that combines the best properties of the hashing and global approaches and is more effective than both of them. (See Section 12.8 for citations.)

12.5 Copy Propagation

Copy propagation is a transformation that, given an assignment $x \leftarrow y$ for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y.

From here on, we generally need to represent the structure of a procedure as an array of basic blocks, each of which is an array of instructions. We use the variable `nblocks` and the arrays `ninsts[1..nblocks]` and `Block[1..nblocks][..]`, declared as

```
nblocks: integer
ninsts: array [1..nblocks] of integer
Block: array [1..nblocks] of array [..] of Instruction
```

where `Block[i]` consists of instructions `Block[i][1]` through `Block[i][ninsts[i]]`, to do so.

Before proceeding to discuss copy propagation in detail, we consider its relationship to register coalescing, which is discussed in detail in Section 16.3. The two transformations are identical in their effect, as long as optimization is done on a low-level intermediate code with registers (symbolic[1] and/or real) in place of identifiers. However, the methods for determining whether register coalescing or copy propagation applies to a particular copy assignment are different: we use data-flow analysis for copy propagation and the interference graph for register coalescing. Another difference is that copy propagation can be performed on intermediate code at any level from high to low.

For example, given the flowgraph in Figure 12.23(a), the instruction `b ← a` in block `B1` is a copy assignment. Neither `a` nor `b` is assigned a value in the flowgraph following this instruction, so all the uses of `b` can be replaced by uses of `a`, as shown in Figure 12.23(b). While this may not appear to be a great improvement in the code, it does render `b` useless—there are no instructions in (b) in which it appears as an operand—so dead-code elimination (see Section 18.10) can remove the assignment `b ← a`; and the replacement makes it possible to compute the value assigned to `e` by a left shift rather than an addition, assuming that `a` is integer-valued.

Copy propagation can reasonably be divided into local and global phases, the first operating within individual basic blocks and the latter across the entire flowgraph, or it can be accomplished in a single global phase. To achieve a time bound that is linear in n, we use a hashed implementation of the table *ACP* of the available copy instructions in the algorithm in Figure 12.24. The algorithm assumes that an array of MIR instructions `Block[m][1], ..., Block[m][n]` is provided as input.

1. Symbolic registers, as found, for example, in LIR, are an extension of a machine's real register set to include as many more as may be needed to generate code for a program. It is the task of global register allocation (Chapter 16) to pack the symbolic registers into the real registers, possibly generating stores and loads to save and restore their values, respectively, in the process.

