Example of Global Data-Flow Analysis

Reaching Definitions

We will illustrate global data-flow analysis with the computation of reaching definition for a very simple and structured programming language.

We want to compute for each block *b* of the program the set REACHES(*b*) of the statements that compute values which are available on entry to *b*.

An assignment statement of the form `x:=E` or a read statement of the form `read x` *must define* `x`.

A call statement `call sub(x)` where `x` is passed by reference or an assignment of the form `*q:= y` where nothing is known about pointer `q`, *may define* `x`.

We say that a statement *s* that must or may define a variable `x` reaches a block *b* if there is a path from *s* to *b* such that there is no statement *t* in the path that kills *s*, that is there is no statement *t* that must define `x`. A block may be reached by more than one statement in a path: a *must define statement s* and one or more *may define statements* on the same variable that appears after *s* in a path to *b*.

We must be conservative to guarantee correct transformations. In the case of reaching definitions conservative is to assume that a definition can reach a point even if it might not whenever the compiler cannot guarantee that this is not the case.

Data-Flow Equations

Consider the following language:

S ::= id:=expression | S;S | **if** expression **then** S **else** S | **do** S **while** expression

We define the following equations to compute reaching definitions. In these equations we compute reaching definitions for statements instead of blocks. We use the following terms:

- gen[S] is the set of definitions “generated by S”.
- kill[S] is the set of definitions “killed” by S.
- in[S] is the set of definitions reaching S (or the top of S)
- out[S] is the set of definitions that reach the bottom of S.

1. When S has the form `a` *:=expression* and label d:
   gen[S] = {d}
   kill[S] = D_a-{d}. D_a is the set of all definitions of `a`.

   out[S] = gen[S] + (in[S]-kill[S])

2. When S is of the form S_1; S_2:
   gen[S] = gen[S_2]+(gen[S_1]-kill[S_2])
   kill[S] = kill[S_2]+(kill[S_1]-gen[S_2])

   in[S_1] = in[S]
   in[S_2] = out[S_1]
   out[S] = out[S_2]

3. When S is of the form **if** ... **then** S_1 **else** S_2

   $gen[S]=gen[S_1]+gen[S_2]$
   $kill[S]=kill[S_1]\cap kill[S_2]$

   $in[S_1] = in[S]$
   $in[S_2] = in[S]$
   $out[S] = out[S_1]+out[S_2]$

4. When S is of the form **do** S_1 **while** ...:
   $gen[S] = gen[S_1]$
   $kill[S] = kill[S_1]$

   $in[S_1] = in[S]+gen[S_1]$
   $out[S] = out[S_1]$

For this problem we don't use a control flow graph. We use an abstract syntax tree for simplicity. Gen[S] and kill[S] are computed bottom up on that tree.

Notice that the real gen[S] is always a subset of the computed gen[S]. For example when S is an **if** statement that always takes the "true" branch, $gen[S_1]+gen[S_2]$ is bigger than the real $gen[S]=gen[S_1]$. At the same time, the real kill[S] is a superset of the computed kill[S].

Notice that out[S] is not the same as gen[S]. The former contains all the definitions that reach the bottom of S while the latter includes only those definitions within S that reach the bottom of S.

In[S] and out[S] are computed starting at the statement S_0 representing the whole program.

Algorithm IN-OUT: Compute in[S] and out[S] for all statements S

Input: An abstract syntax tree of program S_0 and the gen and kill sets for all the statements within the program.

*Output:*in[S] and out[S] for all statements within the program

```
compute_out(S):
    case S

        a := ... :      return(out[S]=gen[S]+(in[S]-kill[S]))

        S_1;S_2 :       in[S_1]=in[S]
                        in[S_2],out[S_1]=compute_out(S_1)
                        return(out[S_2]=compute_out(S_2))

        if ...
        then S_1
        else S_2 :      in[S_1], in[S_2] = in[S]
                        return(out[S]=compute_out(S_1)
                                + compute_out(S_2))

        do S_1
        while ... :     in[S_1] = in[S]+gen[S_1]
                        return(out[S_1]=compute_out(S_1))

    end case
end

in[S_0] = ∅
compute_out(S_0)
```

The sets of statements can be represented with bit vectors. Then unions and intersections take the form of **or** and **and** operations. Only the statements that assign values to program variables have to be taken into account. That is statements (or instructions) assigning to compiler-generated temporaries can be ignored.

In an implementation it is better to do the computations for the basic blocks only. The reaching sets for each statement can be then obtained by applying rule for “$S_1;S_2$” above.

