17 Dataflow Analysis

anal-y-sis: an examination of a complex, its elements, and their relations

Webster's Dictionary

An optimizing compiler transforms programs to improve their efficiency without changing their output. There are many transformations that improve efficiency:

Register allocation: Keep two nonoverlapping temporaries in the same register.

Common-subexpression elimimination: If an expression is computed more than once, eliminate one of the computations.

Dead-code elimination: Delete a computation whose result will never be used.

Constant folding: If the operands of an expression are constants, do the computation at compile time.

This is not a complete list of optimizations. In fact, there can never be a complete list.

NO MAGIC BULLET

Computability theory shows that it will always be possible to invent new optimizing transformations.

Let us say that a *fully optimizing compiler* is one that transforms each program P to a program **Opt**(P) that is the *smallest* program with the same input/output behavior as P. We could also imagine optimizing for speed instead of program size, but let us choose size to simplify the discussion.

For any program Q that produces no output and never halts, **Opt**(Q) is short and easily recognizable:

L_1 : **goto** L_1

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

Therefore, if we had a fully optimizing compiler we could use it to solve the halting problem; to see if there exists an input on which P halts, just see if $\mathbf{Opt}(P)$ is the one-line infinite loop. But we know that no computable algorithm can always tell whether programs halt, so a fully optimizing compiler cannot be written either.

Since we can't make a *fully* optimizing compiler, we must build *optimizing compilers* instead. An optimizing compiler transforms P into a program P' that always has the same input/output behavior as P, and might be smaller or faster. We hope that P' runs faster than the optimized programs produced by our competitors' compilers.

No matter what optimizing compiler we consider, there must always exist another (usually bigger) optimizing compiler that does a better job. For example, suppose we have an optimizing compiler A. There must be some program P_x which does not halt, such that $A(P_x) \neq \mathbf{Opt}(P_x)$. If this were not the case, then A would be a fully optimizing compiler, which we could not possibly have. Therefore, there exists a better compiler B:

$$B(P) = \textbf{if } P = P_x \textbf{ then } [\texttt{L : goto L}] \textbf{ else } A(P)$$

Although we don't know what P_x is, it is certainly just a string of source code, and given that string we could trivially construct B.

The optimizing compiler B isn't very useful – it's not worth handling special cases like P_x one at a time. In real life, we improve A by finding some reasonably general program transformation (such as the ones listed at the beginning of the chapter) that improves the performance of many programs. We add this transformation to the optimizer's "bag of tricks" and we get a more competent compiler. When our compiler knows enough tricks, we deem it *mature*.

This theorem, that for any optimizing compiler there exists a better one, is known as the *full employment theorem for compiler writers*.

17.1 INTERMEDIATE REPRESENTATION FOR FLOW ANALYSIS

In this chapter we will consider *intraprocedural global optimization. Intraprocedural* means the analysis stays within a single procedure or function (of a language like Tiger); *global* means that the analysis spans all the statements or basic blocks within that procedure. *Interprocedural* optimization is more global, operating on several procedures and functions at once.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

Each of the optimizing transformations listed at the beginning of the chapter can be applied using the following generic recipe:

Dataflow analysis: Traverse the flow graph, gathering information about what may happen at run time (this will necessarily be a conservative approximation).

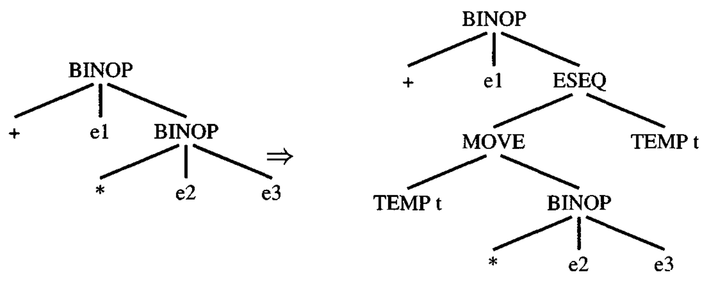
Transformation: Modify the program to make it faster in some way; the information gathered by analysis will guarantee that the program's result is unchanged.

There are many dataflow analyses that can provide useful information for optimizing transformations. Like the *liveness analysis* described in Chapter 10, most can be described by *dataflow equations*, a set of simultaneous equations derived from nodes in the flow graph.

QUADRUPLES

Chapter 10's liveness analysis operates on `Assem` instructions, which clearly indicate *uses* and *defs* but whose actual operations are machine-dependent assembly-language strings. Liveness analysis, and register allocation based on it, do not need to know what operations the instructions are performing, just their uses and definitions. But for the analyses and optimizations in this chapter, we need to understand the *operations* as well. Therefore, instead of `Assem` instructions we will use `Tree`-language terms (Section 7.2), simplified even further by ensuring that each `exp` has only a single MEM or BINOP node.

We can easily turn ordinary `Tree` expressions into simplified ones. Wherever there is a nested expression of one BINOP or MEM inside another, or a BINOP or MEM inside a JUMP or CJUMP, we introduce a new temporary using ESEQ:



and then apply the `Canon` module to remove all the ESEQ nodes.

We also introduce new temporaries to ensure that any *store* statement (that

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

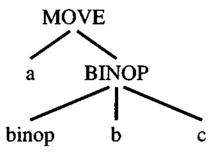
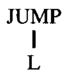
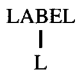
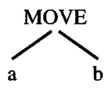
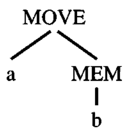
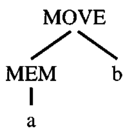
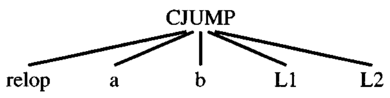
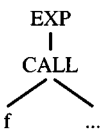
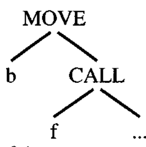
 $a \leftarrow b\ binop\ c$	 goto L	 $L:$
 $a \leftarrow b$	 $a \leftarrow M[b]$	 $M[a] \leftarrow b$
 if a *relop* b goto L_1 else goto L_2	 $f(a_1, \ldots, a_n)$	 $b \leftarrow f(a_1, \ldots, a_n)$

TABLE 17.1. Quadruples expressed in the `Tree` language. Occurrences of a, b, c, f, L denote TEMP, CONST, or LABEL nodes only.

is, a MOVE whose left-hand side is a MEM node) has only a TEMP or a CONST on its right-hand side, and only a TEMP or CONST under the MEM.

