Dataflow analysis
Dataflow analysis: what is it?

• A common framework for expressing algorithms that compute information about a program

• Why is such a framework useful?

• It provides a common language, which makes it easier to:
  – communicate your analysis to others
  – compare analyses
  – adapt techniques from one analysis to another
  – reuse implementations (eg: dataflow analysis frameworks)
Data flow analysis

• Goal:
  – collect information about how a procedure manipulates its data

• This information is used in various optimizations
  – For example, knowledge about what expressions are available at some point helps in common subexpression elimination.

• IMPORTANT!
  – Soundness is a must: Data flow analysis should never tell us that a transformation is safe when in fact it is not.
  – It is better to not perform a valid optimization that to perform one that changes the function of the program.
Soundness is a must!

- Data flow analysis should never tell us that a transformation is safe when in fact it is not.

- When doing data flow analysis we must be
  - Conservative
    - Do not consider information that may not preserve the behavior of the program
  - Aggressive
    - Try to collect information that is as exact as possible, so we can get the greatest benefit from our optimizations.
Global Iterative Data Flow Analysis

• **Global:**
  – Performed on the control flow graph
  – Goal = to collect information at the *beginning* and *end* of each basic block

• **Iterative:**
  – Construct data flow *equations* that describe how information flows through each basic block and solve them by iteratively converging on a solution.
  – The “ingredients” of the equations:
    • Algebraic representation of the property of interest
    • Labels associated to the control flow diagrams
Global Iterative Data Flow Analysis

- Components of data flow equations
  - Sets containing collected information
    - **In** (or **entry**) set: information coming into the BB from outside (following flow of data)
    - **gen** set: information generated/collected within the BB
    - **kill** set: information that, due to action within the BB, will affect what has been collected outside the BB
    - **out** (or **exit**) set: information leaving the BB
  - Functions (operations on these sets)
    - **Transfer functions** describe how information changes as it flows through a basic block
    - **Meet functions** describe how information from multiple paths is combined.
Global Iterative Data Flow Analysis

- Algorithm sketch
  - Typically, a bit vector is used to store the information.
    - For example, in reaching definitions, each bit position corresponds to one definition.
  - We use an iterative fixed-point algorithm.
  - Depending on the nature of the problem we are solving, we may need to traverse each basic block in a forward (top-down) or backward direction.
    - The order in which we "visit" each BB is not important in terms of algorithm correctness, but is important in terms of efficiency.
  - In & Out sets should be initialized in a conservative and aggressive way.

```
Initialize gen and kill sets
Initialize in or out sets (depending on "direction")
while there are no changes in in and out sets {
    for each BB {
        apply meet function
        apply transfer function
    }
}
```
Typical problems

- Reaching definitions
  - For each use of a variable, find all definitions that reach it.

- Upward exposed uses
  - For each definition of a variable, find all uses that it reaches.

- Live variables
  - For a point $p$ and a variable $v$, determine whether $v$ is live at $p$.

- Available expressions
  - Find all expressions whose value is available at some point $p$.

- Very Busy expressions
  - Find all expressions whose value will be used in all the next paths.
Reaching definitions

• Determine which definitions of a variable may reach a use of the variable.
  – For each use, list the definitions that reach it. This is also called a ud-chain.
  – In global data flow analysis, we collect such information at the endpoints of a basic block, but we can do additional local analysis within each block.

• Uses of reaching definitions:
  – constant propagation
    • we need to know that all the definitions that reach a variable assign it to the same constant
  – copy propagation
    • we need to know whether a particular copy statement is the only definition that reaches a use.
  – code motion
    • we need to know whether a computation is loop-invariant
boolean x = true;
while (x) {
    ... // no change to x (and no exit/return stmt)
}

• The program doesn’t terminate.
• **Proof**: the only assignment to x is at top, so x is always true.
As a Control Flow Graph

```plaintext
x = true

if x == true

"body"
```
Formulation: Reaching Definitions

• Each place some variable $x$ is assigned is a *definition*.

• **Ask:**
  for this use of $x$, where *could* $x$ last have been defined?

• **In our example:**
  only at $x=\text{true}$. 
Example: Reaching Definitions

\[ d_1: x = \text{true} \]

\[ \text{if } x == \text{true} \]

\[ d_2: a = 10 \]
Clincher

- Since at $x = \text{true}$, $d_1$ is the only definition of $x$ that reaches, it must be that $x$ is true at that point.

- The conditional is not really a conditional and can be replaced by a branch.
Not Always That Easy

```java
int i = 2; int j = 3;
while (i != j) {
    if (i < j) i += 2;
    else j += 2;
}
```

- We’ll develop techniques for this problem, but later …
The Control Flow Graph

d_1: i = 2

d_2: j = 3

if i != j

if i < j

if i != j

if i < j

\[ d_1 \] \[ d_2 \]

\[ d_3 \] \[ d_4 \]

\[ d_1, d_2, d_3, d_4 \]

\[ d_3: i = i+2 \]

\[ d_4: j = j+2 \]
DFA is Sufficient Only

- In this example, \(i\) can be defined in two places, and \(j\) in two places.
- No obvious way to discover that \(i \neq j\) is always true.
- But OK, because reaching definitions is sufficient to catch most opportunities for constant folding (replacement of a variable by its only possible value).
Example: Be Conservative

```java
boolean x = true;
while (x) {
    . . . *p = false; . . .
}
```

• Is it possible that `p` points to `x`?
As a Control Flow Graph

\[
\begin{align*}
\text{d}_1: \ & x = \text{true} \\
\text{d}_2: \ & *p = \text{false}
\end{align*}
\]

Another def of x
Possible Resolution

• Just as data-flow analysis of “reaching definitions” can tell what definitions of \( x \) might reach a point, another DFA can eliminate cases where \( p \) definitely does not point to \( x \).