Use-definition chains

Data interconnections may be expressed in a pure form which directly links instructions that produce values to instructions that use them. This may take the following form:

1. For each statement S with input variable v,
2. DEFS(v,S) = in{S}$\cap D_v$. If v is not input to S, DEFS(v,S) is the empty set.
3. For each definition S with output variable v, USES(S) ={T| S is in DEFS(v,T)}.

Once DEFS is computed, USES can be computed by simple inversion

Algorithm US: USES Computation

Input: DEFS

Output: USES

Method:

```
begin
    USES(:):=∅
    for each statement T in the program do
        for each input variable v of T do
            for each statement S in DEFS(v,T) do
                USES(S) += {T}
            od
        od
    od
```

Three applications of the use-definition chains

(From: Kennedy, K. A survey of dataflow analysis techniques. In Muchnick S, and N. Jones. Program Flow Analysis: Theory and Applications, prentice-Hall, N.J.)

Algorithm MK: Mark Useful Definitions

Input:

1. DEFS
2. Set of critical statements, CRIT, which are useful by definition (e.g writes)

Output: For each definition S, MARK(S) = true iff S is useful

Method:

```
begin
    MARK(:) = false
    PILE=CRIT
    while PILE $\neq \emptyset$ do
        S from PILE
        MARK(S)=true
        for each T in $\bigcup_{\forall v} DEFS(v,S)$ do
            if $\neg$MARK(T) then PILE +={T} fi
        od
    od
end
```

Algorithm CP: Constant Propagation

Input:

1. A program PROG
2. A flag CONST(v,S) for each statement S and input variable v of S. Also, CONST(S) for the output variable of S. Initially, CONST(S) is true only if the rhs of S is a constant. CONST(v,S) is false for all v and S.
3. USES and DEFS

Output:

1. The modified CONST flags
2. The mapping VAL(v,S) which provides the run-time constant value of input variable v at statement S. VAL(v,S) is defined only if CONST(v,S) is true. VAL(S) is the value of the output of S. VAL(S) is defined only if CONST(S) is true.

```
        end

*Method*:
        begin /*start with the trivially constant statements */
            PILE = {S in PROG | the rhs of S is a constant}
            while PILE ≠ ∅ do
                T from PILE
                v = output(T)
                for each S in USES(T) do
                    /*check for constant inputs */
                    for each U in DEFS(v,S)-{T} do
                        if ¬CONST(U) or VAL(U)≠VAL(T)
                        then next(2)
                    od
                    /* If it is a constant */
                    CONST(v,S)=true
                    VAL(v,S)=VAL(T)
                    /* is the instruction now constant? */
                    if CONST(w,S) true for all inputs w of S then
                        CONST(S) = true
                        VAL(S) = compute(S)
                        PILE += {S}
                    fi
                od
            od
        end
```

The third example is type analysis. This is important in very high level languages such as SETL where there are no declarations. A statement of the form `x:=y+z` could mean in SETL union of sets or of tuples, integer addition, real addition, etc. A run-time check is needed unless it can be determined at comp[ile time the type of the operands.

We assume that there is a number of atomic type symbols such as I (integer), R (real), N (number, i.e. real or integer), UD (undefined), NS (set of arbitrary elements), S (sets), Z (error), etc.

We also assume an algebra of types encoded in a transition function F which, for each operation op and input types t_1, t_2, ..., t_n of the operands, produces $t_0=F_{op}(t_1, t_2, ..., t_n)$. e.g. real+real is real, real + integer is also real, integer + integer is integer.

Algorithm TA: Type Analysis

Input:

1. A program PROG
2. A mapping TYPE, such that TYPE(v,S) is the best inital estimate of the type of the variable v at the input of S (for most variables this is ‘UD’). Also, TYPE(S) is the type of the output (lhs) of S.
3. DEFS and USES

Output: For each instruction S and input or output variable v, Type(v,S), a conservative approximation to the most specific type information provably true at S.

```
begin
    PILE={S in PROG | no variable in the rhs is of type 'UD'}
    while PILE ≠ ∅ do
        S from PILE
        v = output (S)
        for each T in USES(S) do
            /* recompute type */
            oldtype = TYPE(v,T)
            TYPE(v,T) = $\bigcup_{U \in DEFS(v,T)} TYPE(U)$
            if TYPE(v,T) ≠ oldtype then
                /* a type refinement */
                TYPE(T) = $F_{op(T)}$ (input types of T)
                PILE += {T}
            fi
    od
end
```

Some simple iterative algorithms for data flow analysis

The data flow analysis above was done assuming a structured program and operated on an abstract syntax tree.