The statements that remain are all quite simple; they take one of the forms shown in Table 17.1.

Because the "typical" statement is $a \leftarrow b \oplus c$ with four components $(a, b, c, \oplus)$, these simple statements are often called *quadruples*. We use $\oplus$ to stand for an arbitrary *binop*.

A more efficient compiler would represent quadruples with their own data type (instead of using `Tree` data structures), and would translate from trees to quadruples all in one pass.

Intraprocedural optimizations take these quadruples that come out of the `Canon` phase of the compiler, and transform them into a new set of quadruples. The optimizer may move, insert, delete, and modify the quadruples. The resulting procedure-body must then be fed into the instruction-selection phase

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

of the compiler. However, the tree matching will not be very effective on the "atomized" trees where each expression contains only one BINOP or MOVE. After the optimizations are completed, there will be many MOVE statements that define temporaries that are used only once. It will be necessary to find these and turn them back into nested expressions.

We make a control flow graph of the quadruples, with a directed edge from each node (statement) n to its successors – that is, the nodes that can execute immediately after n.

17.2 VARIOUS DATAFLOW ANALYSES

A dataflow analysis of a control flow graph of quadruples collects information about the execution of the program. One dataflow analysis determines how definitions and uses are related to each other, another estimates what values a variable might have at a given point, and so on. The results of these analyses can be used to make optimizing transformations of the program.

REACHING DEFINITIONS

For many optimizations we need to see if a particular assignment to a temporary t can directly affect the value of t at another point in the program. We say that an *unambiguous definition* of t is a particular statement (quadruple) in the program of the form $t \leftarrow a \oplus b$ or $t \leftarrow M[a]$. Given such a definition d, we say that d *reaches* a statement u in the program if there is some path of control flow edges from d to u that does not contain any unambiguous definition of t.

An *ambiguous* definition is a statement that might or might not assign a value to t. For example, if t is a global variable, and the statement s is a CALL to a function that sometimes modifies t but sometimes does not, then s is an ambiguous definition. But our Tiger compiler treats escaping variables as memory locations, not as temporaries subject to dataflow analysis. This means that we never have ambiguous definitions; unfortunately, we also lose the opportunity to perform optimizations on escaping variables. For the remainder of this chapter, we will assume all definitions are unambiguous.

We can express the calculation of reaching definitions as the solution of dataflow equations. We label every MOVE statement with a definition-ID, and we manipulate sets of definition-IDs. We say that the statement $d_1 : t \leftarrow x \oplus y$ *generates* the definition d_1, because no matter what other definitions reach

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

Statement s	$gen[s]$	$kill[s]$
$d: t \leftarrow b \oplus c$	$\{d\}$	$defs(t) - \{d\}$
$d: t \leftarrow M[b]$	$\{d\}$	$defs(t) - \{d\}$
$M[a] \leftarrow b$	$\{\}$	$\{\}$
if a relop b goto L_1 else goto L_2	$\{\}$	$\{\}$
goto L	$\{\}$	$\{\}$
$L:$	$\{\}$	$\{\}$
$f(a_1, \ldots, a_n)$	$\{\}$	$\{\}$
$d: t \leftarrow f(a_1, \ldots, a_n)$	$\{d\}$	$defs(t) - \{d\}$

TABLE 17.2. *Gen* and *kill* for reaching definitions.

the beginning of this statement, we know that d_1 reaches the end of it. And we say that this statement *kills* any other definition of t, because no matter what other definitions of t reach the beginning of the statement, they do not reach the end (they cannot directly affect the value of t after this statement).

Let us define $defs(t)$ as the set of all definitions (or definition-IDs) of the temporary t. Table 17.2 summarizes the *generate* and *kill* effects of the different kinds of quadruples.

Using *gen* and *kill*, we can compute $in[n]$ (and $out[n]$) the set of definitions that reach the beginning (and end) of each node n:

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

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

These equations can be solved by iteration: first $in[n]$ and $out[n]$ are initialized to the empty set, for all n; then the equations are treated as assignment statements and repeatedly executed until there are no changes.

We will take Program 17.3 as an example; it is annotated with statement numbers that will also serve as definition-IDs. In each iteration, we recalculate *in* and *out* for each statement in turn:

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

```
1 :        $a \leftarrow 5$
2 :        $c \leftarrow 1$
3 : $L_1$ : if $c > a$ goto $L_2$
4 :        $c \leftarrow c + c$
5 :        goto $L_1$
6 : $L_2$ : $a \leftarrow c - a$
7 :        $c \leftarrow 0$
```

PROGRAM 17.3.

			Iter. 1		Iter. 2		Iter. 3	
n	*gen*[*n*]	*kill*[*n*]	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]	*in*[*n*]	*out*[*n*]
1	1	6		1		1		1
2	2	4,7	1	1,2	1	1,2	1	1,2
3			1,2	1,2	1,2,4	1,2,4	1,2,4	1,2,4
4	4	2,7	1,2	1,4	1,2,4	1,4	1,2,4	1,4
5			1,4	1,4	1,4	1,4	1,4	1,4
6	6	1	1,2	2,6	1,2,4	2,4,6	1,2,4	2,4,6
7	7	2,4	2,6	6,7	2,4,6	6,7	2,4,6	6,7

Iteration 3 serves merely to discover that nothing has changed since iteration 2.

Having computed reaching definitions, what can we do with the information? The analysis is useful in several kinds of optimization. As a simple example, we can do *constant propagation:* only one definition of a reaches statement 3, so we can replace the test $c > a$ with $c > 5$.

AVAILABLE EXPRESSIONS

Suppose we want to do *common-subexpression elimination*; that is, given a program that computes $x \oplus y$ more than once, can we eliminate one of the duplicate computations? To find places where such optimizations are possible, the notion of *available expressions* is helpful.

An expression $x \oplus y$ is *available* at a node n in the flow graph if, on every path from the entry node of the graph to node n, $x \oplus y$ is computed at least once *and* there are no definitions of x or y since the most recent occurrence of $x \oplus y$ on that path.

