Data-Flow Analysis

Proving Little Theorems
Data-Flow Equations
Major Examples
An Obvious Theorem

```java
boolean x = true;
while (x) {
    . . . // no change to x
}
```

![Doesn’t terminate.](image)

![Proof: only assignment to x is at top, so x is always true.](image)
As a Flow Graph

- $x = \text{true}$
- if $x == \text{true}$
  - “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 $x == \texttt{true}$, $d_1$ 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 Flow Graph

\[ d_1: i = 2 \]
\[ d_2: j = 3 \]

\[ \text{if } i \neq j \]

\[ d_3 \]
\[ d_4 \]

\[ \text{if } i < j \]

\[ d_1, d_2, d_3, d_4 \]

\[ d_3: i = i+2 \]
\[ d_4: j = j+2 \]
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 \textit{constant folding} (replacement of a variable by its only possible value).
Be Conservative!

◆ (Code optimization only)
◆ It’s OK to discover a subset of the opportunities to make some code-improving transformation.
◆ It’s not OK to think you have an opportunity that you don’t really have.
Example: Be Conservative

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

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

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

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

\[ d_2: *p = \text{false} \]

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 \( x \).
A definition $d$ of a variable $x$ is said to reach a point $p$ in a flow graph if:

1. Every path from the entry of the flow graph to $p$ has $d$ on the path, and
2. After the last occurrence of $d$ there is no possibility that $x$ is redefined.
A basic block can generate a definition.

A basic block can either

1. Kill a definition of $x$ if it surely redefines $x$.
2. Transmit a definition if it may not redefine the same variable(s) as that definition.
Data-Flow Equations --- (2)

◆ Variables:

1. \textbf{IN}(B) = \text{set of definitions reaching the beginning of block B.}
2. \textbf{OUT}(B) = \text{set of definitions reaching the end of B.}
Data-Flow Equations --- (3)

◆ Two kinds of equations:

1. **Confluence equations**: IN(B) in terms of outs of predecessors of B.

2. **Transfer equations**: OUT(B) in terms of IN(B) and what goes on in block B.
Confluence Equations

\[ \text{IN}(B) = \bigcup_{\text{predecessors } P \text{ of } B} \text{OUT}(P) \]
Transfer Equations

- **Generate** a definition in the block if its variable is not *definitely* rewritten later in the basic block.
- **Kill** a definition if its variable is definitely rewritten in the block.
- An internal definition may be both killed and generated.
Example: Gen and Kill

IN = \{d_2(x), d_3(y), d_3(z), d_5(y), d_6(y), d_7(z)\}

Kill includes \{d_1(x), d_2(x), d_3(y), d_5(y), d_6(y),...\}

Gen = \{d_2(x), d_3(x), d_3(z),..., d_4(y)\}

OUT = \{d_2(x), d_3(x), d_3(z),..., d_4(y), d_7(z)\}

\[
\begin{align*}
d_1: & \quad y = 3 \\
d_2: & \quad x = y+z \\
d_3: & \quad *p = 10 \\
d_4: & \quad y = 5
\end{align*}
\]
Transfer Function for a Block

◆ For any block B:

\[ \text{OUT}(B) = (\text{IN}(B) - \text{Kill}(B)) \cup \text{Gen}(B) \]
Iterative Solution to Equations

- For an n-block flow graph, there are $2n$ equations in $2n$ unknowns.
- Alas, the solution is not unique.
  - Standard theory assumes a field of constants; sets are not a field.
- Use iterative solution to get the least fixedpoint.
  - Identifies any def that might reach a point.
Iterative Solution --- (2)

\[ \text{IN(entry)} = \emptyset; \]
for each block \( B \) do \( \text{OUT}(B) = \emptyset; \)
while (changes occur) do
    for each block \( B \) do {
        \[ \text{IN}(B) = \bigcup_{\text{predecessors } P \text{ of } B} \text{OUT}(P); \]
        \[ \text{OUT}(B) = (\text{IN}(B) - \text{Kill}(B)) \cup \text{Gen}(B); \]
    }

Example: Reaching Definitions

IN($B_1$) = {}
OUT($B_1$) = \{ d_1 \}

IN($B_2$) = \{ d_1, d_2 \}
OUT($B_2$) = \{ d_1, d_2 \}

IN($B_3$) = \{ d_1, d_2 \}
OUT($B_3$) = \{ d_2 \}
Aside: Notice the Conservatism

- Not only the most conservative assumption about when a def is killed or gen’d.
- Also the conservative assumption that any path in the flow graph can actually be taken.
- Fine, as long as the optimization is triggered by limitations on the set of RD’s, not by the assumption that a def does not reach.
Another Data-Flow Problem: Available Expressions

◆ An expression $x+y$ is available at a point if no matter what path has been taken to that point from the entry, $x+y$ has been evaluated, and neither $x$ nor $y$ have even possibly been redefined.