One simple approach that works even when the flow graph is not reducible is to use iterative algorithms.

These algorithms try to find a solution to a system of equations. For example, for the reaching definitions problem, we have the following equations defined on each node B (a basic block) of the flow graph:

$$in[B] = \bigcup_{C \in PREDECESSOR(B)} out[C]$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

or, in terms of *in* alone:

$$in[B] = \bigcup_{C \in PRED(B)} gen[C] \cup (in[C] - kill[C])$$

Notice that gen and kill are defined for basic blocks. Also, in[S] is assumed to be the empty set. S is the source node of the flow graph.

These equations do not have a unique solution. What we want is the smallest solution. Otherwise the result would not be as accurate as possible.

There are other important problems that can also be expressed as systems of equations.

The **available expressions** problem is used to do global common subexpression elimination. An expression `x+y` is available at a point p if every path from the initial node to p evaluates `x+y` and after the last such evaluation before reaching p, there are no subsequent assignments to either operand.

A block *kills* expression `x+y` if it assigns (or may assign) `x` or `y` and does not subsequently recompute `x+y`. A block *generates* expression `x+y` if it definitely evaluates `x+y` and does not subsequently redefine `x`or `y`.

The equations are (Here *in* is the set of expressions available at the top of a basic block):

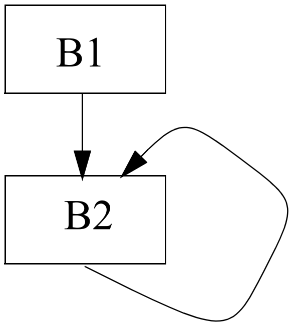
$$out[B] = egen[B] \cup (in[B] - ekill[B])$$

$$in[B] = \bigcap_{C \in PRED[B]} out[C]$$

or in terms of *in* alone:

$$in[B] = \bigcap_{C \in PRED[B]} egen[C] \cup (in[C] - ekill[C])$$

In this case, we want the largest solution to the system of equations. Consider for example, the following flow graph:



where *ekill*[B2] = ∅ and *egen*[B2] = ∅. Also, *out*[B1] is {`x+y`}. By applying the previous equations we can see that {`x+y`} and ∅ are both solutions to the system of equations. Clearly {`x+y`} is more accurate.

Again in this case we assume that in[S] is the empty set.

Another interesting problem is **live variable analysis**. This is computed in the direction opposite to the flow of control. In live variable analysis we wish to know for variable `x` and point p whether the value of `x` at p could be used along some path in the flow graph starting at p. If so, `x` is live at p; otherwise is dead at p.

After a value is computed in a register within a block, it is not necessary to store that value if it is dead at the end of the block. Also, if all registers are full and we need another register, we should favor using a register with a dead value, since that value does not have to be stored.

Let *def*[B] be the set of variables definitively assigned values in B prior to any use of the variable in B, and let *use*[B] be the set of variables whose values may be used in B prior to any definition of the variable.

The equations are (Here *out*[B] are the variables live at the end of the block, and *in*[B], the variables live just before block B.

$$in[B] = use[B] \cup (out[B] - def[B])$$

$$out[B] = \bigcup_{C \in SUCESSOR(B)} in[C]$$

or in terms of *out* alone:

$$out[B] = \bigcup_{C \in SUC(B)} use[C] \cup (out[C] - def[C])$$

The final problem we will mention is **very busy expressions**. An expression that is evaluated regardless of the path taken from a given point is said to be very busy at that point. Such expressions could be moved to the earliest point beyond which they would always be evaluated. It is a transformation that almost always reduces the space occupied by a program but that may affect its execution positively, negatively, or not at all. we define *eval*[B] as the set of expressions that are evaluated in basic block B before any of the operands are assigned to it in the block. We define *kill*[B] to be the set of expressions killed by block B. An expression is killed by a basis block if one of its operands is assigned to in the block.

$$in[B] = eval[B] \cup (out[B] - kill[B])$$

$$out[B] = \bigcap_{C \in SUCESSOR(B)} in[B]$$

The four types of bit vector frameworks

	Set Intersection, “and”, ∀-problems	**Set Union, “or”, ∃-problems**
Top-down problems (operation over prede-cessors)	Available expressions	Reaching definitions
Bottom-up problems (operations over suc-cessors)	Very busy expressions	Live variables

Observation 1. A top-down (respectively bottom-up) problem defined on a single-exit flow graph G can be transformed into a bottom-up (respectively top-down) problem on the reverse graph of G (reverse the arcs) such that a solution to the modified problem gives an almost immediate solution to the original problem.