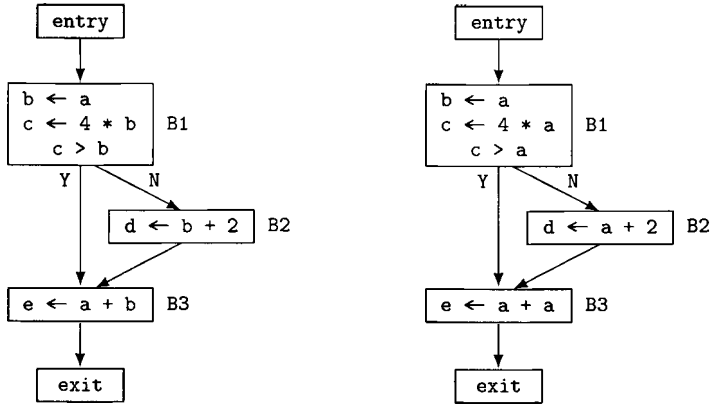

FIG. 12.23 (a) Example of a copy assignment to propagate, namely, b ← a in B1, and (b) the result of doing copy propagation on it.

```
   procedure Local_Copy_Prop(m,n,Block)
      m, n: in integer
      Block: inout array [1··n] of array [··] of MIRInst
   begin
      ACP := ∅: set of (Var × Var)
      i: integer
      for i := 1 to n do
         || replace operands that are copies
         case Exp_Kind(Block[m][i].kind) of
binexp:     Block[m][i].opd1.val := Copy_Value(Block[m][i].opd1,ACP)
            Block[m][i].opd2.val := Copy_Value(Block[m][i].opd2,ACP)
unexp:      Block[m][i].opd.val := Copy_Value(Block[m][i].opd,ACP)
listexp:    for j := 1 to |Block[m][i].args| do
               Block[m][i].args↓j@1.val :=
                  Copy_Value(Block[m][i].args↓j@1,ACP)
            od
default: esac
         || delete pairs from ACP that are invalidated by the current
         || instruction if it is an assignment
         if Has_Left(Block[m][i].kind) then
            Remove_ACP(ACP,Block[m][i].left)
         fi
         ||  insert pairs into ACP for copy assignments
         if Block[m][i].kind = valasgn & Block[m][i].opd.kind = var
            & Block[m][i].left ≠ Block[m][i].opd.val then
            ACP ∪= {<Block[m][i].left,Block[m][i].opd.val>}
         fi
      od
   end    || Local_Copy_Prop
```

(continued)

FIG. 12.24 *O(n)* algorithm for local copy propagation.

```
procedure Remove_ACP(ACP,v)
   ACP: inout set of (Var × Var)
   v: in Var
begin
   T := ACP: set of (Var × Var)
   acp: Var × Var
   for each acp ∈ T do
      if acp@1 = v ∨ acp@2 = v then
         ACP -= {acp}
      fi
   od
end    || Remove_ACP

procedure Copy_Value(opnd,ACP) returns Var
   opnd: in Operand
   ACP: in set of (Var × Var)
begin
   acp: Var × Var
   for each acp ∈ ACP do
      if opnd.kind = var & opnd.val = acp@1 then
         return acp@2
      fi
   od
   return opnd.val
end    || Copy_Value
```

FIG. 12.24 *(continued)*

As an example of the use of the resulting $O(n)$ algorithm, consider the code in Figure 12.25. The second column shows a basic block of five instructions before applying the algorithm, the fourth column shows the result of applying it, and the third column shows the value of ACP at each step.

To perform global copy propagation, we first do a data-flow analysis to determine which copy assignments reach uses of their left-hand variables unimpaired, i.e., without having either variable redefined in between. We define the set $COPY(i)$ to consist of the instances of copy assignments occurring in block i that reach the end of block i. More explicitly, $COPY(i)$ is a set of quadruples $\langle u, v, i, pos \rangle$, such that $u \leftarrow v$ is a copy assignment and pos is the position in block i where the assignment occurs, and neither u nor v is assigned to later in block i. We define $KILL(i)$ to be the set of copy assignment instances killed by block i, i.e., $KILL(i)$ is the set of quadruples $\langle u, v, blk, pos \rangle$ such that $u \leftarrow v$ is a copy assignment occurring at position pos in block $blk \neq i$. For the example in Figure 12.26, the $COPY(\,)$ and $KILL(\,)$ sets are as follows:

$$\begin{aligned} COPY(\text{entry}) &= \emptyset \\ COPY(\text{B1}) &= \{\langle \text{d}, \text{c}, \text{B1}, 2 \rangle\} \\ COPY(\text{B2}) &= \{\langle \text{g}, \text{e}, \text{B2}, 2 \rangle\} \end{aligned}$$

Position	Code Before	ACP	Code After
		∅	
1	b ← a		b ← a
		{⟨b,a⟩}	
2	c ← b + 1		c ← a + 1
		{⟨b,a⟩}	
3	d ← b		d ← a
		{⟨b,a⟩,⟨d,a⟩}	
4	b ← d + c		b ← a + c
		{⟨d,a⟩}	
5	b ← d		b ← a
		{⟨d,a⟩,⟨b,a⟩}	

FIG. 12.25 An example of the linear-time local copy-propagation algorithm.

$$
\begin{aligned}
&COPY(\texttt{B3}) &&= \emptyset\\
&COPY(\texttt{B4}) &&= \emptyset\\
&COPY(\texttt{B5}) &&= \emptyset\\
&COPY(\texttt{B6}) &&= \emptyset\\
&COPY(\texttt{exit}) &&= \emptyset\\
\\
&KILL(\texttt{entry}) &&= \emptyset\\
&KILL(\texttt{B1}) &&= \{\langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\}\\
&KILL(\texttt{B2}) &&= \emptyset\\
&KILL(\texttt{B3}) &&= \emptyset\\
&KILL(\texttt{B4}) &&= \emptyset\\
&KILL(\texttt{B5}) &&= \emptyset\\
&KILL(\texttt{B6}) &&= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle\}\\
&KILL(\texttt{exit}) &&= \emptyset
\end{aligned}
$$