We can express this in dataflow equations using *gen* and *kill* sets, where the sets are now sets of expressions.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

Statement s	$gen[s]$	$kill[s]$
$t \leftarrow b \oplus c$	$\{b \oplus c\} - kill[s]$	*expressions containing* t
$t \leftarrow M[b]$	$\{M[b]\} - kill[s]$	*expressions containing* t
$M[a] \leftarrow b$	$\{\}$	*expressions of the form* $M[x]$
if $a > b$ goto L_1 else goto L_2	$\{\}$	$\{\}$
goto L	$\{\}$	$\{\}$
$L:$	$\{\}$	$\{\}$
$f(a_1, \ldots, a_n)$	$\{\}$	*expressions of the form* $M[x]$
$t \leftarrow f(a_1, \ldots, a_n)$	$\{\}$	*expressions containing* t, *and expressions of the form* $M[x]$

TABLE 17.4. *Gen* and *kill* for available expressions.

Any node that computes $x \oplus y$ *generates* $\{x \oplus y\}$, and any definition of x or y *kills* $\{x \oplus y\}$; see Table 17.4.

Basically, $t \leftarrow b + c$ generates the expression $b + c$. But $b \leftarrow b + c$ does not generate $b + c$, because after $b + c$ there is a subsequent definition of b. The statement $gen[s] = \{b \oplus c\} - kill[s]$ takes care of this subtlety.

A *store* instruction ($M[a] \leftarrow b$) might modify any memory location, so it kills any *fetch* expression ($M[x]$). If we were sure that $a \neq x$, we could be less conservative, and say that $M[a] \leftarrow b$ does not kill $M[x]$. This is called *alias analysis*; see Section 17.5.

Given *gen* and *kill*, we compute *in* and *out* almost as for reaching definitions, except that we compute the *intersection* of the *out* sets of the predecessors instead of a union. This reflects the fact that an expression is available only if it is computed on *every* path into the node.

$$\begin{aligned} in[n] &= \bigcap_{p \in pred[n]} out[p] && \text{if } n \text{ is not the start node} \\ out[n] &= gen[n] \cup (in[n] - kill[n]) \end{aligned}$$

To compute this by iteration, we define the *in* set of the start node as empty, and initialize all other sets to *full* (the set of all expressions), not empty. This is because the intersection operator makes sets *smaller*, not bigger as the union operator does in the computation of reaching definitions. This algorithm then finds the *greatest* fixed point of the equations.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

REACHING EXPRESSIONS

We say that an expression $t \leftarrow x \oplus y$ (in node s of the flow graph) reaches node n if there is a path from s to n that does not go through any assignment to x or y, or through any computation of $x \oplus y$. As usual, we can express *gen* and *kill*; see Exercise 17.1.

In practice, the *reaching expressions* analysis is needed by the *common-subexpression elimination* optimization only for a small subset of all the expressions in a program. Thus, reaching expressions are usually computed ad hoc, by searching backward from node n and stopping whenever a computation $x \oplus y$ is found. Or reaching expressions can be computed during the calculation of available expressions; see Exercise 17.4.

LIVENESS ANALYSIS

Chapter 10 has already covered liveness analysis, but it is useful to note that liveness can also be expressed in terms of *gen* and *kill*. Any use of a variable generates liveness, and any definition kills liveness:

Statement s	$gen[s]$	$kill[s]$
$t \leftarrow b \oplus c$	$\{b, c\}$	$\{t\}$
$t \leftarrow M[b]$	$\{b\}$	$\{t\}$
$M[a] \leftarrow b$	$\{b\}$	$\{\}$
if $a > b$ goto L_1 else goto L_2	$\{a, b\}$	$\{\}$
goto L	$\{\}$	$\{\}$
$L:$	$\{\}$	$\{\}$
$f(a_1, \ldots, a_n)$	$\{a_1, \ldots, a_n\}$	$\{\}$
$t \leftarrow f(a_1, \ldots, a_n)$	$\{a_1, \ldots, a_n\}$	$\{t\}$

The equations for *in* and *out* are similar to the ones for reaching definitions and available expressions, but *backward* because liveness is a *backward* dataflow analysis:

$$in[n] = gen[n] \cup (out[n] - kill[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

17.3 TRANSFORMATIONS USING DATAFLOW ANALYSIS

Using the results of dataflow analysis, the optimizing compiler can improve the program in several ways.

COMMON-SUBEXPRESSION ELIMINATION

Given a flow-graph statement $s : t \leftarrow x \oplus y$, where the expression $x \oplus y$ is *available* at s, the computation within s can be eliminated.

Algorithm. Compute *reaching expressions*, that is, find statements of the form $n : v \leftarrow x \oplus y$, such that the path from n to s does not compute $x \oplus y$ or define x or y.

Choose a new temporary w, and for such n, rewrite as

$$n : w \leftarrow x \oplus y$$
$$n' : v \leftarrow w$$

Finally, modify statement s to be

$$s : t \leftarrow w$$

We will rely on copy propagation to remove some or all of the extra assignment quadruples.

CONSTANT PROPAGATION

Suppose we have a statement $d : t \leftarrow c$ where c is a constant, and another statement n that uses t, such as $n : y \leftarrow t \oplus x$.

We know that t is constant in n if d reaches n, and no other definitions of t reach n.

In this case, we can rewrite n as $y \leftarrow c \oplus x$.

COPY PROPAGATION

This is like constant propagation, but instead of a constant c we have a variable z.

Suppose we have a statement $d : t \leftarrow z$. and another statement n that uses t, such as $n : y \leftarrow t \oplus x$.

If d reaches n, and no other definition of t reaches n, and there is no definition of z on any path from d to n (including a path that goes through n one or more times), then we can rewrite n as $n : y \leftarrow z \oplus x$.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

A good graph-coloring register allocator will do *coalescing* (see Chapter 11), which is a form of copy propagation. It detects any intervening definitions of z in constructing the interference graph – an assignment to z while d is live makes an interference edge (z, d), rendering d and z uncoalesceable.

