Data Flow Analysis

We saw in the introduction how values used in one block of a flow graph could be used in other blocks.

Example

`A[j]` computed in one block by code

```
offset := 4*j;
value := base[offset]
```

If a later block also references `A[j]`, we may be able to use *value* without recalculation, or if an assignment to `A[j]` occurs, we may be able to use `offset`.

How do we know that `value` and `offset` can be used again? Could `j` have changed in the interrum?

Many kinds of information about global properties of programs can be gathered and used locally. These are called *data flow analysis problems*.

Example

Above, the global property is that along any path from the first block to the second, the values of the expressions $4 * j$ and $A[j]$ do not change.

`A[j]`

No changes to `j, A[j]`

`:= A[j]`
`A[j] :=`

The Constant Computation Problem

A simple example that we shall focus on is *constant computations*: is it true that a particular variable always has the same value whenever a particular point in the program is reached?

Example

Variable `debug` is set to 1 on entry to suspect code, set to 0 on leaving that code, and not changed otherwise.

If we know the value of `debug` at compile time, either throughout the program or at a particular use, then we can generate better code.

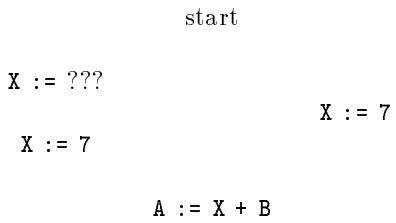
Finding Constant Computations

Suppose we have a use of a variable `X` in a statement like

```
A := X + B
```

One way to deduce that the value of `X` at this statement has a particular constant value is to consider all paths from the start node to the node containing the statement `A := X + B`.

Suppose along every such path (of which there may be an infinite number) there is at least one assignment to `X`, the last of which is `X := 7`.



On entry to the block containing `A := X + B` we can be sure that `X` has the value 7.

If `X` is not changed before use in the block, we can replace statement `A := X + B` by `A := 7 + B`, presumably cheaper.

Incompleteness of DFA Rules

When performing DFA, we never get all potentially useful facts.

"Constant computations" was really an attempt to prove little theorems of the form:

> "Every path from the start node to block B
> has the property that $\cdots$"

In our example, the property was that the last assignment to `X` along the path was `X := 7`.

- This property is sufficient to deduce that `X` has the value 7, but it is not necessary.

Example

There might be a path to block B with the code

```
read Y
Z := Y + 7
X := Z - Y
```

- This code also makes $X = 7$, yet our method would not discover that fact.

Example

There might be a path in the flow graph that led to block B but could not be taken in any real computation, e.g., because it had statements

```
C := D + 1
if C=D goto ···
```

If this last path assigned `X` the value 8, we could not deduce by our method that any computation that really led to block B gave `X` the value 7.

Safe and Unsafe Uses of Information

Since our methods provide only a subset of the true facts, we must be careful about how we use the information provided by DFA.

If we can prove $X = 7$ at a point, then surely `X` is 7 there. However, just because we cannot deduce $X = 7$ by the DFA method chosen does not mean that `X` does not really have the value 7 every time the point is reached.

It would be an error if we made some code improvement that depended for its correctness on the "fact" that `X` was not always 7.

- We must use information in the *safe* direction (what's proved is true).
- Never use information in the *unsafe* direction (what's not proved is false).

Example

We can use DFA to check a program for uses of a variable `X` where there is a path in the flow graph to that use along which `X` might not be defined.

It is tempting to say there must be an error, and refuse to run the program. However, the programmer may know that that path is impossible, for reasons the compiler is not capable of deducing.

Example

`A[i] :=` cannot be an assignment to `A[j]` if we can prove $i \neq j$ at the point.

It is not sufficient to make an "honest effort" to prove $i = j$, and conclude $i \neq j$ if we fail.

DFA Problem #1: Reaching Definitions

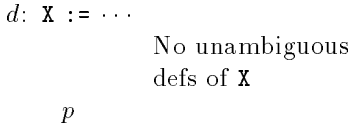
The *reaching definitions* problem is to determine for each block B and each variable `X`, what statements of the program could be the last definition of `X` along some path to the beginning of B.