Next, we define data-flow equations for $CPin(i)$ and $CPout(i)$ that represent the sets of copy assignments that are available for copy propagation on entry to and exit from block i, respectively. A copy assignment is available on entry to block i if it is available on exit from all predecessors of block i, so the path-combining operator is intersection. A copy assignment is available on exit from block j if it is either in $COPY(j)$ or it is available on entry to block j and not killed by block j, i.e., if it is in $CPin(j)$ and not in $KILL(j)$. Thus, the data-flow equations are

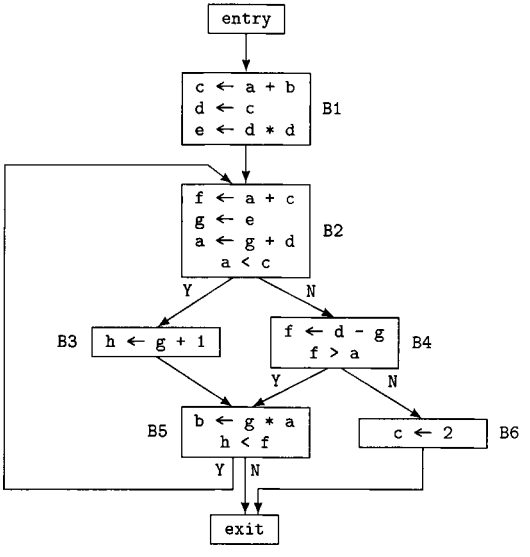

FIG. 12.26 Another example for copy propagation.

$$CPin(i) = \bigcap_{j \in Pred(i)} CPout(j)$$

$$CPout(i) = COPY(i) \cup (CPin(i) - KILL(i))$$

and the proper initialization is $CPin(\texttt{entry}) = \emptyset$ and $CPin(i) = U$ for all $i \neq \texttt{entry}$, where U is the universal set of quadruples, or at least

$$U = \bigcup_i COPY(i)$$

The data-flow analysis for global copy propagation can be performed efficiently with a bit-vector representation of the sets.

Given the data-flow information *CPin*( ) and assuming that we have already done local copy propagation, we perform global copy propagation as follows:

1. For each basic block B, set ACP = {a ∈ Var × Var where ∃w ∈ integer such that ⟨a@1,a@2,B,w⟩ ∈ CPin(B)}.
2. For each basic block B, perform the local copy-propagation algorithm from Figure 12.24 on block B (omitting the assignment ACP := ∅).

For our example in Figure 12.26, the *CPin*( ) sets are

$$CPin(\texttt{entry}) = \emptyset$$
$$CPin(\texttt{B1}) = \emptyset$$

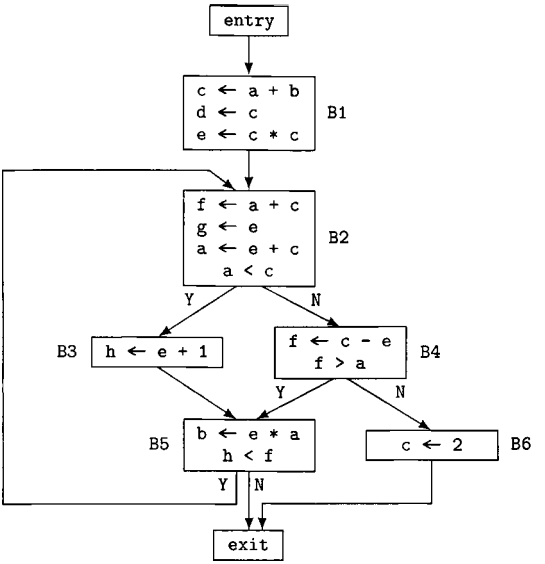

FIG. 12.27 Flowgraph from Figure 12.26 after copy propagation.

$$
\begin{aligned}
CPin(\texttt{B2}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle\} \\
CPin(\texttt{B3}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle, \langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\} \\
CPin(\texttt{B4}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle, \langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\} \\
CPin(\texttt{B5}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle, \langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\} \\
CPin(\texttt{exit}) &= \{\langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\}
\end{aligned}
$$

Doing local copy propagation within B1 and global copy propagation across the entire procedure turns the flowgraph in Figure 12.26 into the one in Figure 12.27.