If we do copy propagation before register allocation, then we may increase the number of spills. Thus, if our only reason to do copy propagation were to delete redundant MOVE instructions, we should wait until register allocation. However, copy propagation at the quadruple stage may enable the recognition of other optimizations such as common-subexpression elimination. For example, in the program

$$\begin{aligned} a &\leftarrow y + z \\ u &\leftarrow y \\ c &\leftarrow u + z \end{aligned}$$

the two $+$-expressions are not recognized as common subexpressions until after the copy propagation of $u \leftarrow y$ is performed.

DEAD-CODE ELIMINATION

If there is a quadruple $s : a \leftarrow b \oplus c$ or $s : a \leftarrow M[x]$, such that a is not *live-out* of s, then the quadruple can be deleted.

Some instructions have implicit side effects. For example, if the computer is configured to raise an exception on an arithmetic overflow or divide by zero, then deletion of an exception-causing instruction will change the result of the computation.

The optimizer should never make a change that changes program behavior, even if the change seems benign (such as the removal of a run-time "error"). The problem with such optimizations is that the programmer cannot predict the behavior of the program – and a program debugged with the optimizer enabled may fail with the optimizer disabled.

17.4 SPEEDING UP DATAFLOW ANALYSIS

Many dataflow analyses – including the ones described in this chapter – can be expressed using simultaneous equations on finite sets. So also can many of the algorithms used in constructing finite automata (Chapter 2) and parsers (Chapter 3). The equations can usually be set up so that they can be solved by *iteration*: by treating the equations as assignment statements and repeatedly

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

executing all the assignments until none of the sets changes any more.

There are several ways to speed up the evaluation of dataflow equations.

BIT VECTORS

A set S over a finite domain (that is, where the elements are integers in the range $1-N$ or can be put in an array indexed by $1-N$) can be represented by a *bit vector*. The ith bit in the vector is a 1 if the element i is in the set S.

In the bit-vector representation, unioning two sets S and T is done by a bitwise-*or* of the bit vectors. If the word size of the computer is W, and the vectors are N bits long, then a sequence of N/W *or* instructions can union two sets. Of course, $2N/W$ fetches and N/W stores will also be necessary, as well as indexing and loop overhead.

Intersection can be done by bitwise-*and*, set complement can be done by bitwise complement, and so on.

Thus, the bit-vector representation is commonly used for dataflow analysis. It would be inadvisable to use bit vectors for dataflow problems where the sets are expected to be very sparse (so the bit vectors would be almost all zeros), in which case a different implementation of sets would be faster.

BASIC BLOCKS

Suppose we have a node n in the flow graph that has only one predecessor, p, and p has only one successor, n. Then we can combine the *gen* and *kill* effects of p and n and replace nodes n and p with a single node. We will take *reaching definitions* as an example, but almost any dataflow analysis permits a similar kind of combining.

Consider what definitions reach *out* of the node n:

$$\mathit{out}[n] = \mathit{gen}[n] \cup (\mathit{in}[n] - \mathit{kill}[n])$$

We know $\mathit{in}[n]$ is just $\mathit{out}[p]$; therefore

$$\mathit{out}[n] = \mathit{gen}[n] \cup ((\mathit{gen}[p] \cup (\mathit{in}[p] - \mathit{kill}[p])) - \mathit{kill}[n])$$

By using the identity $(A \cup B) - C = (A - C) \cup (B - C)$ and then $(A - B) - C = A - (B \cup C)$, we have

$$\mathit{out}[n] = \mathit{gen}[n] \cup (\mathit{gen}[p] - \mathit{kill}[n]) \cup (\mathit{in}[p] - (\mathit{kill}[p] \cup \mathit{kill}[n]))$$

If we want to say that node pn combines the effects of p and n, then this last

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

equation says that the appropriate *gen* and *kill* sets for *pn* are:

$$
\begin{aligned}
gen[pn] &= gen[n] \cup (gen[p] - kill[n]) \\
kill[pn] &= kill[p] \cup kill[n]
\end{aligned}
$$

We can combine all the statements of a basic block in this way, and agglomerate the *gen* and *kill* effects of the whole block. The control-flow graph of basic blocks is much smaller than the graph of individual statements, so the multipass iterative dataflow analysis works much faster on basic blocks.

Once the iterative dataflow analysis algorithm is completed, we may recover the dataflow information of an individual statement (such as n) within a block (such pn in our example) by starting with the *in* set computed for the entire block and – in one pass – applying the *gen* and *kill* sets of the statements that precede n in the block.

ORDERING THE NODES

In a *forward* dataflow problem (such as reaching definitions or available expressions), the information coming *out* of a node goes *in* to the successors. If we could arrange that every node was calculated before its successors, the dataflow analysis would terminate in one pass through the nodes.

This would be possible if the control-flow graph had no cycles. We would *topologically sort* the flow graph – this just gives an ordering where each node comes before its successors – and then compute the dataflow equations in sorted order. But often the graph will have cycles, so this simple idea won't work. Even so, quasi-topologically sorting a cyclic graph by depth-first search helps to reduce the number of iterations required on cyclic graphs; in quasi-sorted order, most nodes come before their successors, so information flows forward quite far through the equations on each iteration.

Depth-first search (Algorithm 17.5) topologically sorts an acyclic graph graph, or quasi-topologically sorts a cyclic graph, quite efficiently. Using *sorted*, the order computed by depth-first search, the iterative solution of dataflow equations should be computed as

```
repeat
    for $i \leftarrow 1$ to $N$
        $n \leftarrow sorted[i]$
        $in \leftarrow \bigcup_{p \in pred[n]} out[p]$
        $out[n] \leftarrow gen[n] \cup (in - kill[n])$
until no $out$ set changed in this iteration
```

There is no need to make *in* a global array, since it is used only locally in

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

```
Topological-sort:
N ← number of nodes
for all nodes i
    mark[i] ← false
DFS(start-node)

function DFS(i)
  if mark[i] = false
    mark[i] ← true
    for each successor s of node i
        DFS(s)
    sorted[N] ← i
    N ← N − 1
```

ALGORITHM 17.5. Topological sort by depth-first search.

computing *out*.

For *backward* dataflow problems such as liveness analysis, we use a version of Algorithm 17.5 starting from *exit-node* instead of *start-node*, and traversing *predecessor* instead of *successor* edges.