Observation 2. A set intersection (respectively set union) problem can be transformed into a set union (respectively set intersection) problem such that a solution to the modified problem gives a solution to the original problem. (By DeMorgan’s Laws).

Two simple iterative algorithms for the available expressions problem

We will now use the available expressions problem to illustrate two simple iterative algorithms to compute the *in*[B] sets. We will use a matrix *in*[B,e], that will be one iff an expression *e* is in *in*[B]. Similar matrices for egen and ekill will be used (but not necessarily implemented). To understand the algorithm, consider again the equation:

$$in[B] = \bigcap_{C \in PRED[B]} egen[C] \cup (in[C] - ekill[C])$$

We can rewrite it as a boolean equation:

$$in[B,e] = \prod_{C \in PRED[B]} egen[C,e] \vee (in[C,e] \wedge \overline{ekill[C,e]})$$

or

$$\overline{in[B,e]} = \sum_{C \in PRED[B]} ((\overline{in[C,e]} \wedge \overline{egen[C,e]}) \vee (ekill[C,e] \wedge \overline{egen[C,e]}))$$

Note here the $\sum$ and the $\prod$ symbols stand for logical OR and AND respectively.

```
begin
    PILE = $\emptyset$
    for each expression $e$ in PROG do
        for each basic block B of PROG do
            if B is S or there is C in PRED(B) such that
              $ekill[C,e] \wedge (\neg\ egen[C,e])$ then
                /* By second term of equation */
                $in[B,e] = 0$
                PILE += {B}
            else
                $in[B,e]=1$
            fi
        od
        while PILE $\neq \emptyset$ do
            C from PILE
            if $\neg egen[C,e]$ then
                for each B in SUC(C) do
                    if $in[B,e]=1$ then
                        /* By first term */
                        $in[B,e]=0$
                        PILE += {B}
                    fi
                od
            fi
        od
```

Theorem. The previous algorithm terminates and is correct.

Proof

Termination. For each expression *e*, no node is placed in PILE more than once and each iteration of the **while** loop removes an entry from PILE.

Correctness. *in*[B,e] should be 0 iff either:

1. There is a path from S to B such that *e* is not generated in any node preceding B on this path or
2. there is a path to B such that *e* is killed in the first node of this path and not subsequently generated.

If *in*[*B,e*] is set to zero because of the second term of the equation clearly either 1 or 2 hold. A straightforward induction on the number of iterations of the **while** loop shows that if *in*[*B,e*] is set to zero because of the first term of the equation, then 1 or 2 must hold.

Conversely if 1 or 2 hold, then a straightforward induction shows that *in*[*B,e*] is eventually set to 0.

Theorem The algorithm requires at most O(*mr*) elementary steps, where *m* is the number of expressions and *r* is the number of arcs.

Proof The "**for each** basic block" loop takes O(*n*) steps. The **while** loop considers each arc once. Therefore O(*r*). Outermost loop is *m* iterations and $r \geq n\text{-}1$. Therefore O(*mr*)

Here is another algorithm (Kildall's):

```
begin
    $in[1]=0^m$
    $in[2:n]=1^m$
    for each C in SUC(1) do
        PILE += {(C,egen[1,:])}
    od
    for each node C do
        T(:) = $egen[C,:]$ or $\neg$ $ekill[C,:]$
        for each B in SUC(C) do
            PILE += {B,T}
        od
    od
    while PILE $\neq \emptyset$ do
        (C,T) from PILE
        if ($\neg$T and $in$[C]) $\neq 0^m$ then
            $in$[C,:] = $in$[C,:] and T(:)
            T(:)=($in$[C,:] and $\neg ekill$[C,:]) or $egen$[C,:]
            PILE += {(D,T) | D in SUC(C)}
        fi
    od
end
```

Round-Robin Version of the Algorithm

This is the most straightforward version:

```
begin
    $in[S] = \emptyset$
    $out[S] = egen[S]$
    for each block $B \neq S$ do
        $out[B] = U - ekill[B]$
    od
    chage = true
    while change do
        change = false
        for each block $B \neq S$ do /* in rPOSTORDER */
            $in[B] = \bigcap_{C \in PRED[B]} out[C]$
            oldout= $out[B]$
            $out[B] = egen[B] \cup (in[B]- ekill[B])$
            if $out[B] \neq$ oldout then
                change = true
            fi
        od
    od
end
```