The local copy-propagation algorithm can easily be generalized to work on extended basic blocks. To do so, we process the basic blocks that make up an extended basic block in preorder, i.e., each block before its successors, and we initialize the table ACP for each basic block other than the initial one with the final value of ACP from its predecessor block. Correspondingly, the global copy-propagation algorithm can be generalized to use extended basic blocks as the nodes with which data-flow information is associated. To do so, we must associate a separate *CPout*( ) set with each exit from an extended basic block, since the paths through the extended basic block will generally make different copy assignments available.

If we do local copy propagation followed by global copy propagation (both on extended basic blocks) for our example in Figure 12.26, the result is the same, but more of the work happens in the local phase. Blocks B2, B3, B4, and B6 make up an extended basic block and the local phase propagates the value of e assigned to g in block B2 to all of them.

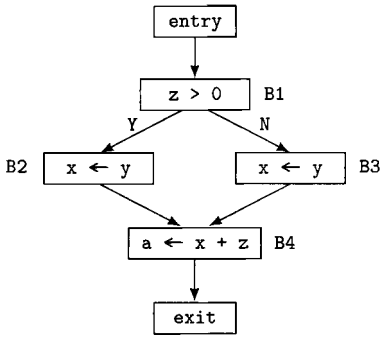

FIG. 12.28 Copy assignments not detected by global copy propagation.

Note that the global copy-propagation algorithm does not identify copy assignments such as the two x ← y statements in blocks B2 and B3 in Figure 12.28. The transformation known as tail merging (see Section 18.8) will replace the two copy assignments by one, in effect moving the copy into a separate basic block of its own. Copy propagation will then recognize it and propagate the copy into block B4. However, this presents a phase-ordering problem for some compilers: tail merging is generally not done until machine instructions have been generated.

Alternatively, either partial-redundancy elimination (Section 13.3) applied to assignments or code hoisting (Section 13.5) can be used to move both occurrences of the statement x ← y to block B1, and that can be done during the same optimization phase as copy propagation.

12.6 Sparse Conditional Constant Propagation

Constant propagation is a transformation that, given an assignment $x \leftarrow c$ for a variable x and a constant c, replaces later uses of x with uses of c as long as intervening assignments have not changed the value of x. For example, the assignment b ← 3 in block B1 in Figure 12.29(a) assigns the constant value 3 to b and no other assignment in the flowgraph assigns to b. Constant propagation turns the flowgraph into the one shown in Figure 12.29(b). Note that all occurrences of b have been replaced by 3 but neither of the resulting constant-valued expressions has been evaluated. This is done by constant-expression evaluation (see Section 12.1).

Constant propagation is particularly important for RISC architectures because it moves small integer constants to the places where they are used. Since all RISCs provide instructions that take a small integer constant as an operand (with the definition of "small" varying from one architecture to another), knowing that an operand is such a constant allows more efficient code to be generated. Also, some RISCs (e.g., MIPS) have an addressing mode that uses the sum of a register and a small constant but not one that uses the sum of two registers; propagating a small

3. Ullman's $T1$-$T2$ analysis;
4. Kennedy's node-listing algorithm;
5. Farrow, Kennedy, and Zucconi's graph-grammar approach;
6. elimination methods, e.g., interval analysis (see Section 8.8);
7. Rosen's high-level (syntax-directed) approach;
8. structural analysis (see Section 8.7); and
9. slotwise analysis (see Section 8.9).