USE-DEF AND DEF-USE CHAINS

Information about reaching definitions can be kept as *use-def chains*, that is, for each use of a variable x, a list of the definitions of x reaching that use. Use-def chains do not allow faster dataflow analysis per se, but allow efficient implementation of the optimization algorithms that use the results of the analysis.

A generalization of use-def chains is *static single-assignment form*, described in Chapter 19. SSA form not only provides more information than use-def chains, but the dataflow analysis that computes it is very efficient.

One way to represent the results of liveness analysis is via *def-use chains*: a list, for each definition, of all possible uses of that definition. SSA form also contains def-use information.

WORK-LIST ALGORITHMS

If any *out* set changes during an iteration of the **repeat-until** loop of an iterative solver, then all the equations are recalculated. This seems a pity, since most of the equations may not be affected by the change.

A *work-list* algorithm keeps track of just which *out* sets must be recalculated. Whenever node n is recalculated *and its out set is found to change*, all the successors of n are put onto the work-list (if they're not on it already). This is illustrated in Algorithm 17.6.

The algorithm will converge faster if, whenever a node is removed from

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

```
$W \leftarrow$ the set of all nodes
while $W$ is not empty
  remove a node $n$ from $W$
  $old \leftarrow out[n]$
  $in \leftarrow \bigcup_{p \in pred[n]} out[p]$
  $out[n] \leftarrow gen[n] \cup (in - kill[n])$
  if $old \neq out[n]$
    for each successor $s$ of $n$
      if $s \notin W$
        put $s$ into $W$
```

ALGORITHM 17.6. A work-list algorithm for reaching definitions.

W for processing, we choose the node in W that occurs earliest in the *sorted* array produced by Algorithm 17.5.

The coalescing, graph-coloring register allocator described in Chapter 11 is an example of a work-list algorithm with many different work-lists. Section 19.3 describes a work-list algorithm for constant propagation.

INCREMENTAL DATAFLOW ANALYSIS

Using the results of dataflow analysis, the optimizer can perform program transformations: moving, modifying, or deleting instructions. But optimizations can cascade:

- Removal of the dead code $a \leftarrow b \oplus c$ might cause b to become dead in a previous instruction $b \leftarrow x \oplus y$.
- One common-subexpression elimination begets another. In the program

  $$\begin{aligned} x &\leftarrow b + c \\ y &\leftarrow a + x \\ u &\leftarrow b + c \\ v &\leftarrow a + u \end{aligned}$$

  after $u \leftarrow b + c$ is replaced by $u \leftarrow x$, copy propagation changes $a + u$ to $a + x$, which is a common subexpression and can be eliminated.

A simple way to organize a dataflow-based optimizer is to perform a global flow analysis, then make all possible dataflow-based optimizations, then repeat the global flow analysis, then perform optimizations, and so on until no

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

more optimizations can be found. At best this iterates two or three times, so that on the third round there are no more transformations to perform.

But the worst case is very bad indeed. Consider a program in which the statement $z \leftarrow a_1 + a_2 + a_3 + \cdots + a_n$ occurs where z is dead. This translates into the quadruples

$$\begin{array}{ll} x_1 & \leftarrow a_1 + a_2 \\ x_2 & \leftarrow x_1 + a_3 \\ & \vdots \\ x_{n-2} & \leftarrow x_{n-3} + a_{n-1} \\ z & \leftarrow x_{n-2} + a_n \end{array}$$

Liveness analysis determines that z is dead; then dead-code elimination removes the definition of z. Then another round of liveness analysis determines that x_{n-2} is dead, and then dead-code elimination removes x_{n-2}, and so on. It takes n rounds of analysis and optimization to remove x_1 and then determine that there is no more work to do.

A similar situation occurs with common-subexpression elimination, when there are two occurrences of an expression such as $a_1 + a_2 + a_3 + \cdots + a_n$ in the program.

To avoid the need for repeated, global calculations of dataflow information, there are several strategies:

Cutoff: Perform no more than k rounds of analysis and optimization, for $k = 3$ or so. Later rounds of optimization may not be finding many transformations to do anyway. This is a rather unsophisticated approach, but at least the compilation will terminate in a reasonable time.

Cascading analysis: Design new dataflow analyses that can predict the cascade effects of the optimizations that will be done.

Incremental dataflow analysis: When the optimizer makes a program transformation – which renders the dataflow information invalid – instead of discarding the dataflow information, the optimizer should "patch" it.

Value numbering. The *value numbering* analysis is an example of a cascading analysis that, in one pass, finds all the (cascaded) common subexpressions within a basic block.

The algorithm maintains a table T, mapping *variables* to *value numbers*, and also mapping triples of the form (*value number, operator, value number*) to value numbers. For efficiency, T should be represented as a hash table. There is also a global number N counting how many distinct values have been seen so far.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

```
$T \leftarrow empty$
$N \leftarrow 0$
for each quadruple $a \leftarrow b \oplus c$ in the block
    if $(b \mapsto k) \in T$ for some $k$
        $n_b \leftarrow k$
    else
        $N \leftarrow N + 1$
        $n_b \leftarrow N$
        put $b \mapsto n_b$ into $T$
    if $(c \mapsto k) \in T$ for some $k$
        $n_c \leftarrow k$
    else
        $N \leftarrow N + 1$
        $n_c \leftarrow N$
        put $c \mapsto n_c$ into $T$
    if $((n_b, \oplus, n_c) \mapsto m) \in T$ for some $m$
        put $a \mapsto m$ into $T$
        mark this quadruple $a \leftarrow b \oplus c$ as a common subexpression
    else
        $N \leftarrow N + 1$
        put $(n_b, \oplus, n_c) \mapsto N$ into $T$
        put $a \mapsto N$ into $T$
```

ALGORITHM 17.7. Value numbering.

Using T and N, the value-numbering algorithm (Algorithm 17.7) scans the quadruples of a block from beginning to end. Whenever it sees an expression $b + c$, it looks up the value number of b and the value number of c. It then looks up hash$(n_b, n_c, +)$ in T; if found, it means that $b + c$ repeats the work of an earlier computation; we mark $b + c$ for deletion, and use the previously computed result. If not found, we leave $b + c$ in the program and also enter it in the hash table.

