Copy Propagation

When we eliminated global CSE's we introduced *copy* statements of the form `A := B`. There are many other sources for copy statements, such as the original source code and the intermediate code generator.

Many copy statements can be eliminated. Suppose we have a use of `A` in a statement like

s: `X := A + Y`

and on every path from the start node to s we encounter a copy `A := B` or `B := A`. Moreover, after the last such occurrence, neither `A` nor `B` is redefined.

Then `A` and `B` have the same value at s, and we may replace s by

`X := B + Y`

This change makes no apparent difference, but in some cases it will make the copy statement(s) `A := B` useless.

- Useless statements can be eliminated; we study this question when we take up the "reached uses" DFA problem.

We say copy `X := Y` *reaches* point p if every path from the start node to p has an occurrence of `X := Y`, and there are no defs of `X` or `Y` after the last occurrence.

Define IN(B) to be the set of copies that reach the beginning of block B and OUT(B) to be the set of copies that reach the end of B.

KILL Sets

KILL(B) is the set of copies `X := Y` such that either `X` or `Y` may be redefined in B.

GEN Sets

GEN(B) is the set of copies `X := Y` appearing in B, with no possibility that either `X` or `Y` is subsequently redefined in B.

- The equations for copy propagation are exactly the same as for AE's. Their solution is likewise the same.
- Whether `X := Y` or `Y := X` reaches a point, the conclusion that `X` and `Y` have the same value can be made. The only reason for distinguishing between copies in the two directions is that we want to know whether to replace `X` by `Y` or vice-versa to help eliminate copies.

Live Variables

When we generate object code for a block, we may keep certain variables in registers during execution of the block. If we have not given that variable a register "permanently," then we must store it on exit from the block.

- If it were known that the variable was never subsequently used before recomputation, then we could save the store instruction.

We call a variable `X` *live* at point p if there is some path from p along which the value of `X` may be used before redefinition. Otherwise, `X` is *dead.*

Equations for Live Variable Analysis

The equations are similar to RD's, but data flows "backwards."

- While in RD's, we propagated definitions forward as far as they could go, for LV's we propagate uses backwards until they encounter an unambiguous definition.

Let IN(B) be the set of variables live at the beginning of B and OUT(B) the set of variables live at the end of B.

KILL Sets

KILL(B) = the set of variables that are unambiguously defined in B prior to any possible use in block B.

- To be conservative, we have to regard ambiguous uses like `A := *P` as a possible use of any `X` that `P` could point to.

GEN Sets

GEN(B) = the set of variables that may be used, possibly ambiguously, before any unambiguous redefinition in block B.

- Note that our definitions of GEN and KILL are motivated by the assumption that it is safe to assume a variable is live when it isn't, but unsafe to assume it is dead when it isn't.

Transfer Equations

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

Confluence Equations

- Really "divergence rules," since they tell what happens when flow leaves a block and goes in two or more different directions.

A variable is live coming out of a block if it is used before redefinition along some path emanating from that block, that is, if the variable is live entering some successor block. Thus:

$$\mathrm{OUT}(B) = \cup_{\text{succ. } S \text{ of } B}\ \mathrm{IN}(S)$$

Algorithm for Computing Live Variables

As always, the equations do not have a unique solution.

- We want the smallest, so we do not say a variable is live unless we can actually find a path along which it might be used.

We start by assuming nothing is live on exit from any block and apply the equations until no more changes to the IN's can be made. The process must stop because we are only adding variables to sets.

```
for all blocks $B$ do
    IN($B$) := GEN($B$);

while changes to some IN occur do
    for each block $B$ do begin
        OUT($B$) :=
            $\cup_{\text{succ. } S \text{ of } B}$ IN($S$);
        IN($B$) :=
            (OUT($B$) $-$ KILL($B$))
            $\cup$ GEN($B$)
    end
```

Data Structures for Live Variables

The ideas are the same as before. Identify variables with positions in a bit vector, using a table of pointers (to the symbol table, this time).

Sets are again represented by bit vectors and the set operations by logical operations.

Reached Uses

A use of variable `A` is *reached* by a def of `A` if there is a path from the def to the use along which `A` is not unambiguously redefined.

```
A := ···
                No unambiguous
                defs of A
X := A + Y
```

Ambiguous Uses

Like defs, a use can be *ambiguous*, e.g., a procedure call or a pointer reference

```
A := *P
```

could use anything the procedure can access or the pointer `P` could point to.

