1 Control flow graphs for program analysis

We have seen that we can easily convert the lowered IR representation into a control flow graph (CFG) in which each node n has one of the following contents:

Node contents	Lowered IR equivalent
An assignment $x := e$	$\mathbf{MOVE}(\mathbf{TEMP}(x), e)$
A memory store $[e_1] := e_2$	$\mathbf{MOVE}(\mathbf{MEM}(e_1), e_2)$
A conditional branch **if** e	$\mathbf{CJUMP}(e, e_2)$
A start node **START**.	—
An exit node **EXIT**.	—

Expressions e can be any of the following, except that function calls can appear only at top level:

$$
\begin{aligned}
e &::= e_1\ OP\ e_2 \\
&\mid f(e_1, \ldots e_n) \\
&\mid [e] \\
OP &::= + \mid - \mid * \mid / \mid \texttt{mod} \mid \texttt{lshift} \mid \texttt{rshift} \mid \ldots
\end{aligned}
$$

Other nodes such as **LABEL** and **JUMP** are represented by the graph structure. (Assuming that we can identify all the possible values of e as nodes in the CFG, we can treat a node $\mathbf{JUMP}(e)$ where e is a complex expression as a fancy kind of `if` node in which there are any number of exit edges.) It is handy for describing some analyses to have distinguished **START** and **EXIT** nodes, though often these are omitted when drawing CFGs.

A node has zero or more entry edges and either one or two exit edges; only `if` has more than one exit edge, as depicted in Figure 1.

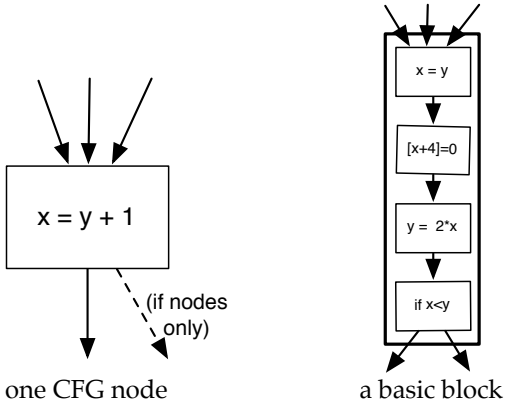

Figure 1: CFG nodes

We will often find that program CFGs have linear sequences of several nodes where each node except the first has a single entry edge, and each node except that last has a single exit edge. Such a sequence of nodes is a *basic block*. It is possible to do program analysis over a CFG of basic blocks instead of over individual nodes. This complicates the analysis, but gives the same result. For some analyses, it is possible to speed up analysis somewhat by performing it at the granularity of basic blocks.

2 Live variable analysis: recap

In a dataflow analysis, we associate information with each *program point*, where program points are points in between the execution of nodes: at the beginning and end of each node. We write $in(n)$ to represent the information at the program point before the node is executed, and $out(n)$ to represent the information just after it is executed.

For live variable analysis, the information associated with each program points is the set of live variables: a set containing each variable whose value might be used before the value of the variables is changed.

2.1 Dataflow equations

A variable is live on exit from a node if it is live on entry to any successor node . And it is live on entry to the node if it is used by the node, or if it is live on exit from the node and this node does not redefine the value of the variable. These observations are captured by the following *dataflow equations*. Note that to denote that node n' is a successor of n, we write either $n' \in succ(n)$, or $n' \succ n$.

$$\begin{aligned} in(n) &= use(n) \cup (out(n) - def(n)) \\ out(n) &= \bigcup_{n' \succ n} in(n') \end{aligned}$$

The functions $use(n)$ and $def(n)$ are defined according to the following table, where $use(e)$ refers to the set of variables used in the expression e:

n	$use(n)$	$def(n)$
$x = e$	$use(e)$	x
$[e_1] = e_2$	$use(e_1) \cup use(e_2)$	$\emptyset$
`if` e	$use(e)$	$\emptyset$
START	$\emptyset$	$\emptyset$
EXIT	$\emptyset$ or `rv`	$\emptyset$

3 Worklist algorithm

How do we solve the dataflow equations? A first step is to substitute the definition of $out(n)$ into that of $in(n)$, eliminating half the equations:

$$in(n) = use(n) \cup (\bigcup_{n' \succ n} in(n') - def(n))$$

If we want to use this dataflow equation to define the value of $in(n)$, it is clear that the information is flowing backward along the arrows in the CFG. For this reason, we say that live variable analysis is a *backward analysis*.

The usual way to solve dataflow equations is the *worklist algorithm*, which uses a FIFO queue called the *worklist* to keep track of nodes whose equations might not be satisfied at any given step. The algorithm is as follows for a backward analysis such as live variable analysis:

1. Initialize the worklist to contain all nodes.

2. Initialize the value of $in(n)$ to some initial value (for live variable analysis, $\emptyset$).

3. While the worklist contains some node n:

   - Remove n from the worklist.
   - Set the value of $in(n)$ using the dataflow equation. For live variable analysis:

$$in(n) := use(n) \cup (\bigcup_{n' \succ n} in(n') - def(n))$$

   - Push all predecessors of n onto the worklist.