Figure 17.8 illustrates value numbering on a basic block: (a) is the list of quadruples, and (b) is the table (after the algorithm is finished). We can view the table as a directed acyclic graph (DAG), if we view an entry $(m, \oplus, n) \mapsto q$ as a node q with edges to nodes m and n, as shown in Figure 17.8c.

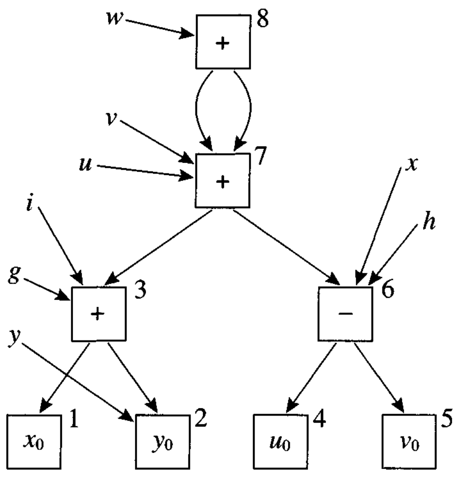
Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

$$\begin{aligned}
g &\leftarrow x + y\\
h &\leftarrow u - v\\
i &\leftarrow x + y\\
x &\leftarrow u - v\\
u &\leftarrow g + h\\
v &\leftarrow i + x\\
w &\leftarrow u + v
\end{aligned}$$

(a)

~~x~~	~~$\mapsto 1$~~
y	$\mapsto 2$
$(1, +, 2)$	$\mapsto 3$
g	$\mapsto 3$
~~u~~	~~$\mapsto 4$~~
~~v~~	~~$\mapsto 5$~~
$(4, -, 5)$	$\mapsto 6$
h	$\mapsto 6$
i	$\mapsto 3$
x	$\mapsto 6$
$(3, +, 6)$	$\mapsto 7$
u	$\mapsto 7$
v	$\mapsto 7$
$(7, +, 7)$	$\mapsto 8$
w	$\mapsto 8$

(b)



(c)

FIGURE 17.8. An illustration of value numbering. (a) A basic block; (b) the table created by the value-numbering algorithm, with hidden bindings shown crossed out; (c) a view of the table as a DAG.

Value numbering is an example of a single dataflow analysis that calculates the effect of cascaded optimizations: in this case, cascaded common-subexpression elimination. But the optimizer would like to perform a wide variety of transformations – especially when the loop optimizations described in the next chapter are included. It is very hard to design a single dataflow analysis capable of predicting the results of many different optimizations in combination.

Instead, we use a general-purpose dataflow analyzer and a general-purpose optimizer; but when the optimizer changes the program, it must tell the analyzer what information is no longer valid.

Incremental liveness analysis. For example, an incremental algorithm for liveness analysis must keep enough information so that if a statement is inserted or deleted, the liveness information can be efficiently updated.

Suppose we delete this statement $s\ :\ a \leftarrow b \oplus c$ from a flow graph on which we have *live-in* and *live-out* information for every node. The changes to the dataflow information are as follows:

1. a is no longer defined here. Therefore, if a is *live-out* of this node, it will now be *live-in* where it was not before.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

2. b is no longer used here. Therefore, if b is not *live-out* of this node, it will no longer be *live-in*. We must propagate this change backwards, and do the same for c.

A work-list algorithm will be useful here, since we can just add the predecessor of s to the work-list and run until the work-list is empty; this will often terminate quickly.

Propagating change (1) does the same kind of thing that the original (non-incremental) work-list algorithm for liveness does: it makes the live-sets bigger. Thus, our proof (Exercise 10.2) that the algorithm finds a least fixed point of the liveness equations also applies to the propagation of additional liveness caused by the deletion of the definition of a. Even the proof that the liveness analysis terminates was based on the idea that any change makes things bigger, and there was an a priori limit to how big the sets could get.

But change (2) makes live-sets smaller, not bigger, so naively running our original algorithm starting from the previously computed *in* and *out* sets may find a fixed point that is not a least fixed point. For example, suppose we have the following program:

```
0           $d \leftarrow 4$
1           $a \leftarrow 0$
2     $L_1 :$ $b \leftarrow a + 1$
3           $c \leftarrow c + b$
3a          $a \leftarrow d$
4           $a \leftarrow b \cdot 2$
5           if $a < N$ goto $L_1$
6           return $c$
```

Liveness analysis shows that d is *live-in* at statements 1, 2, 3, 3a, 4, 5. But a is not *live-out* of statement 3a, so this statement is dead code, and we can delete it. If we then start with the previously computed dataflow information and use Algorithm 10.4 (page 214) until it reaches a fixed point, we will end up with the column Y of Table 10.7, which is not the best possible approximation of the actual liveness information.

A more refined liveness analysis. Therefore, we must use a better algorithm. The solution is that at each point where a variable d is defined, we must keep track of exactly what uses it might have. Our liveness calculation will be very much like Algorithm 10.4, but it will operate on sets of *uses* instead of sets of *variables*. In fact, it is just like the reaching definitions algorithm in

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

reverse. Let $uses(v)$ be the set of all uses of variable v in the program. Given a statement $s :\ a \leftarrow b \oplus c$, the set

$$live\text{-}out[s] \cap uses(a)$$

contains all the uses of a that could possibly be reached by this definition.

Now, when we delete a quadruple that uses some variable b, we can delete that use of b from all the *live-in* and *live-out* sets. This gives the least fixed point, as we desire.

Cascades of dead code After deleting statement 3a from the program above, the incremental liveness analysis will find that statement 0 is dead code and can be deleted. Thus, incremental liveness analysis cooperates well with dead-code elimination. Other kinds of dataflow analysis can similarly be made incremental; sometimes, as in the case of liveness analysis, we must first refine the analysis.

17.5 ALIAS ANALYSIS

The analyses we have described in this chapter consider only the values of `Tree`-language temporaries. Variables that *escape* are represented (by the front end of the compiler) in memory locations with explicit fetches and stores, and we have not tried to analyze the definitions, uses, and liveness of these variables. The problem is that a variable or memory location may have several different names, or *aliases*, so that it is hard to tell which statements affect which variables.