A statement *defines* `X` if it may assign `X` a value. Typical definitions of `X` include

1. An assignment or read of `X`.
2. A call of a procedure that can access `X`, e.g., because `X` is in the scope of the procedure or the call passes `X` as a parameter (but not by-value).
3. An indirect assignment `*P := A`, where `P` is a pointer variable that could point to `X`.

- Notice that a statement could define more than one variable.
- Definitions of type (1) are *unambiguous*; they surely define the variable.
- Types (2) and (3) are examples of *ambiguous*; they may or may not assign a value to the variable.
- This formulation of "defs" ignores *aliasing* problems, where due to procedure linkages, two variables that appear to be different are actually the same.

Formally, definition d *reaches* point p if there is some path from the start node to p along which d occurs, and subsequent to the last occurrence of d there are no other unambiguous definitions of `X`.



Example

d: `X := A + B`
`*P := C`

Definition d still reaches the point below the pointer assignment.

Uses of Reaching Definitions

- Many uses in algorithms for loop optimization.

Another example: constant computations. If we know all the places where a particular use of `X` could be defined, then we can tell if all such places are statements of the form

`X :=` the same constant

Getting RD's to the Interiors of Blocks

Reaching definitions are computed only for the beginnings of blocks, rather than for all statements, for efficiency reasons.

Knowing what definitions of `X` reach the beginning of block B, we can tell what definitions of `X` reach any statement s of B by the following rules.

1. If no unambiguous defs of `X` occur in B before s, then the reaching definitions of `X` at s include those that reach the beginning of B.
2. If s is preceded by one or more unambiguous defs of `X` within the block, then only the last of these reaches s.
3. If there are ambiguous defs of `X` between the last unambiguous def and s, then these also reach s.

That is, in

d_1 d_2

d_3: `*P := 2`
`A := X + B`

all three defs of `X` may reach the use of `X`.

Data Flow Equations

To reduce calculations about infinite sets of paths to finite ones, and for efficiency in computing reaching definitions, we create and solve a set of *data flow equations.*

The variables of these equations are called IN(B) and OUT(B); there is one pair for each block B.

The intent is that after solution of the equations:

IN(B) = the set of defs reaching the beginning of B

OUT(B) = the set of defs reaching the end of B

We group the equations we write into two classes:

1. *Transfer Equations* that relate OUT for the block to IN, and
2. *Confluence Rules* that tell what happens when paths come together.

Transfer Equations

If we know IN(B) we can compute OUT(B) looking only at what is in the block B. That is, there is some function f_B for which

$$\text{OUT}(B) = f_B(\text{IN}(B))$$

We call this relationship the *transfer equation for* B.

- Different DFA problems yield different transfer equations.

For reaching definitions (and many other problems) f_B has a simple form.

KILL Sets

Say def d *kills* def e if d and e define the same variable, and d is unambiguous.

- Ambiguous defs cannot kill other defs.

KILL(B) is the set of defs not in B that are killed by some def in B.

GEN Sets

Block B *generates* def d if d is in B and is not followed in B by any unambiguous defs of the same variable as d.

GEN(B) is the set of defs generated by B.

The transfer equation for B is:

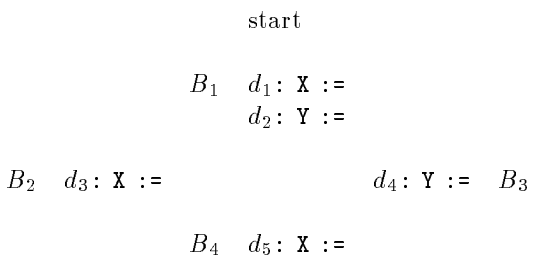
$$\text{OUT}(B) = (\text{IN}(B) - \text{KILL}(B)) \cup \text{GEN}(B)$$