Here we concentrate on three approaches: (1) the simple iterative approach, with several strategies for determining the order of the iterations; (2) an elimination or control-tree-based method using intervals; and (3) another control-tree-based method using structural analysis. As we shall see, these methods present a range of ease of implementation, speed and space requirements, and ease of incrementally updating the data-flow information to avoid totally recomputing it. We then make a few remarks about other approaches, such as the recently introduced slotwise analysis.

8.4 Iterative Data-Flow Analysis

Iterative analysis is the method we used in the example in Section 8.1 to perform reaching definitions analysis. We present it first because it is the easiest method to implement and, as a result, the one most frequently used. It is also of primary importance because the control-tree-based methods discussed in Section 8.6 need to be able to do iterative analysis (or node splitting or data-flow analysis over a lattice of functions) on improper (or irreducible) regions of code.

We first present an iterative implementation of forward analysis. Methods for backward and bidirectional problems are easy generalizations.

We assume that we are given a flowgraph $G = \langle N, E \rangle$ with `entry` and `exit` blocks in N and a lattice L and desire to compute $in(B), out(B) \in \mathrm{L}$ for each $B \in N$, where $in(B)$ represents the data-flow information on entry to B and $out(B)$ represents the data-flow information on exit from B, given by the data-flow equations

$$in(B) = \begin{cases} Init & \text{for } B = \texttt{entry} \\ \prod_{P \in Pred(B)} out(P) & \text{otherwise} \end{cases}$$

$$out(B) = F_B(in(B))$$

where $Init$ represents the appropriate initial value for the data-flow information on entry to the procedure, $F_B(\ )$ represents the transformation of the data-flow information corresponding to executing block B, and $\sqcap$ models the effect of combining the data-flow information on the edges entering a block. Of course, this can also be expressed with just $in(\ )$ functions as

```
   procedure Worklist_Iterate(N,entry,F,dfin,Init)
      N: in set of Node
      entry: in Node
      F: in Node × L ⟶ L
      dfin: out Node ⟶ L
      Init: in L
   begin
      B, P: Node
      Worklist: set of Node
      effect, totaleffect: L
      dfin(entry) := Init
*     Worklist := N - {entry}
      for each B ∈ N do
         dfin(B) := ⊤
      od
      repeat
*        B := ♦Worklist
         Worklist -= {B}
         totaleffect := ⊤
         for each P ∈ Pred(B) do
            effect := F(P,dfin(P))
            totaleffect ⊓= effect
         od
         if dfin(B) ≠ totaleffect then
            dfin(B) := totaleffect
*           Worklist ∪= Succ(B)
         fi
      until Worklist = ∅
   end    || Worklist_Iterate
```

FIG. 8.6 Worklist algorithm for iterative data-flow analysis (statements that manage the worklist are marked with asterisks).

$$in(B) = \begin{cases} Init & \text{for } B = \text{entry} \\ \bigsqcap_{P \in Pred(B)} F_P(in(P)) & \text{otherwise} \end{cases}$$

If $\sqcup$ models the effect of combining flow information, it is used in place of $\sqcap$ in the algorithm. The value of *Init* is usually $\top$ or $\bot$.

The algorithm `Worklist_Iterate( )`, given in Figure 8.6, uses just *in*( ) functions; the reader can easily modify it to use both *in*( )s and *out*( )s. The strategy is to iterate application of the defining equations given above, maintaining a worklist of blocks whose predecessors' *in*( ) values have changed on the last iteration, until the worklist is empty; initially the worklist contains all blocks in the flowgraph except `entry`, since its information will never change. Since the effect of combining information from edges entering a node is being modeled by $\sqcap$, the appropriate initialization for `totaleffect` is $\top$. The function $F_B(x)$ is represented by `F(B,x)`.

The computational efficiency of this algorithm depends on several things: the lattice L, the flow functions F_B( ), and how we manage the worklist. While the lattice

TABLE 8.2 Flow functions for the flowgraph in Figure 7.4.