Useless Statements

An important application of "reached uses" is in detecting useless statements. An assignment is *useless* iff it has no reached uses.

- Typically, useless statements are created by the code optimizer itself, e.g., when it applies copy propagation information to replace uses of `A` by uses of `B`, where `A := B` is the copy statement in question.

Propagating Uses Inside Blocks

If we know what uses are reached by the end of block B, we can find the uses reached by statements within B.

1. A def of `A` within block B reaches those uses of `A` that the end of B reaches, provided that there is no subsequent, unambiguous def of `A` within B.
2. In any event, a def of `A` within B also reaches those uses of `A` following it in B, up to the next unambiguous def of `A`.

UD and DU Chains

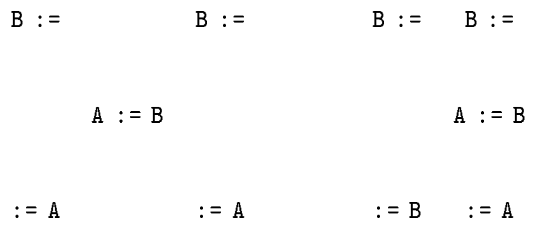
Given reached uses and reaching definitions, we can construct for each def a list of its reached uses, called a *def-use chain.*

Similarly, attach to each use a *use-def chain* of all its reaching defs.

- DU + UD chains make it easy to update def/use information when we eliminate statements.

Example

Consider the changes in du- and ud- links needed when one of the reached uses of a copy statement `A := B` has the use of `A` changed to a use of `B`.



Equations for Reached Uses

The equations are essentially the same as for LV's.

Uses are represented by pairs (s, A), giving a statement s and a variable `A` used (or possibly used) by s. Thus,

s: `X := Y + Z`

yields (s, Y) and (s, Z).

KILL Sets

KILL(B) = $\{(s, A)$ | block B contains an unambiguous def of `A`$\}$.

GEN Sets

GEN(B) = $\{(s, A)$ | statement s is in B, variable `A` is used by s (possibly ambiguously), and there are no unambiguous defs of `A` prior to $s\}$.

Transfer and Confluence Equations

The transfer and confluence equations for reached uses is exactly the same as for LV's. So is the method of solution.

Summary

We can classify DFA problems by

1. The direction of data flow (forward or backward).
2. The confluence operator (union or intersection).

	Union	Intersection
Forward	Reaching Defs. Undefined Vars.*	Available Exprs. Copy Propagation
Backward	Live Vars. Reached Uses	Very Busy Exprs.**

* Trick: Create a dummy def of every variable in a dummy start node prior to the real start node.

** Defined in text. Expression $X+Y$ is *very busy* at point p if every path from p evaluates $X+Y$ before redefinition of `X` or `Y`. If so, we can save code space by evaluating $X+Y$ at p.

Improvements to DFA Algorithms

Two possible improvements to the simple iterative DFA algorithm.

1. Using depth-first ordering for speedup.
2. Interval analysis: structure-based DFA.

Depth of Flow Graphs

Given a depth-first spanning tree for a flow graph, the *depth* of that flow graph (with respect to that DFST) is the maximum number of backward (head is an ancestor of tail) edges along any acyclic path.

The concept of a "reducible" flow graph will turn out to be very important; it corresponds roughly to the notion of a "structured" program, one in which there are no uncontrolled goto's that jump into the middle of loops.

- If the flow graph is reducible, then "backward" = "back" (head dominates tail), and the depth is independent of the actual DFST chosen.
- An observed fact is that the average depth of flow graphs is very low.

Example

3 nested while-loops depth = 3   3 nested repeat-loops depth = 1

DFA by Propagation along Acyclic Paths

Many common DFA problems involve the propagation of data that may be restricted to propagation along acyclic paths.

Example: Reaching Definitions

If a def reaches a point, then it does so along some acyclic path.

Example: Available Expressions

If an expression is not available at point p, there is some acyclic path to p that demonstrates this fact. Either the expression is never computed, or it is killed after its last evaluation.

