with those value numbers. Actually, $\tau_6$ is useless and would be eliminated during local live variable analysis. Also $\tau_7$, being a temporary, would not be computed itself; rather, uses of $\tau_7$ would be replaced by uses of (18).

**Copy Propagation**

Algorithm 10.5 just presented, and various other algorithms such as induction-variable elimination discussed later in this section, introduce copy statements of the form $x := y$. Copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction discussed in Section 9.8. It is sometimes possible to eliminate copy statement $s: x := y$ if we determine all places where this definition of $x$ is used. We may then substitute $y$ for $x$ in all these places, provided the following conditions are met by every such use $u$ of $x$.

1. Statement $s$ must be the only definition of $x$ reaching $u$ (that is, the ud-chain for use $u$ consists only of $s$).

2. On every path from $s$ to $u$, including paths that go through $u$ several times (but do not go through $s$ a second time), there are no assignments to $y$.

Condition (1) can be checked using ud-chaining information, but what of condition (2)? We shall set up a new data-flow analysis problem in which $in[B]$ is the set of copies $s: x := y$ such that every path from the initial node to the beginning of $B$ contains the statement $s$, and subsequent to the last occurrence of $s$ there are no assignments to $y$. The set $out[B]$ can be defined correspondingly, but with respect to the end of $B$. We say copy statement $s: x := y$ is *generated* in block $B$ if $s$ occurs in $B$ and there is no subsequent assignment to $y$ within $B$. We say $s: x := y$ is *killed* in $B$ if $x$ or $y$ is assigned there and $s$ is not in $B$. The notion that assignments to $x$ "kill" $x := y$ is familiar from reaching definitions, but the idea that assignments to $y$ do so is special to this problem. Note the important consequence of the fact that different assignments $x := y$ kill each other; $in[B]$ can contain only one copy statement with $x$ on the left.

Let $U$ be the "universal" set of all copy statements in the program. It is important to note that different statements $x := y$ are different in $U$. Define $c_{gen}[B]$ to be the set of all copies generated in block $B$, and $c_{kill}[B]$ to be the set of copy in $U$ that are killed in $B$. Then the following equations relate the quantities defined:

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

$$in[B] = \bigcap_{P\text{ a predecessor of } B} out[P]\text{ for } B \text{ not initial}$$

$$in[B_1] = \emptyset \text{ where } B_1 \text{ is the initial block}$$

(10.12)
Equations 10.12 are identical to Equations 10.10, if \( e_{\text{kill}} \) is replaced by \( e_{\text{kill}} \) and \( e_{\text{gen}} \) by \( e_{\text{gen}} \). Thus, 10.12 can be solved by Algorithm 10.3, and we shall not discuss the matter further. We shall, however, give an example that exposes some of the nuances of copy optimization.

Example 10.19. Consider the flow graph of Fig. 10.35. Here, \( e_{\text{gen}}[B_1] = \{x := y\} \) and \( e_{\text{gen}}[B_3] = \{x := z\} \). Also, \( e_{\text{kill}}[B_2] = \{x := y\} \), since \( y \) is assigned in \( B_2 \). Finally, \( e_{\text{kill}}[B_1] = \{x := z\} \) since \( x \) is assigned in \( B_1 \), and \( e_{\text{kill}}[B_3] = \{x := y\} \) for the same reason.

![Flow graph](image)

**Fig. 10.35.** Example flow graph.

The other \( e_{\text{gen}} \)'s and \( e_{\text{kill}} \)'s are \( \emptyset \). Also, \( \text{in}[B_1] = \emptyset \) by Equations 10.12. Algorithm 10.3 in one pass determines that

\[
\text{in}[B_2] = \text{in}[B_3] = \text{out}[B_1] = \{x := y\}
\]

Likewise, \( \text{out}[B_2] = \emptyset \) and

\[
\text{out}[B_3] = \text{in}[B_4] = \text{out}[B_4] = \{x := z\}
\]

Finally, \( \text{in}[B_3] = \text{out}[B_2] \cap \text{out}[B_4] = \emptyset \).

We observe that neither copy \( x := y \) nor \( x := z \) "reaches" the use of \( x \) in \( B_4 \), in the sense of Algorithm 10.5. It is true but irrelevant that both these definitions of \( x \) "reach" \( B_3 \) in the sense of reaching definitions. Thus, neither copy may be propagated, as it is not possible to substitute \( y \) (respectively \( z \)) for \( x \) in all uses of \( x \) that definition \( x := y \) (respectively \( x := z \)) reaches. We could substitute \( x \) for \( z \) in \( B_4 \), but that would not improve the code.

We now specify the details of the algorithm to remove copy statements.

Algorithm 10.6. Copy propagation.

**Input.** A flow graph \( G \), with ud-chains giving the definitions reaching block \( B \), and with \( c_{\text{in}}[B] \) representing the solution to Equations 10.12, that is, the set of copies \( x := y \) that reach block \( B \) along every path, with no assignment to \( x \) or \( y \) following the last occurrence of \( x := y \) on the path. We also need du-
chains giving the uses of each definition.

Output. A revised flow graph.

Method. For each copy \( s : x := y \) do the following.

1. Determine those uses of \( x \) that are reached by this definition of \( x \), namely, \( s : x := y \).

2. Determine whether for every use of \( x \) found in (1), \( s \) is in \( c_{in}(B) \), where \( B \) is the block of this particular use, and moreover, no definitions of \( x \) or \( y \) occur prior to this use of \( x \) within \( B \). Recall that if \( s \) is in \( c_{in}(B) \), then \( s \) is the only definition of \( x \) that reaches \( B \).

3. If \( s \) meets the conditions of (2), then remove \( s \) and replace all uses of \( x \) found in (1) by \( y \).

Detection of Loop-Invariant Computations

We shall make use of ud-chains to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. As discussed in Section 10.4, a loop is a region consisting of a set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header. We also require that a loop have at least one way to get back to the header from any block in the loop.

If an assignment \( x := y + z \) is at a position in the loop where all possible definitions of \( y \) and \( z \) are outside the loop (including the special case where \( y \) and/or \( z \) is a constant), then \( y + z \) is loop-invariant because its value will be the same each time \( x := y + z \) is encountered, as long as control stays within the loop. All such assignments can be detected from the ud-chains, that is, a list of all definition points of \( y \) and \( z \) reaching assignment \( x := y + z \).

Having recognized that the value of \( x \) computed at \( x := y + z \) does not change within the loop, suppose there is another statement \( v := x + w \), where \( w \) could only have been defined outside the loop. Then \( x + w \) is also loop-invariant.

We can use the above ideas to make repeated passes over the loop, discovering more and more computations whose value is loop-invariant. If we have both ud- and du-chains, we do not even have to make repeated passes over the code. The du-chain for definition \( x := y + z \) will tell us where this value of \( x \) could be used, and we need only check among these uses of \( x \) within the loop, that use no other definition of \( x \). These loop-invariant assignments may be moved to the preheader, provided their operands besides \( x \) are also loop-invariant, as discussed in the next algorithm.

Algorithm 10.7. Detection of loop-invariant computations.

Input. A loop \( L \) consisting of a set of basic blocks, each block containing a sequence of three-address statements. We assume ud-chains, as computed in Section 10.5, are available for the individual statements.