* \} \cup \{ (r, 2) \}$
```
[r = 1;]2
```
- $RD_{exit}(3) = RD_{entry}(3) \,/\, \{ (i, l') \mid l' \in Lab_* \} \cup \{ (i, 3) \}$
```
[i = 2;]3
```
- $RD_{exit}(4) = RD_{entry}(4)$
```
[while (i <= n) { … }]4
```
- $RD_{exit}(5) = RD_{entry}(5) \,/\, \{ (r, l') \mid l' \in Lab_* \} \cup \{ (r, 5) \}$
```
[r = r * i;]5
```
- $RD_{exit}(6) = RD_{entry}(6) \,/\, \{ (i, l') \mid l' \in Lab_* \} \cup \{ (i, 6) \}$
```
[i = i + 1;]6
```
- $RD_{exit}(7) = RD_{entry}(7)$
```
[return r;]7
```

The Equational Approach (2)

- Statement sequencing is accounted for by introducing equations that propagate facts between control flow predecessors and successors.
  - $RD_{entry}(l) = \bigcup_{l' \in preds(l)} RD_{exit}(l')$
- Since definitions are not introduced in transitioning between statements
  - any fact that holds at the exit of a statement also holds at the entry of each of its successors
- The initial statement has no predecessor so treat it specially
  - $RD_{entry}(b_{init}) = \{ (x,\bullet) \mid x \in (\mathrm{Param}_* \cup \mathrm{Field}_*)\} \cup \{ (x,?) \mid x \in \mathrm{Local}_* \}$

Equations for Inter-statement Propagation: Factorial Example

- $RD_{entry}(1) = \{ (\mathtt{n},\bullet), (\mathtt{r},?), (\mathtt{i},?) \}$
- $RD_{entry}(2) = RD_{exit}(1)$
- $RD_{entry}(3) = RD_{exit}(2)$
- $RD_{entry}(4) = RD_{exit}(3) \cup RD_{exit}(6)$
- $RD_{entry}(5) = RD_{exit}(4)$
- $RD_{entry}(6) = RD_{exit}(5)$
- $RD_{entry}(7) = RD_{exit}(4)$

Solving The System of Equations

- Two approaches
  - elimination/substitution method
  - fixpoint iteration method
- We'll look at the second method
  - easily automated
  - finds the *least* solution
  - minor requirements on data flow facts and equations

Lattice of Data Flow Facts

- Variables in the equations range over sets of data flow facts
  - there are finitely many facts
  - reaching definition facts are naturally ordered by inclusion
- A powerset ordered by inclusion is a complete lattice

Posets

- A partially ordered set – poset (S, $\sqsubseteq$) is,
  - a set S
  - a partial order $\sqsubseteq$, which is a reflexive, transitive and anti-symmetric relation
- Examples
  - ({ 1, 2, 3 }, ≤)
  - ({ {1}, {2}, {1, 2} }, ⊆)
- While not guaranteed to exists, all of the posets we will discuss have greatest lower (*glb*) and least upper (*lub*) bounds.

Bounds

- A subset Y of a poset $(S, \sqsubseteq)$ has $l \in S$ as an
  - upper bound if $\forall l' \in Y.\ l' \sqsubseteq l$
    - e.g., ({ 1, 2, 3 }, ≤), Y = { 2 }, $l \in$ { 2, 3 }
  - lower bound if $\forall l' \in Y.\ l \sqsubseteq l'$
    - e.g., ({ {1}, {2}, {1, 2} }, $\subseteq$), Y = { {1, 2} }, $l \in$ { {1}, {2}, {1, 2} }
- A *least upper bound* (*lub*) l of Y is an upper bound such that $l \sqsubseteq l'$ where l' is an upper bound of Y
  - e.g., ({ 1, 2, 3 }, ≤), Y = { 2 }, *lub* = 2
- A *greatest lower bound* (*glb*) l of Y is a lower bound such that $l' \sqsubseteq l$ where l' is a lower bound of Y
  - e.g., ({ {1}, {2}, {1, 2} }, $\subseteq$), Y = { {1}, {2} }, no *glb*