Confluence Rules

A def reaches the beginning of B if and only if it reaches the end of one of B's predecessors. That is:

$$\text{IN}(B) = \cup_{\text{pred. } P \text{ of } B} \text{OUT}(P)$$

Example



$$\begin{aligned}
&\mathrm{GEN}(B_1) = \{d_1, d_2\}\\
&\mathrm{GEN}(B_2) = \{d_3\}\\
&\mathrm{GEN}(B_3) = \{d_4\}\\
&\mathrm{GEN}(B_4) = \{d_5\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{KILL}(B_1) = \{d_3, d_4, d_5\}\\
&\mathrm{KILL}(B_2) = \{d_1, d_5\}\\
&\mathrm{KILL}(B_3) = \{d_2\}\\
&\mathrm{KILL}(B_4) = \{d_1, d_3\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{OUT}(B_1) = (\mathrm{IN}(B_1) - \{d_3, d_4, d_5\}) \cup \{d_1, d_2\}\\
&\mathrm{OUT}(B_2) = (\mathrm{IN}(B_2) - \{d_1, d_5\}) \cup \{d_3\}\\
&\mathrm{OUT}(B_3) = (\mathrm{IN}(B_3) - \{d_2\}) \cup \{d_4\}\\
&\mathrm{OUT}(B_4) = (\mathrm{IN}(B_4) - \{d_1, d_3\}) \cup \{d_5\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{IN}(B_1) = \mathrm{OUT}(B_4)\\
&\mathrm{IN}(B_2) = \mathrm{OUT}(B_1)\\
&\mathrm{IN}(B_3) = \mathrm{OUT}(B_1) \cup \mathrm{OUT}(B_4)\\
&\mathrm{IN}(B_4) = \mathrm{OUT}(B_2) \cup \mathrm{OUT}(B_3)
\end{aligned}$$

Solving Data Flow Equations

We have 8 equations in 8 unknowns, so it looks like we can solve these equations by Gaussian elimination.

- Nonsense! Gaussian elimination works only when the values of the unknowns are taken from a field.
- As a result, the equations cannot be solved by Gaussian elimination or any of the "standard" methods.

In fact, the solution to the equations need not be unique. For example, we could add a spurious def d_6 to all IN's and OUT's, and get another valid solution. Fortunately:

- The DF equations have a *least solution,* in which the IN's and OUT's are subsets of their values in any other solution.
- The least solution is exactly what we want: the set of defs that really reach the points in question along some path.

The Iterative Solution to DF Equations

One simple way to produce the least solution is to

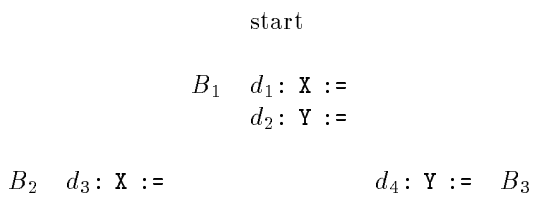
1. Start by assuming nothing reaches anywhere.
2. Repeatedly visit all the nodes, applying the confluence rules to get new IN's and applying the transfer rules to get new OUT's.

```
for each block B do
    OUT(B) := GEN(B);

while changes to any OUT occur do
    for each block B do begin
        IN(B) :=
            ∪_{pred. P of B} OUT(P);
        OUT(B) :=
            (IN(B)− KILL(B))
            ∪ GEN(B)
    end
```

- Note that IN and OUT sets never decrease when recomputed in the while-loop.

Example



	Initial	Round 1	Round 2
IN(B_1)	$\emptyset$	$\{d_5\}$	$\{d_2, d_4, d_5\}$
OUT(B_1)	$\{d_1, d_2\}$	$\{d_1, d_2\}$	$\{d_1, d_2\}$
IN(B_2)	$\emptyset$	$\{d_1, d_2\}$	$\{d_1, d_2\}$
OUT(B_2)	$\{d_3\}$	$\{d_2, d_3\}$	$\{d_2, d_3\}$
IN(B_3)	$\emptyset$	$\{d_1, d_2, d_5\}$	$\{d_1, d_2, d_4, d_5\}$
OUT(B_3)	$\{d_4\}$	$\{d_1, d_4, d_5\}$	$\{d_1, d_4, d_5\}$
IN(B_4)	$\emptyset$	{all}	{all}
OUT(B_4)	$\{d_5\}$	$\{d_2, d_4, d_5\}$	$\{d_2, d_4, d_5\}$

There are no more changes to the OUT's (although changes to IN's occurred on the second round). No further iterations are necessary, and the rightmost column gives the minimal solution.