Suppose we perform a forward DFA (e.g., reaching defs) by visiting the nodes in depth-first order on each pass. Then except for the backward edges, we update IN and OUT for the node at the tail of each edge before the node at the head.

Thus, information can propagate along any path of nonbackward edges in one pass.

- Only when the path requires a backward edge does the propagation stop and wait for the next pass.

Example

$$\underbrace{1 \rightarrow \cdots \rightarrow 27}_{\text{Pass 1}} \rightarrow \underbrace{13 \rightarrow \cdots \rightarrow 19}_{\text{Pass 2}} \rightarrow \underbrace{8 \rightarrow \cdots \rightarrow 42}_{\text{Pass 3}}$$

As a result, if propagation along acyclic paths is sufficient, and the depth of the flow graph is d, then $d+1$ passes suffice for DFA.

- We may need pass $d+2$ to know that we have converged.

Backward DFA

Similarly, if we are doing backward DFA, e.g., LV's, we may choose the reverse of depth-first order as the order in which we visit the nodes on each pass.

If we do so, then each nonbackward edge has its head visited before its tail. We can thus propagate information backward along any path of nonbackward edges in one pass.

- If propagation along acyclic paths is sufficient, then backwards DFA problems also converge in $d+1$ passes if the flow graph has depth d and the reverse of DF order is used.

Interval Analysis

- Uses structure of flow graph.
- Especially useful when the source language allows no gotos and the structure can be determined from the source (for-, while-, etc.).

We compute transfer functions for collections of nodes, working up the hierarchy. Finally, we apply the transfer functions computed for the entire flow graph to "zero" (no information) and transmit the results down the hierarchy to compute IN and OUT for every node.

- We shall define intervals principally for historical reasons, and then give an equivalent, simpler approach.

Regions

A *region* in a flow graph is any set of nodes with a header that dominates all nodes in the set.

- Every loop, as we defined the term, is a region, but not vice-versa, since there may be no graph-theoretic cycles in a region.
- For example, every node by itself is a region.

Intervals

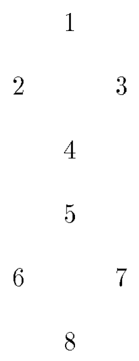
The *interval* with header h in a given flow graph is defined as follows.

1. h is in I.
2. If all predecessors of node n (not the start node) are in I, then n is also in I.

- Note that if there are several candidates for n, the order in which they are chosen doesn't matter; the others will remain candidates.

3. Nothing else is in I.

Example



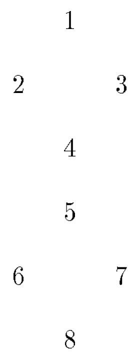
For instance, $I(1)$, the interval with header 1, consists of 1 and 3. We start with 1. We add 3 because its only predecessor, 1, is in $I(1)$. We cannot add 2 because 4 isn't in; we can't add 4 because 6 isn't in, and so on.

Interval Partitions

1. Start by constructing the interval whose header is the start node.
2. Until all nodes have been assigned an interval, find some node not yet assigned, but one of whose predecessors has been assigned.

- Such a node can never be part of another's interval, so make it a header and compute its interval.
- Each interval is a region dominated by its header.
- The collection of intervals does not depend on the order in which headers are chosen in (2).

Example



Interval Graphs

Having partitioned a graph into intervals, we can construct the *interval graph* for G, denoted $I(G)$, by making one node for each interval.

Place an arc from J to K if some node in interval J has an arc to the header of K.

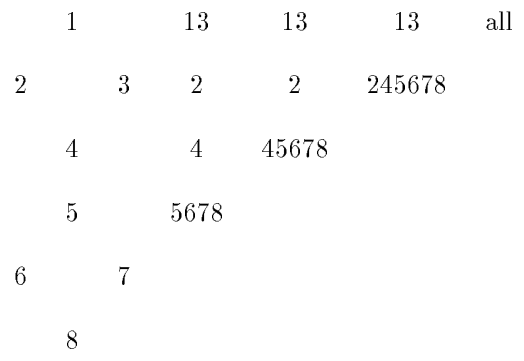
- Note that an arc from J cannot enter any node of K besides the header.

Avoid arcs from a node to itself, however.

The start node of $I(G)$ is the interval containing the start node of G.

The graph $I(G)$ can itself have intervals computed, the graph $I\big(I(G)\big)$ computed, and so on.