◆ Useful for global common-subexpression elimination.
Equations for AE

- The equations for AE are essentially the same as for RD, with one exception.
- Confluence of paths involves intersection of sets of expressions rather than union of sets of definitions.
Gen(B) and Kill(B)

- An expression $x+y$ is *generated* if it is computed in $B$, and afterwards there is no possibility that either $x$ or $y$ is redefined.

- An expression $x+y$ is *killed* if it is not generated in $B$ and either $x$ or $y$ is possibly redefined.
Example

\[
\begin{align*}
x & = x + y \\
z & = a + b
\end{align*}
\]

Generates \(a + b\)

Kills \(x + y, w \times x, \text{ etc.}\)

Kills \(z - w, x + z, \text{ etc.}\)

Generates \(a + b\)
Transfer Equations

Transfer is the same idea:

\[ \text{OUT}(B) = (\text{IN}(B) - \text{Kill}(B)) \cup \text{Gen}(B) \]
Confluence Equations

- Confluence involves intersection, because an expression is available coming into a block if and only if it is available coming out of each predecessor.

\[
\text{IN}(B) = \bigcap_{\text{predecessors } P \text{ of } B} \text{OUT}(P)
\]
Iterative Solution

\[ \text{IN(entry)} = \emptyset; \]
for each block B do OUT(B) = ALL;
while (changes occur) do
    for each block B do {
        \[ \text{IN(B)} = \bigcap_{\text{predecessors P of B}} \text{OUT(P)}; \]
        \[ \text{OUT(B)} = (\text{IN(B)} - \text{Kill(B)}) \cup \text{Gen(B)}; \]
    }

Why It Works

- An expression $x+y$ is unavailable at point $p$ iff there is a path from the entry to $p$ that either:
  1. Never evaluates $x+y$, or
  2. Kills $x+y$ after its last evaluation.

- $\text{IN(entry)} = \emptyset$ takes care of (1).

- $\text{OUT(B)} = \text{ALL}$, plus intersection during iteration handles (2).
Example

Entry

x + y never gen’d

x + y killed

point p

x + y never gen’d
Subtle Point

◆ It is conservative to assume an expression isn’t available, even if it is.
◆ But we don’t have to be “insanely conservative.”
  ♦ If after considering all paths, and assuming \( x+y \) killed by any possibility of redefinition, we still can’t find a path explaining its unavailability, then \( x+y \) is available.
Live Variable Analysis

◆ Variable x is *live* at a point p if on some path from p, x is used before it is redefined.

◆ Useful in code generation: if x is not live on exit from a block, there is no need to copy x from a register to memory.
Equations for Live Variables

- LV is essentially a “backwards” version of RD.
- In place of Gen(B): Use(B) = set of variables $x$ possibly used in B prior to any certain definition of $x$.
- In place of Kill(B): Def(B) = set of variables $x$ certainly defined before any possible use of $x$. 
Transfer Equations

Transfer equations give IN’s in terms of OUT’s:

$$\text{IN}(B) = (\text{OUT}(B) - \text{Def}(B)) \cup \text{Use}(B)$$
Confluence Equations

Confluence involves union over successors, so a variable is in OUT(B) if it is live on entry to any of B’s successors.

\[ \text{OUT}(B) = \bigcup_{\text{successors } S \text{ of } B} \text{IN}(S) \]
Iterative Solution

\[
\text{OUT}(\text{exit}) = \emptyset;
\]

for each block \( B \) do \( \text{IN}(B) = \emptyset; \)

while (changes occur) do

for each block \( B \) do \{

\[
\text{OUT}(B) = \bigcup_{\text{successors } S \text{ of } B} \text{IN}(S);
\]

\[
\text{IN}(B) = (\text{OUT}(B) - \text{Def}(B)) \cup \text{Use}(B);
\]

\}