Variables that can be aliases include:

- variables passed as call-by-reference parameters (in Pascal, C++, Fortran);
- variables whose address is taken (in C, C++);
- *l*-value expressions that dereference pointers, such as `p.x` in Tiger or `*p` in C;
- *l*-value expressions that explicitly subscript arrays, such as `a[i]`;
- and variables used in inner-nested procedures (in Pascal, Tiger, ML).

A good optimizer should optimize these variables. For example, in the program fragment

```
p.x := 5; q.x := 7; a := p.x
```

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

we might want our *reaching definitions* analysis to show that only one definition of `p.x` (namely, 5) reaches the definition of `a`. But the problem is that we cannot tell if one name is an alias for another. Could `q` point to the same record as `p`? If so, there are two definitions (5 and 7) that could reach `a`.

Similarly, with call-by-reference parameters, in the program

```
function f( ref i: int, ref j: int) =
   (i := 5; j := 7; return i)
```

a naive computation of reaching definitions would miss the fact that `i` might be the same variable as `j`, if `f` is called with `f(x,x)`.

The may-alias relation We use *alias analysis*, a kind of dataflow analysis, to learn about different names that may point to the same memory locations. The result of alias analysis is a *may-alias* relation: p may-alias q if, in some run of the program, p and q might point to the same data. As with most dataflow analyses, static (compile-time) information cannot be completely accurate, so the may-alias relation is conservative: we say that p may-alias q if we cannot prove that p is never an alias for q.

ALIAS ANALYSIS BASED ON TYPES

For languages with *strong typing* (such as Pascal, Java, ML, Tiger) where if two variables have incompatible types they cannot possibly be names for the same memory location, we can use the type information to provide a useful may-alias relation. Also in these languages the programmer cannot explicitly make a pointer point to a local variable, and we will use that fact as well.

We divide all the memory locations used by the program into disjoint sets, called *alias classes*. For Tiger, here are the classes we will use:

- For every frame location created by `Frame.allocLocal(true)`, we have a new class;
- For every record field of every record type, a new class;
- For every array type a, a new class.

The semantic analysis phase of the compiler must compute these classes, as they involve the concept of *type*, of which the later phases are ignorant. Each class can be represented by a different integer.

The `Translate` functions must label every fetch and store (that is, every MEM node in the `Tree` language) with its class. We will need to modify the `Tree` data structure, putting an `aliasClass` field into the MEM node.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

```
type list = {head: int,
             tail: list}
var p : list := nil
var q : list := nil
q := list{head=0, tail=nil};
p := list{head=0, tail=q};
q.head := 5;
a := p.head
```

(a) Tiger program

```
{int *p, *q;
 int h,i;
 p = &h;
 q = &i;
 *p = 0;
 *q = 5;
 a = *p;
}
```

(b) C program

PROGRAM 17.9. p and q are not aliases.

Given two MEM nodes $M_i[x]$ and $M_j[y]$, where i and j are the alias classes of the MEM nodes, we can say that $M_i[x]$ may-alias $M_j[y]$ if $i = j$.

This works for Tiger and Java. But it fails in the presence of call-by-reference or type casting.

ALIAS ANALYSIS BASED ON FLOW

Instead of, or in addition to, alias classes based on types, we can also make alias classes based on *point of creation*.

In Program 17.9a, even though p and q are the same type, we know they point to different records. Therefore we know that a must be assigned 0; the definition q.head:=5 cannot affect a. Similarly, in Program 17.9b we know p and q cannot be aliases, so a must be 0.

To catch these distinctions automatically, we will make an alias class for each point of creation. That is, for every different statement where a record is allocated (that is, for each call to malloc in C or new in Pascal or Java) we make a new alias class. Also, each different local or global variable whose address is taken is an alias class.

A pointer (or call-by-reference parameter) can point to variables of more than one alias class. In the program

```
1  p := list {head=0, tail=nil};
2  q := list {head=6, tail=p};
3  if a=0
4       then p:=q;
5  p.head := 4;
```

at line 5, q can point only to alias class 2, but p might point to alias class 1 or 2, depending on the value of a.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

Statement s		$trans_s(A)$
$t \leftarrow b$		$(A - \Sigma_t) \cup \{(t,d,k) \mid (b,d,k) \in A\}$
$t \leftarrow b + k$	(k is a constant)	$(A - \Sigma_t) \cup \{(t,d,i) \mid (b,d,i-k) \in A\}$
$t \leftarrow b \oplus c$		$(A - \Sigma_t) \cup \{(t,d,i) \mid (b,d,j) \in A \vee (c,d,k) \in A\}$
$t \leftarrow M[b]$		$A \cup \Sigma_t$
$M[a] \leftarrow b$		A
if $a > b$ goto L_1 else L_2		A
goto L		A
$L:$		A
$f(a_1, \ldots, a_n)$		A
$d: t \leftarrow$ `allocRecord`(a)		$(A - \Sigma_t) \cup \{(t,d,0)\}$
$t \leftarrow f(a_1, \ldots, a_n)$		$A \cup \Sigma_t$

TABLE 17.10. Transfer function for alias flow analysis.

So we must associate with each MEM node a set of alias classes, not just a single class. After line 2 we have the information $p \mapsto \{1\}, q \mapsto \{2\}$; out of line 4 we have $p \mapsto \{2\}, q \mapsto \{2\}$. But when two branches of control flow merge (in the example, we have the control edges $3 \to 5$ and $4 \to 5$) we must merge the alias class information; at line 5 we have $p \mapsto \{1, 2\}, q \mapsto \{2\}$.

Algorithm. The dataflow algorithm manipulates sets of tuples of the form (t, d, k) where t is a variable and d, k is the alias class of all instances of the kth field of a record allocated at location d. The set $in[s]$ contains (t, d, k) if $t - k$ might point to a record of alias class d at the beginning of statement s. This is an example of a dataflow problem where bit vectors will not work as well as a tree or hash table representation better suited to sparse problems.

Instead of using *gen* and *kill* sets, we use a transfer function: we say that if A is the alias information (set of tuples) on entry to a statement s, then $trans_s(A)$ is the alias information on exit. The transfer function is defined by Table 17.10 for the different kinds of quadruples.