$F_{\texttt{entry}} = id$
$F_{\texttt{B1}}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle) = \langle 111x_4x_500x_8\rangle$
$F_{\texttt{B2}} = id$
$F_{\texttt{B3}}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle) = \langle x_1x_2x_31x_5x_6x_70\rangle$
$F_{\texttt{B4}} = id$
$F_{\texttt{B5}} = id$
$F_{\texttt{B6}}(\langle x_1x_2x_3x_4x_5x_6x_7x_8\rangle) = \langle x_10001111\rangle$

and flow functions are determined by the data-flow problem we are solving, the management of the worklist is independent of it. Note that managing the worklist corresponds to how we implement the statements marked with asterisks in Figure 8.6. The easiest implementation would use a stack or queue for the worklist, without regard to how the blocks are related to each other by the flowgraph structure. On the other hand, if we process all predecessors of a block before processing it, then we can expect to have the maximal effect on the information for that block each time we encounter it. This can be achieved by beginning with an ordering we encountered in the preceding chapter, namely, reverse postorder, and continuing with a queue. Since in postorder a node is not visited until all its depth-first spanning-tree successors have been visited, in reverse postorder it is visited before any of its successors have been. If A is the maximal number of back edges on any acyclic path in a flowgraph G, then $A + 2$ passes through the `repeat` loop are sufficient if we use reverse postorder.[4] Note that it is possible to construct flowgraphs with A on the order of $|N|$, but that this is very rare in practice. In almost all cases $A \leq 3$, and frequently $A = 1$.

As an example of the iterative forward algorithm, we repeat the example we did informally in Section 8.1. The flow functions for the individual blocks are given in Table 8.2, where *id* represents the identity function. The initial value of *dfin*(*B*) for all blocks is ⟨00000000⟩. The path-combining operator is ⊔ or bitwise logical or on the bit vectors. The initial worklist is {`B1`, `B2`, `B3`, `B4`, `B5`, `B6`, `exit`}, in reverse postorder.

Entering the `repeat` loop, the initial value of `B` is `B1`, with the worklist becoming {`B2`, `B3`, `B4`, `B5`, `B6`, `exit`}. `B1`'s only predecessor is `P = entry`, and the result of computing `effect` and `totaleffect` is ⟨00000000⟩, unchanged from the initial value of `dfin(B1)`, so `B1`'s successors are not put into the worklist.

Next, we get `B = B2` and the worklist becomes {`B3`, `B4`, `B5`, `B6`, `exit`}. The only predecessor of `B2` is `P = B1`, and the result of computing `effect` and `totaleffect` is ⟨11100000⟩, which becomes the new value of `dfin(B2)`, and `exit` is added to the worklist to produce {`B3`, `B4`, `B5`, `B6`, `exit`}.

Next, we get `B = B3`, and the worklist becomes {`B4`, `B5`, `B6`, `exit`}. `B3` has one predecessor, namely, `B1`, and the result of computing `effect` and `totaleffect` is

4. If we keep track of the number of blocks whose data-flow information changes in each pass, instead of simply whether there have been any changes, this bound can be reduced to $A + 1$.

⟨11100000⟩, which becomes the new value of `dfin(B3)`, and `B4` is put onto the worklist.

Then we get `B` = `B4`, and the worklist becomes {`B5`, `B6`, `exit`}. `B4` has two predecessors, `B3` and `B6`, with `B3` contributing ⟨11110000⟩ to `effect`, `totaleffect`, and `dfin(B4)`, and `B6` contributing ⟨00001111⟩, so that the final result of this iteration is `dfin(B4)` = ⟨11111111⟩ and the worklist becomes {`B5`, `B6`, `exit`}.

Next, `B` = `B5`, and the worklist becomes {`B6`, `exit`}. `B5` has one predecessor, `B4`, which contributes ⟨11111111⟩ to `effect`, `totaleffect`, and `dfin(B5)`, and `exit` is added to the worklist.

Next, `B` = `B6`, and the worklist becomes {`exit`}. `B6`'s one predecessor, `B4`, contributes ⟨11111111⟩ to `dfin(B6)`, and `B4` is put back onto the worklist.

Now `exit` is removed from the worklist, resulting in {`B4`}, and its two predecessors, `B2` and `B5`, result in `dfin(exit)` = ⟨11111111⟩.

The reader can check that the body of the `repeat` loop is executed twice more for each element of the worklist, but that no further changes result in the `dfin( )` values computed in the last iteration. One can also check that the results are identical to those computed in Section 8.1.