Now let's look at why this algorithm works. Each node has its own dataflow equation, so for N total nodes, there are N dataflow equations. The worklist algorithm simply chooses to apply these equations iteratively in a particular order to update the value of $in(n)$, until convergence. What we haven't explained is why this order of application ends in the correct result.

3.1 Monotonicity

The dataflow equation for a given node n has an interesting monotonicity property: adding more elements to $in(n')$ for its successor nodes n can only add elements to (or have no effect on) the value of $in(n)$ according to the equation. Think about an iteration of the algorithm that updates $in(n)$. There was some previous iteration that updated $in(n)$ (which might be the original initialization to $\emptyset$). Suppose all changes that have occurred to the values of $in(n')$ for successor nodes n' have been to add elements. In that case this update, if it has any effect, must also add elements.

Clearly the very first iteration of the worklist algorithm can only add elements (or have no effect), since $in(n) = \emptyset$. Therefore the second iteration of the algorithm can also only add elements, since all prior iterations have only added elements. Inductively, we can see that every iteration of the worklist algorithm, when using the live variable analysis dataflow equation, can only add elements.

Therefore a given set $in(n)$ only increases in size during the execution of the worklist algorithm.

3.2 Complexity

There is a finite number of variables (call it V), and the set $in(n)$ can only grow, so the maximum number of times it can change during execution of the algorithm is V. How many times can the main loop of the algorithm execute? Once for each time a node is pushed onto the worklist. How many times can a node be pushed onto the worklist? Once at the beginning, and once for each change to one of its successor nodes. Since a node has at most two successors, this means at most $2V + 1$ pushes. Therefore the running time is $O(VN)$, or $O(N^2)$. With some reasonable assumptions about the structure of control flow graphs for real programs, this can be lowered to $O(dN)$ where d corresponds to the loop nesting depth of the code, and is typically no more than 4 or so.

3.3 Correctness

We've just seen that the algorithm must terminate. If it does terminate, we would like to know that all the dataflow equations are satisfied. To see this, notice that the worklist algorithm maintains a loop invariant: every node that is not on the worklist has its equation satisfied. Clearly this invariant holds at the beginning because all nodes are on the worklist. Each time that a node n is removed from the worklist, it is because its equation is satisfied by updating $in(n)$. And each time this is done, the nodes whose equations might have become unsatisfied (the predecessors) are pushed onto the worklist.

Therefore, when the worklist is empty, all dataflow equations are satisfied.

4 Available copies analysis

You may be suspecting that we can generalize the live variable dataflow analysis into a general framework for dataflow analysis. Before we try to do that, let's look at another dataflow analysis so we can identify what is in common. We will consider an analysis called *available copies*, which keeps track of variable copies. This is useful for doing *copy propagation* optimizations. It is really a special case of a more general analysis called *available expressions*, which we will see later.

4.1 Copy propagation

The idea of this optimization is that we replace variables with other variables known to contain the same information. This means we aren't wasting registers on redundant information. For example, in the code below, the variables `x` and `y` hold the same value after the assignment. Assuming there are no assignments to either variable between that point and the assignment to `z`, we can replace the use of `x` with `y`, as shown on the right. If that transformation means the variable `x` is no longer live, the assignment `x=y` becomes dead code and can be removed.

```
x = y                    x = y
⋮              ⟹        ⋮
z = 2*x + 1              z = 2*y + 1
```

4.2 Dataflow values

The information associated with each program point will be a set of equalities known to hold between different variables. Such a set might look like $\{x_1 = y_1, x_2 = y_2, \ldots, x_n = y_n\}$. In addition, the $y's$ are variables that were assigned their values earlier than the corresponding $x's$. We can use a set of equalities like this to determine whether two variables are definitely known to be equal, which is the information copy propagation needs.
[1]

4.3 Dataflow equations

An equation only holds on entry to a node if it holds on exit from *all* predecessor nodes. Therefore, we have the following equation:

$$in(n) = \bigcap_{n' \prec n} out(n')$$

An equation holds on exit from a node n if either it is established by the node (we use $gen(n)$ to represent the equalities introduced by node n), or if the equation held on entry to the node, but the node makes the equation untrue (we use $kill(n)$ to represent such equalities). The equation for $out(n)$ is therefore:

$$out(n) = gen(n) \cup (in(n) - kill(n))$$

The functions $gen()$ and $kill()$ are defined by the following table:

[1] A better but more complicated representation is to keep track of the equivalence classes of variables, which allows more equalities to be proved. This can be done by ensuring that none of the variables y_i appears on the LHS of an equality. Then each variable y_i stands for the equivalence class of variables that are known to be copies; path compression is used to support efficient updating and testing of equality. This approach is equivalent to *value numbering*.

n	$gen(n)$	$kill(n)$
$x = y$	$\{x = y\}$	$x = z, z = x(forallz)$
$x = e$where $e \neq y$	$\emptyset$	$x = z,$
$[e_1] = e_2$	$\emptyset$	$\emptyset$
`if` e	$\emptyset$	$\emptyset$
START	$\emptyset$	$\emptyset$
EXIT	$\emptyset$ or `rv`	$\emptyset$