The initial set A_0 includes the binding (FP, *frame*,0) where *frame* is the special alias class of all frame-allocated variables of the current function.

We use the abbreviation Σ_t to mean the set of all tuples (t, d, k), where d, k is the alias class of any record field whose type is compatible with variable t. Cooperation from the front end in providing a "small" Σ_t for each t makes the analysis more accurate. Of course, in a typeless language, or one with type-casts, Σ_t might have to be the set of all alias classes.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

The set equations for alias flow analysis are:

$$\begin{aligned} in[s_0] &= A_0 \qquad \text{where } s_0 \text{ is the start node} \\ in[n] &= \textstyle\bigcup_{p \in pred[n]} out[p] \\ out[n] &= trans_n(in[n]) \end{aligned}$$

and we can compute a solution by iteration in the usual way.

Producing may-alias information. Finally, we say that

p may-alias q at statement s

if there exists d, k such that $(p, d, k) \in in[s]$ and $(q, d, k) \in in[s]$.

USING MAY-ALIAS INFORMATION

Given the may-alias relation, we can treat each alias class as a "variable" in dataflow analyses such as reaching definitions and available expressions.

To take available expressions as an example, we modify one line of Table 17.4, the *gen* and *kill* sets:

Statement s	$gen[s]$	$kill[s]$
$M[a] \leftarrow b$	$\{\}$	$\{M[x] \mid a$ may alias x at $s\}$

Now we can analyze the following program fragment:

$$\begin{array}{lll} 1: & u & \leftarrow M[t] \\ 2: & M[x] & \leftarrow r \\ 3: & w & \leftarrow M[t] \\ 4: & b & \leftarrow u + w \end{array}$$

Without alias analysis, the store instruction in line 2 would *kill* the availability of $M[t]$, since we would not know whether t and x were related. But suppose alias analysis has determined that t may alias x at 2 is *false*; then $M[t]$ is still available at line 3, and we can eliminate the common subexpression; after copy propagation, we obtain:

$$\begin{array}{lll} 1: & z & \leftarrow M[t] \\ 2: & M[x] & \leftarrow r \\ 4: & b & \leftarrow z + z \end{array}$$

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

What I have shown here is intraprocedural alias analysis. But an interprocedural analysis would help to analyze the effect of CALL instructions. For example, in the program

```
1 :   $t \leftarrow fp + 12$
2 :   $u \leftarrow M[t]$
3 :   $f(t)$
4 :   $w \leftarrow M[t]$
5 :   $b \leftarrow u + w$
```

does the function f modify $M[t]$? If so, then $M[t]$ is not available at line 4.

However, interprocedural alias analysis is beyond the scope of this book.

ALIAS ANALYSIS IN STRICT PURE-FUNCTIONAL LANGUAGES

Some languages have *immutable* variables that cannot change after their initialization. For example, **const** variables in the C language, most variables in the ML language, and all variables in PureFun-Tiger (see Chapter 15) are immutable.

Alias analysis is not needed for these variables. The purpose of alias analysis is to determine whether different statements in the program interfere, or whether one definition *kills* another. Though it is true that there could be many pointers to the same value, none of the pointers can cause the value to change, i.e. no immutable variable can be killed.

This is a good thing for the optimizer, and also for the the programmer. The optimizer can do constant propagation and loop-invariant detection (see Chapter 18) without being bothered by aliases; and the programmer can understand what a segment of the program is doing also without the confusion and complexity introduced by stores through aliased pointers.

FURTHER READING

Gödel [1931] proved the *full employment theorem for mathematicians*. Turing [1937] proved that the halting problem is undecidable, and Rice [1953] proved the *full employment theorem for compiler writers*, even before there were any compiler writers.

Ershov [1958] developed value numbering. Allen [1969] codified many program optimizations; Allen [1970] and Cocke [1970] designed the first global dataflow analysis algorithms. Kildall [1973] first presented the fixed-point iteration method for dataflow analysis.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

Landi and Ryder [1992] give an algorithm for interprocedural alias analysis.

EXERCISES

17.1 Show the dataflow equations for *reaching expressions* (page 385). Be specific about what happens in the case of quadruples such as $t \leftarrow t \oplus b$ or $t \leftarrow M[t]$ where the defined temporary also appears on the right-hand side. The elements of the *gen* and *kill* sets will be definition-IDs, as in *reaching definitions*. **Hint:** If the definition on page 385 is not clear enough to formulate a precise definition, be guided by the role that reaching expressions must play in common-subexpression elimination (page 386).

17.2 Write down the control-flow graph of basic blocks (not just statements) for Program 17.3, and show the *gen* and *kill* sets (for reaching definitions) of each block.

***17.3** Show how to combine the *gen* and *kill* effects of two adjacent statements in the same basic block for each of:

a. Available expressions

b. Liveness analysis

****17.4** Modify the algorithm for computing *available expressions* to simultaneously compute *reaching expressions*. To make the algorithm more efficient, you may take advantage of the fact that if an expression is not available at statement s, then we do not need to know if it reaches s or not (for purposes of common-subexpression elimination). **Hint:** For each available expression $a + b$ that is propagated through statement s, also propagate a set representing all the statements that define $a + b$ and reach s.

17.5 Consider the calculation of *reaching definitions* on the following program:

```
x := 1;
y := 1;
if z <> 0
  then x := 2
  else y := 2;
w := x+y
```

a. Draw a control-flow graph for this program.

b. Show the *sorted* array that results from running Algorithm 17.5 on the program.

c. Calculate reaching definitions, showing the result of each iteration in tabular format as on page 383. How many iterations are required?

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018

*d. Prove that when *reaching definitions* is computed by iteration on an acyclic graph, taking the nodes in the order given by Algorithm 17.5, only one iteration is necessary (the second iteration merely verifies that nothing has changed).

Hint: Prove, and make use of, the lemma that each node is visited after all of its predecessors.

e. Suppose we order the nodes according to the order they are *first visited* by depth-first search. Calculate reaching definitions using that order, showing the results in tabular format; how many iterations are required?

***17.6** Write down a work-list algorithm for liveness analysis, in a form similar to that of Algorithm 17.6.

Downloaded from https://www.cambridge.org/core. NYU School of Medicine, on 10 Jan 2019 at 11:18:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811449.018