Example



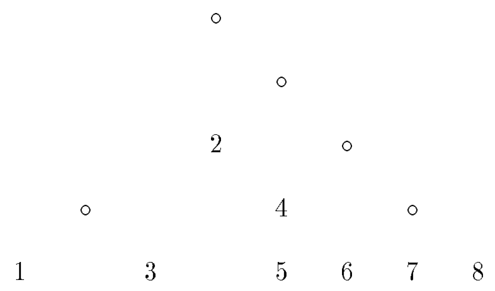
Hierarchy of Intervals

In this manner we group nodes into a hierarchy. That is, each interval of the original flow graph represents a set of nodes in the obvious way.

An interval of $I(G)$ represents the nodes of the original flow graph that the interval's nodes represent, and so on.

Example

The hierarchy implied by our running example is:



Reducible Flow Graphs

If the sequence of interval graphs leads to a graph of a single node, we say the graph is *reducible.*

Another equivalent definition of "reducible" is that the back edges (head dominates the tail) are exactly the edges whose heads precede their tails in DF order.

- Recall that "structured" flow graphs have this property, so we expect the interval graph sequence to lead to a single node almost always.

Node Splitting

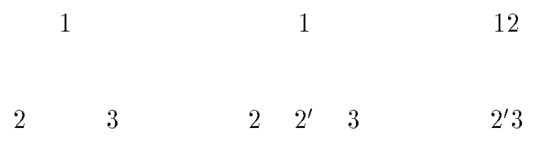
We can guarantee that the sequence of interval graphs can be completed even if the original graph is not reducible.

Take an *irreducible* graph (one that is its own interval graph) and find some node other than the header, say node n.

- Node n will surely have more than one predecessor, else it could have been put in its predecessor's interval.

Replace n by $\{n_1, \ldots, n_k\}$ if n has k predecessors, say $p_1, \ldots, p_k$. Let the arc from p_i to n now go to n_i. The successors of n_i are all the successors of n.

Example



Since each of the introduced nodes has a unique predecessor, a pass of interval analysis will result in each of them becoming part of someone else's interval. Thus, the process of splitting a node and doing interval analysis results in a graph of fewer nodes.

T1-T2 Analysis

There are two simple transformations that are together equivalent to analysis by intervals.

T1

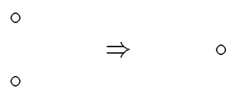
Remove an arc from a node to itself.

$$\circ \qquad \Rightarrow \qquad \circ$$

T2

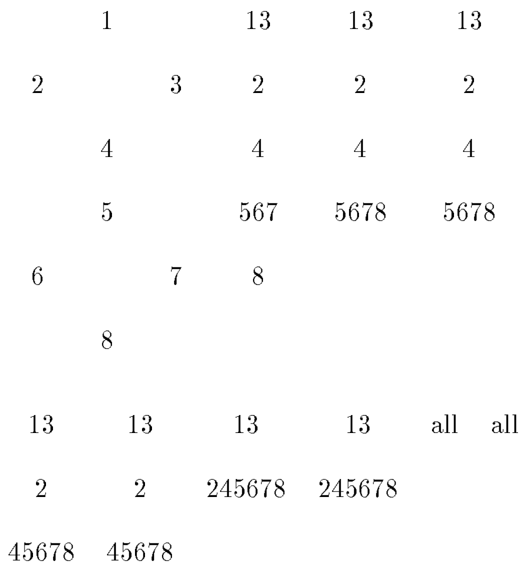
Combine a node (other than the start node) with its lone predecessor.

- The latter *consumes* the former.



- T1, T2 always produce a unique result when they are applied to a graph until no more applications are possible.
- The result is the same as that obtained by applying interval analysis until an irreducible graph is reached.
- Thus, another definition of "reducible flow graph" is one that can be reduced to a single node by T1 and T2.

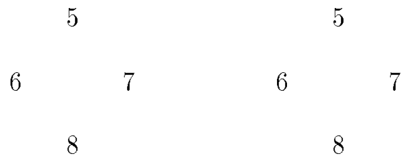
Example



- Each node during a T1-T2 reduction represents a collection of nodes and edges of the original graph.
- Each edge represents a collection of edges.
- The nodes and edges represented by a single node always form a region, except for the possibility that some back edges to the header of the region are missing.