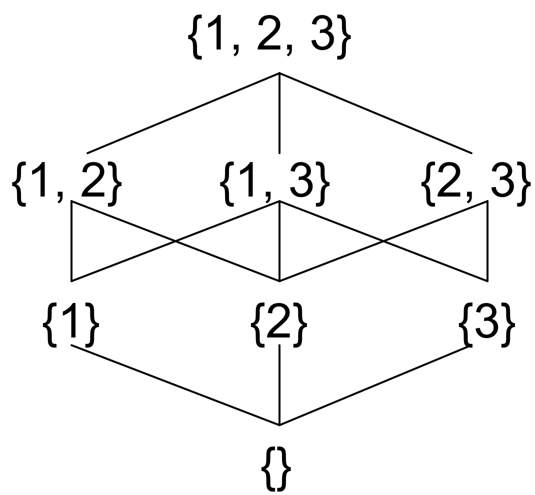
Finding Bounds

- If they exist, *lub* and *glb* are unique.
- The least upper bound of two subsets is denoted $l_1 \sqcup l_2$ which is called the *join* operator.
  - e.g., ({ 1, 2, 3, 4 }, ≤), $\{2, 3\} \sqcup \{2\} = 3$
- The greatest lower bound of two subsets is denoted $l_1 \sqcap l_2$ which is called the *meet* operator.
  - e.g., ({ {}, {1}, { 2 }, {1, 2} }, ⊆), $\{ \{1\} \} \sqcap \{ \{1, 2\} \} = \{1\}$

Complete Lattices

- A complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ consists of
  - a poset $(L, \sqsubseteq)$
  - where *lub* and *glb* exist for all $Y \subseteq L$
  - $\bot = \sqcap L$ is the least element
  - $\top = \sqcup L$ is the greatest element
- In practice we'll only require $\sqsubseteq$ and either $\sqcup$ and $\bot$ or $\sqcap$ and $\top$.

Complete Lattice — Example



$$(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top) = (\{\ \{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\ \}, \subseteq, \supseteq, \cup, \cap, \{\}, \{1,2,3\})$$

i.e., $L = P(\{\ 1, 2, 3\ \})$

Sets of Reaching Definitions Facts

- $\mathcal{P}(\mathrm{Var}_* \times (\{\bullet, ?\} \cup \mathrm{Lab}_*))$
  - $\sqsubseteq \; = \; \subseteq$
  - $\sqcup \; = \; \cup$
  - $\perp \; = \; \emptyset$
- So, we are working with a complete lattice

Vector Representation of Equations

- Consider the 14 variables as a single vector value

  $\overrightarrow{RD} = (RD_{entry}(1), RD_{exit}(1), \ldots, RD_{exit}(7))$

- 14-tuples of complete lattice values, $\overrightarrow{RD} = (RD_1, RD_2, \ldots, RD_{14})$, form a complete lattice
  - $\overrightarrow{RD} \sqsubseteq \overrightarrow{RD'}$ *iff* $\forall$ i. $RD_i \subseteq RD_i'$
  - $\overrightarrow{RD} \sqcup \overrightarrow{RD'} = (RD_1 \cup RD_1', \ldots, RD_{14} \cup RD_{14}')$
  - $\perp = \vec{\emptyset} = (\emptyset, \ldots, \emptyset)$

Equations as A Function

We can write the equation system as

$$\overrightarrow{\mathrm{RD}} = F(\overrightarrow{\mathrm{RD}})$$

where

$$F(\overrightarrow{\mathrm{RD}}) = (\ F_{entry}(1)(\overrightarrow{\mathrm{RD}}),\ \ldots,\ F_{exit}(7)(\overrightarrow{\mathrm{RD}})\ )$$

where the component functions encode the equations, for example,

$$F_{exit}(2)(\ldots) = \mathrm{RD}_{entry}(2)\ /\ \{\ (\mathrm{r}, l')\ |\ l' \in \mathrm{Lab}_*\ \} \cup \{\ (\mathrm{r}, 2)\ \}$$