Data Structures For Reaching Definitions

The sets of definitions IN, OUT, GEN, and KILL can require significant space.

- These sets tend to be dense, so a bit-vector representation is called for.

We must create a table that associates each def with an integer i. The entry for i is either a statement number (for an unambiguous def) or a statement number and variable (for an ambiguous def).

- Simplify by omitting defs of local variables.
- Another space-saving trick: Combine all ambiguous defs of `A` into one. Rationale is that when any ambiguous def of `A` reaches a point, there isn't much we can say anyway.

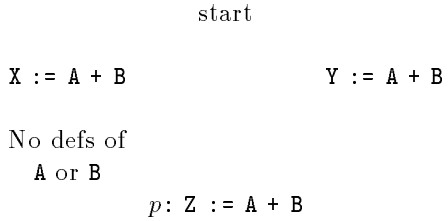
Now, we can represent sets IN, etc., by bit vectors, in which the ith position has 1 if def i is in the set, and 0 if not.

- We can also speed up the DF equations this way. Union is bitwise logical OR; set difference is NOT-AND.

Available Expressions

It is possible to make use of an expression in one block that was computed in another block.

Consider:

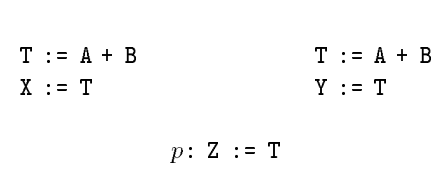


Every path from the start node to a point p contains an evaluation of the expression $A + B$, and after the last calculation of $A + B$ along any such path there is no subsequent def (even ambiguous) of `A` or `B`.

- *Note:* In what follows, we shall often use $+$ as a generic operator.

The expression $A + B$ is said to be *available* at p. If p uses that expression, then it need not be recalculated at p.

Rather, we can introduce a new variable `T` to hold $A + B$, and `T` can be used in place of $A + B$ at p.



We shall later discuss "copy propagation," whereby the "copy" statements `X := T` and `Y := T` can often be eliminated.

- In the worst case, where no copy steps can be avoided, we have increased the code size, although the speed of the code has probably improved.
- Not all occurrences of $A + B$ may require this treatment; we'll discuss the algorithm for global common subexpression elimination later.

Computing Available Expressions

- Similar (but not identical) to RD's.
- We again use variables IN(B) and OUT(B) for each block B.
- The values for variables are subsets of the set of all expressions (that is, right sides of 3-addr statements) computed by the program.
- The intent is that IN(B) is the set of expressions available at the beginning of B and OUT(B) is the set of expressions available at the end of B.

KILL Sets

KILL(B) = the set of expressions $X + Y$ such that there is a definition of `X` or `Y` in B.

- Note this def could be ambiguous, e.g., a pointer assignment.
- If there are potential aliases due to parameter passing, the def could even be of an alias of `X` or `Y`.

GEN Sets

GEN(B) = the set of expressions $X + Y$ computed in B at a position where neither `X` nor `Y` are defined subsequently (including ambiguous defs).

- Unlike RD's these GEN and KILL are not necessarily disjoint, but there is an obvious fixup.

Example

A block consisting of

```
A := B + C
X := Y - Z
B := X + Y
X := A * B
```

- Kills expressions like $A * B$ and $X + Q$ (although $A * B$ is also generated).
- Generates $Y - Z$ and $A * B$.
- $B + C$ is not generated, since C is subsequently defined. Neither is $X + Y$ generated, since `X` is later defined (in fact, `X` is defined by the assignment portion of the same statement that evaluates $X + Y$).