4.4 Worklist algorithm

We can see that available copies is a *forward analysis* because the value at entry to a node depends on the predecessor nodes. We update the worklist algorithm given earlier by using predecessors where it used successors, and vice versa, and by using $out(n)$ where it used $in(n)$:

1. Initialize the worklist to contain all nodes.
2. Initialize the value of $out(n)$ to some initial value (for available copies variable analysis, the set of all possible equalities).
3. While the worklist contains some node n:
   - Remove n from the worklist.
   - Set the value of $out(n)$ using the dataflow equation:

$$in(n) := \bigcap_{n' \prec n} out(n')$$
$$out(n) := gen(n) \cup (in(n) - kill(n))$$

   - Push all successors of n onto the worklist.

This analysis has the same monotonicity property as live variable analysis, but the space of possible values is larger. Therefore the algorithm terminates but can take asymptotically longer.

5 Dataflow analysis framework

We can characterize both of these analyses within a common framework. This has the advantage that it will allow us to quickly decide whether a given analysis is guaranteed to complete, what its complexity is, and whether it always computes the best solution possible. In addition, a common framework for dataflow analysis enables us to implement a general algorithm for doing analyses in the compiler, rather than reimplementing each analysis from scratch.

A dataflow analysis framework has four components: the direction of analysis (forward or backward), the values being propagated, transfer functions for each of the nodes, and a meet operator.

5.1 Dataflow values

A key component of a dataflow analysis framework is the set of values that the analysis is computing on. In live variable analysis, the values were themselves sets of variables. In available copies, the values were sets of equalities. Let L be the set of all values that can be assigned to a program point. We will use ℓ to denote a single value contained in L.

What do dataflow values mean? We can usefully think of them as representing some proposition that must hold at the program point they are attached to. (We can also think of them as representing propositions that *may* hold at the program point, but this is just the same thing as saying the negation of the proposition

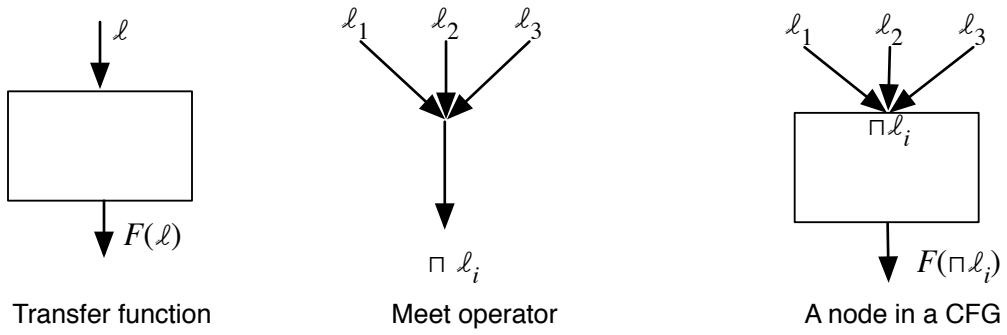

Figure 2: Dataflow analysis framework components

must hold.) For example, in live variable analysis, the meaning of the set of variables attached to a program point is that all the variables in the set must be live at that point.

In the space of values there is one value that conveys the greatest possible information. We will denote this value by the symbol $\top$. In live variable analysis, the greatest information is conveyed by the empty set $\emptyset$. It means no variables are live, and therefore that the entire program is dead code.

5.2 Transfer functions

The second part of a dataflow analysis is a set of *transfer functions* that describe how dataflow values are transformed by a node. For a forward analysis, the transfer function defines how $out(n)$ depends on $in(n)$. For a backward analysis, it's the reverse.

5.3 Meet operator

The final part of a dataflow analysis which defines how to combine values from multiple incoming edges. As depicted in the figure, supposing we have propositions corresponding to l_1, l_2, and l_3 on various edges. At a common program point where all these edges meet, we don't know which edge we came in on, so at most we know the disjunction (or) of the three propositions. The *meet operator* $\sqcap$ defines how to construct the value corresponding to this disjunction. For live variable analysis, the meet operator is union; for available copies analysis, it is intersection.

5.4 Worklist algorithm

We can now see that both versions of the worklist algorithm seen so far are just instances of a more general algorithm. We are trying to solve for the dataflow value for each node n. Without loss of generality, we will consider forward analysis (for backward analysis, just turn all the arrows around). When we start the worklist algorithm, we initialize $in(n)$ to $\top$ for each n. The dataflow equation that is applied to update $in(n)$ at each iteration is $in(n) = F(\sqcap_{n' \prec n}\ in(n'))$.

6 Summary

We've seen that a dataflow analysis framework can be characterized as a four-tuple $(D, L, \sqcap, F)$: the direction of analysis D, the space of values L, transfer functions F_n, and the meet operator $\sqcap$. We're not yet guaranteed that the worklist algorithm works, however.

Next time we'll show that with some reasonable conditions on L, F_n, and $\sqcap$, the worklist algorithm is correct and efficient, and often (but not always) computes the best possible answer to the equations.