In bit notation

```
begin
    $in[S,:] = 0^m$
    for each block B ≠ S do
        $in[B,:] = 1^m$
    od
    change = true
    while change do
        change = false
        for each block B ≠ S in rPOSTORDER do
            new = $\bigwedge_{C \in PRED[B]} egen[C,:] \vee (in[C,:] \wedge \neg ekill[C,:])$
            if new ≠ in[B,:] then
                $in[B,:]$ = new
                change = true
            fi
        od
    od
end
```

To study the complexity of the above algorithm we need the following definition.

Definition Loop-connectedness of a reducible flow graph is the largest number of back arcs on any cycle- free path of the graph.

Lemma Any cycle-free path in a reducible flow graph beginning with the initial node is monotonically increasing in rPOSTORDER.

Lemma The while loop in the above algorithm is executed at most $d+2$ times for a reducible flow graph, where d is the loop connectedness of the graph.

Proof A 0 propagates from its point of origin (either from the initial node or from a “kill”) to the place where it is needed in $d+1$ iterations if it must propagate along a path P of d back arcs. One more iteration is needed to detect that there are no further changes.

Theorem If we ignore initialization, the previous algorithm takes at most $(d+2)(r+n)$ bit vector steps; that is $O(dr)$ or $O(r^2)$ bit vector steps.

Global Common Subexpression Elimination

A very simple algorithm is presented in the book by Aho Sethi and Ullman:

Algorithm GCSE: Global Common Subexpression Elimination

Input: A flow Graph with available expression information.

Output: A revised Flow graph

Method:

```
begin
    for each block B do
        for each statement r in B of the form x=y op z
        with y op z available at the beginning of B and
        neither y nor z defined before r in B do
            for each block C computing the expression
            y op z reaching B do
                let t be the last statement in C of the
                form w=y op z
                replace t with the pair
                    u=y op z
                    w=u
            od
            Replace t with x = u
        od
    od
end
```

To find the statements *t*, the algorithm searches backwards in the flow graph. We could compute something equivalent to use-definitions chains for expressions, but this may produce too much useless information.

Copy propagation can be applied to the program to eliminate the `w = u` statement.

The algorithm can be applied several times to remove complex redundant expressions.

Copy propagation

To eliminate copy statements introduced by the previous algorithm we first need to solve a data flow analysis problem.

Let *cin*[B] be the set of copy statements that (1) dominate B, and (2) their **rhs**s are not rewritten before B.

out[B] is the same, but with respect to the end of B.

cgen[B] is the set of copy statements whose **rhs**s are not rewritten before the end of B.

ckill[B] is the set of copy statements not in B whose rhs or lhs are rewritten in B.

We have the following equation:

$$cin[B] = \bigcap_{C \in PRED[B]} cgen[C] \cup (cin[C] - ckill[C])$$

Here we assume that *cin*[S]=∅

Using the solution to this system of equations we can do copy propagation as follows:

Algorithm CP: Copy Propagation

Input: A flow Graph with use-definiton chains, and *cin*[B] computed as just discussed above.

Output: A revised Flow graph

Method:

```
begin
    for each copy statement t: x=y  do
        if for every use of x,  B:
                t is in cin[B] and
                neither x nor y are redefined within B before the use
                then
                    remove t and
                    replace all uses of x by y.
        fi
    od
end
```

Detection of Loop-Invariant Computation

Input. A loop L. Use-definition chains

Output. The set of statements in L that compute the same value every time they are executed

Method

```
begin
    Mark "invariant" those statements whose operands are
    all either constant or have their reaching definitions
    outside L
    while at least one statement is marked invariant do
        mark invariant those statements with the following
        property: Each operand is either (1) constant, or
        (2) have all their reaching definitions outside L, or
        (3) have exactly one reaching definition, and that
        definition is a statement in L marked invariant.
    od
end
```

Once the "invariant" marking is completed statements can be moved out of the loop. The statements should be moved in the order in which they were marked by the previous algorithm. These statements should satisfy the following conditions:

1. All their definition statements that were within L should have been moved outside L.
2. The statement should dominate all exits of L
3. the lhs of the statement is not defined elsewhere in L, and
4. All uses in L of the lhs of the statement can only be reached by the statement.