• **Example**: the only definition of \( p \) is \( p = \&y \) and there is no possibility that \( y \) is an alias of \( x \).
Formalization:
Reaching definitions Analysis

• How can we formalize a definition $D$?
  By a pair $(x,n)$ where $x$ is the variable modified by $D$, and $n$ identifies the assignment $D$.

• A definition $D$ reaches a point $p$ if there is a path from $D$ to $p$ along which $D$ is not killed.

• A definition $D$ of a variable $x$ is killed when there is a redefinition of $x$.

• How can we represent the set of definitions reaching a point?
Reaching definitions

- What is safe?
  - To assume that a definition reaches a point even if it turns out not to.
  - The computed set of definitions reaching a point $p$ will be a superset of the actual set of definitions reaching $p$.
  - It’s a “possible”, not a “definite” property.
  - Goal: make the set of reaching definitions as small as possible (i.e. as close to the actual set as possible).
Reaching definitions

• How are the **gen** and **kill** sets defined?
  – \( \text{gen}[B] = \{ \text{definitions that appear in B and reach the end of B} \} \)
  – \( \text{kill}[B] = \{ \text{all definitions that never reach the end of B} \} \)

• What is the direction of the analysis?
  – forward
  – \( \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \)
Reaching definitions

• What is the **confluence** operator?
  – union
  – \( \text{in}[P] = \cup \text{out}[Q] \), over the predecessors \( Q \) of \( P \)

• How do we initialize?
  – start small
    • Why? Because we want the resulting set to be as small as possible
    – for each block \( B \) initialize \( \text{out}[B] = \text{gen}[B] \)
Formal specification

- The reaching Definition Analysis is specified by the following equations:
- For each program point,

\[
\begin{align*}
\text{RD}_{\text{in}}(p) &= \begin{cases} 
1 & \text{if } p \text{ is the initial point in the control graph} \\
\cup \{ \text{RD}_{\text{out}}(q) \mid \text{there is an arrow from } q \text{ to } p \text{ in the CFD} \} 
\end{cases} \\
\text{RD}_{\text{out}}(p) &= \text{gen}_{\text{RD}}(p) \cup (\text{RD}_{\text{in}}(p) \setminus \text{kill}_{\text{RD}}(p))
\end{align*}
\]
input n;
m := 1;

m := m * n;
n := n - 1;

output m;

RD_{in}(1) = \{(n,?), (m,?)\}
RD_{out}(1) = \{(n,?), (m,?)\}

RD_{in}(2) = \{(n,?), (m,?)\}
RD_{out}(2) = \{(n,?), (m,2)\}

RD_{in}(3) = \{\text{RD}_{out}(2) \cup \text{RD}_{out}(5)\}
= \{(n,?), (n,5), (m,2), (m,4)\}
RD_{out}(3) = \{(n,?), (n,5), (m,2), (m,4)\}

RD_{in}(4) = \{(n,?), (n,5), (m,2), (m,4)\}
RD_{out}(4) = \{(n,?), (n,5), (m,4)\}

RD_{in}(5) = \{(n,?), (n,5), (m,4)\}
RD_{out}(5) = \{(n,5), (m,4)\}

RD_{in}(6) = \{(n,?), (n,5), (m,2), (m,4)\}
RD_{out}(6) = \{(n,?), (n,5), (m,2), (m,4)\}
Algorithm

• **Input:** Control Graph Diagram

• **Output:** RD

• **Steps:**
  – step 1 (inizialization):
    • $RD_{in}(p)$ is the emptyset for each $p$
    • $RD_{in}(1) = \mathcal{I} = \{(x,?) \mid x \text{ is a program variable}\}$
• Step 2 (iteration)
  – Flag = TRUE;
    while Flag
      Flag = FALSE;
      for each program point p
        new = \{ f(RD,q) \mid (q,p) \text{ is an edge of the graph} \}
        if RD_{in}(p) \neq new
          Flag = TRUE;
          RD_{in}(p) = new;

      where f(RD,q) = gen_{RD}(q) \cup (RD_{in}(q) \setminus \text{kill}_{RD}(q))
Example

\[
\begin{align*}
\text{input } n; \quad &1 \\
\text{m:= 1; } &2 \\
\text{while } n>1 \text{ do } &3 \\
\text{m:= m * n; } &4 \\
\text{n:= n - 1; } &5 \\
\text{output m; } &6 \\
\end{align*}
\]

\[
\begin{align*}
\text{RD}_{\text{in}}(1)= &\{(n,?), (m,?)\} \\
\text{RD}_{\text{in}}(2)= &\{(n,?), (m,?)\} \\
\text{RD}_{\text{in}}(3)= &\{(n,?), (n,5), (m,2), (m,4)\} \\
\text{RD}_{\text{in}}(4)= &\{(n,?), (n,5), (m,4)\} \\
\text{RD}_{\text{in}}(5)= &\{(n,5), (m,4)\} \\
\text{RD}_{\text{in}}(6)= &\{(n,?), (n,5), (m,2), (m,4)\}
\end{align*}
\]