Converting the algorithm above to handle backward problems is trivial, once we have properly posed a backward problem. We can either choose to associate the data-flow information for a backward problem with the entry to each block or with its exit. To take advantage of the duality between forward and backward problems, we choose to associate it with the exit.

As for a forward analysis problem, we assume that we are given a flowgraph $G = \langle N, E \rangle$ with `entry` and `exit` blocks in N and that we desire to compute $out(B) \in \mathbf{L}$ for each $B \in N$ where $out(B)$ represents the data-flow information on exit from B, given by the data-flow equations

$$out(B) = \begin{cases} Init & \text{for } B = \texttt{exit} \\ \bigsqcap_{P \in Succ(B)} in(P) & \text{otherwise} \end{cases}$$

$$in(B) = F_B(out(B))$$

where *Init* represents the appropriate initial value for the data-flow information on exit from the procedure, $F_B( )$ represents the transformation of the data-flow information corresponding to executing block B in reverse, and $\sqcap$ models the effect of combining the data-flow information on the edges exiting a block. As for forward-flow problems, they can also be expressed with just $out( )$ functions as

$$out(B) = \begin{cases} Init & \text{for } B = \texttt{exit} \\ \bigsqcap_{P \in Succ(B)} F_P(out(P)) & \text{otherwise} \end{cases}$$

If $\sqcup$ models the effect of combining flow information, it is used in place of $\sqcap$ in the algorithm.

Now the iterative algorithm for backward problems is identical to that given for forward problems in Figure 8.6, with the appropriate substitutions: *out*( ) for *in*( ), `exit` for `entry`, and *Succ*( ) for *Pred*( ). The most effective way to manage the worklist is by initializing it in reverse preorder, and the computational efficiency bound is the same as for the forward iterative algorithm.

8.5 Lattices of Flow Functions

Just as the objects on which we perform a data-flow analysis are best viewed as elements of a lattice, the set of flow functions we use in performing such an analysis also forms a lattice with its meet and join induced by those of the underlying lattice. As we shall see in Section 8.6, the induced lattice of monotone flow functions is very important to formulating the control-tree-based approaches to data-flow analysis.

In particular, let L be a given lattice and let L^F denote the set of all monotone functions from L to L, i.e.,

$$f \in L^F \text{ if and only if } \forall x, y \in L\ x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y)$$

Then the induced pointwise meet operation on L^F given by

$$\forall f, g \in L^F,\ \forall x \in L\ (f \sqcap g)(x) = f(x) \sqcap g(x)$$

and the corresponding induced join and order functions are all easily checked to be well defined, establishing that L^F is indeed a lattice. The bottom and top elements of L^F are $\bot^F$ and $\top^F$, defined by

$$\forall x \in L\ \bot^F(x) = \bot \text{ and } \top^F(x) = \top$$

To provide the operations necessary to do control-tree-based data-flow analysis, we need to define one more function in and two more operations on L^F. The additional function is simply the *identity* function *id*, defined by $id(x) = x$, $\forall x \in L$. The two operations are composition and Kleene (or iterative) closure. For any two functions $f, g \in L^F$, the *composition* of f and g, written $f \circ g$, is defined by

$$(f \circ g)(x) = f(g(x))$$

It is easily shown that L^F is closed under composition. Also, for any $f \in L^F$, we define f^n by

$$f^0 = id \text{ and for } n \geq 1,\ f^n = f \circ f^{n-1}$$

The *Kleene closure* of $f \in L^F$, written f^*, is defined by

$$\forall x \in L\ f^*(x) = \lim_{n \to \infty} (id \sqcap f)^n(x)$$

Also, as is usual, we define $f^+ = f \circ f^*$. To show that L^F is closed under Kleene closure, we rely on the fact that our lattices all have finite effective heights under all the functions we use. This implies that if we compute for any $x_0 \in L$ the sequence

$$x_{i+1} = (id \sqcap f)(x_i)$$

there is an i such that $x_i = x_{i+1}$ and so, clearly, $f^*(x_0) = x_i$.