Properties of Functions

- We can be assured that solution methods will produce an answer to a system of equations if the function is monotone
- A function, $f : L \rightarrow L$, is *monotone* if

$$\forall\, l, l'.\ l \sqsubseteq l' \Rightarrow f(l) \sqsubseteq f(l')$$

- Such functions are also termed *order preserving* since a pair of values ordered in a poset/lattice will have their images under *f* in the same relative order, e.g.,
  - For $L = \{ \{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \}$, and $f(l) = l \cup \{ \{3\} \}$ is monotone

F is Monotone

- There are three forms of constituent functions in *F*, e.g.,
  - $F_{entry}(2)(\ldots) = RD_{exit}(1)$
  - $F_{entry}(4)(\ldots) = RD_{exit}(3) \cup RD_{exit}(6)$
  - $F_{exit}(2)(\ldots) = RD_{entry}(2) / \{ (r, l') \mid l' \in Lab_* \} \cup \{ (r, 2) \}$

- These are easily seen to be monotone in the lattice of reaching definition facts.

Fixed Point of F is a Solution

- Given the monotonicity of F it follows that
  - $\vec{\emptyset} \sqsubseteq F(\vec{\emptyset})$
  - $F^n(\vec{\emptyset}) \sqsubseteq F^{n+1}(\vec{\emptyset})$
- Given that the lattice is finite there must be some pair of iterates of F such that
  - $F^n(\vec{\emptyset}) = F^{n+1}(\vec{\emptyset})$
  - $F^n(\vec{\emptyset})$ is a fixed point of F
  - $F^n(\vec{\emptyset})$ is a solution to the equation system

Least Solution

- The fixed point of F is the least solution of the system of equations
- Suppose $\overrightarrow{\mathrm{RD}} = F(\overrightarrow{\mathrm{RD}})$, i.e., $\overrightarrow{\mathrm{RD}}$ is a fixed point
  - $\vec{\emptyset} \sqsubseteq \overrightarrow{\mathrm{RD}}$
  - so, $F(\vec{\emptyset}) \sqsubseteq F(\overrightarrow{\mathrm{RD}})$ (monotonicity)
  - $F(\vec{\emptyset}) \sqsubseteq F(\overrightarrow{\mathrm{RD}})$
  - $F(F(\vec{\emptyset})) \sqsubseteq F(\overrightarrow{\mathrm{RD}}) = \overrightarrow{\mathrm{RD}}$ (monotonicity)
  - …
  - $F^n(\vec{\emptyset}) \sqsubseteq F(\overrightarrow{\mathrm{RD}}) = \overrightarrow{\mathrm{RD}}$ (monotonicity)

Chaotic Iteration

- The weakest specification of an iterative algorithm for calculating fix point solutions for flow analysis problems

$RD_1 := \emptyset$
…
$RD_{14} := \emptyset$
while $\exists\, j.\ RD_j \neq F_j\,(RD_1, \ldots, RD_{14})$ **do**
   $RD_j := F_j\,(RD_1, \ldots, RD_{14})$

- The algorithm continues to loop until all components have reached a local fixed point.
- So, if the algorithm terminates a fixed point of F is reached.

Properties of Algorithm

- Will it terminate?
  - executing the loop body means there is some $RD_j \neq F_j$ $(RD_1, \ldots, RD_{14})$
  - this only happens if $RD_j \subset F_j$ $(RD_1, \ldots, RD_{14})$
  - so, the assignment in the loop body increases the size of RD_j
  - but, this can only happen a finite number of times since the lattice is finite
- Will it produce the least solution?
  - think about that the fixed point solution is the least solution