Example

The two nodes labeled 5678 in the previous example respectively represent:



Generalized Transfer Functions

Let us consider a DFA problem like reaching definitions. Suppose we have reduced a flow graph by T1 and T2 to get a hierarchy of regions (possibly with some back edges to the header removed) for the entire flow graph.

We may associate with each region a collection of transfer functions that express what happens when we go from the header to each of the nodes in the region.

Example: RD's

If f is the transfer function associated with node n and region R, then $f(S)$ is the set of defs that reach the end of n given that

1. S is the set of defs reaching the header of R, and
2. Control stays within the nodes and edges of R once reaching the header.

It is important to observe that such functions are of the form

$$f(S) = (S - K) \cup G$$

- K represents defs that will be "killed" along any path to the end of n from the header.
- G represents defs in the region that appear along one or more paths to n within the region and are not subsequently killed.

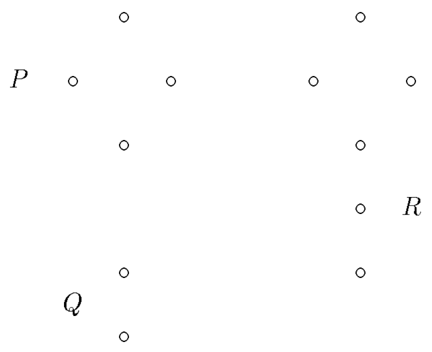
Hierarchical Computation of Transfer Functions

Basis: If the region consists of a single node B, then

$$f_B(S) = (S - \text{KILL}(B)) \cup \text{GEN}(B)$$

Induction: Part I

Suppose we form region R from regions P and Q by T2, where the node representing P consumes that representing Q.



Then for n in P: $f_{R,n}(S) = f_{P,n}(S)$.

If node n is in Q, let $g(S)$ be the union of $f_{P,m}(S)$ for all nodes m in P that are predecessors of the header of Q.

- Recall no other node in Q can be a successor of a node in P.

Then $f_{R,n}(S) = f_{Q,n}\big(g(S)\big)$.

This formula requires taking the union and composition of transfer functions. Since transfer functions are of the form $(S - K) \cup G$, we can perform these operations on functions $f_1(S) = (S - K_1) \cup G_1$ and $f_2(S) = (S - K_2) \cup G_2$ by:

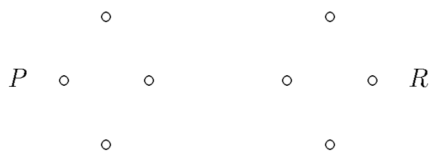
$$f_1 \cup f_2 = \big(S - (K_1 \cap K_2)\big) \cup (G_1 \cup G_2)$$
$$f_2(f_1) = \big(S - (K_1 \cup K_2)\big) \cup (G_2 \cup (G_1 - K_2))$$

Note the result is of the desired form in both cases.

Induction: Part 2

Now suppose region R is formed from a region P because the node corresponding to P has an application of T1 applied to it.

In terms of the original graph, application of T1 means that one or more back edges to the header are made part of the region R, while they were not part of P.



Let $g(S)$ be the union of $f_{P,m}(S)$ taken over each node m in P that is a predecessor of the header (*latch*) along an arc not in P. Then

$$f_{R,n}(S) = f_{P,n}\big(g(S) \cup S\big)$$

for all n in P.

In the union, the term $g(S)$ accounts for defs that reach a latch, are transferred to the header, and not killed from the header to n. The term S accounts for defs reaching the header from outside.

- Note that $g(S)$ could be replaced by $g(\emptyset)$.

Interval Analysis Algorithm for RD's

Inductively, compute for larger and larger regions the transfer functions associated with that region for all nodes in the region.

If the graph is reducible (or can be made so by node splitting), the single node to which it is reduced represents the entire graph as one region R.

It follows that $f_{R,n}(S)$ is the set of defs that reach the end of n starting with set S at the beginning of the start node of the flow graph.

Since no defs are available before the start, S should be $\emptyset$ here. That is,

$$\text{OUT}(n) = f_{R,n}(\emptyset)$$

We may then compute IN's by taking the union of the OUT's of the predecessors.