Transfer Equations

As for RD's we can reason that an expression is available at the end of a block if either it is available at the beginning and not killed, or it is generated (regardless of whether it is killed or not). That is:

$$\text{OUT}(B) = (\text{IN}(B) - \text{KILL}(B)) \cup \text{GEN}(B)$$

Confluence Equations

For AE's, the appropriate confluence operator is intersection, rather than union.

$$\text{IN}(B) = \cap_{\text{pred. } P \text{ of } B} \text{OUT}(P)$$

- As a special case, we know that nothing is available at the start node. Thus, we insist that IN(start) is empty, even though the above equation may say otherwise in the case that the start node has predecessors.

Solving the Equations

- As for RD's, there may be many solutions.
- For AE's, we want the largest solution, so we don't rule out the availability of any expression unless we can really find a path along which it is unavailable.
- Largest solution exists because the union of solutions is a solution.

Thus, initialize the IN's and OUT's to make every expression available everywhere except at the initial node. Iteratively apply the rules until we stabilize.

- Unlike RD's, now things only get thrown out of IN's and OUT's. When we converge, it can be shown that we have the largest solution to the equations.

```
IN(start) := ∅;
OUT(start) := GEN(start);
/* above initializes start node;
    it never changes */

for B other than start do
    OUT(B) :=
        (all expressions − KILL(B))
        ∪ GEN(B);

while changes to OUT's occur do
    for B other than start do begin
        IN(B) :=
            ∩pred. P of B OUT(P);
        OUT(B) :=
            (IN(B) − KILL(B))
            ∪ GEN(B)
    end
```

Example

B_1 `Z := X + Y`

B_2 `X := A + B` `A := X + Y` B_3

B_4 `B := ···`

	B_1	B_2	B_3	B_4
GEN	$X+Y$	$A+B$	$X+Y$	none
KILL	none	$X+Y$	$A+B$	$A+B$

	Initial	Round 1	Round 2
IN(B_1)	none	none	none
OUT(B_1)	$X+Y$	$X+Y$	$X+Y$
IN(B_2)	$X+Y, A+B$	$X+Y$	$X+Y$
OUT(B_2)	$A+B$	$A+B$	$A+B$
IN(B_3)	$X+Y, A+B$	$X+Y$	none
OUT(B_3)	$X+Y$	$X+Y$	$X+Y$
IN(B_4)	$X+Y, A+B$	none	none
OUT(B_4)	$X+Y$	none	none

Using Available Expression Information

Suppose we discover that expression $X + Y$ is available at the beginning of block B, i.e., IN(B) contains $X + Y$. If there is a use of expression $X + Y$ within B, and neither `X` nor `Y` is redefined, even ambiguously, before use in B, then we may alter the flow graph to avoid the computation of $X + Y$ within B.

1. Find all statements in the flow graph outside B that have $X + Y$ as right side.
2. Create a new variable `T` to hold $X + Y$.
3. Replace each statement `Z := X + Y` by

```
T := X + Y
Z := T
```

4. Replace uses `Q := X + Y` in B by `Q := T`.

Limiting the Changes

It is possible to determine that some evaluations of $X + Y$ found in (1) can never reach B, so they do not have to participate in (3).

We can detect such cases by a DFA "reaching expressions."

Another approach is to trace the flow graph backwards from `Q := X + Y`, without going through a computation of $X + Y$, to see what occurrences of $X + Y$ we meet.

Another simplification: eliminate common subexpression $X + Y$ only when it is computed in a block B_1 and used in B_2, and B_1 *dominates* B_2 (every path from the start to B_2 goes through B_1).

Data Structures For Available Expressions

- The bit-vector idea works for AE's as well as for RD's.

We must enumerate the expressions that appear on the right side of some statement in the program. Then, create a table whose ith entry is a pointer to an instance of the ith expression.

Represent sets of expressions, like GEN or OUT, by bit vectors, where position i indicates the presence or absence of the ith expression.

Set difference is again implemented by NOT-AND, while union is implemented by OR and intersection is implemented by AND.