16
Program Optimization

This book has so far discussed the analysis and synthesis required to translate a programming language into interpretable or executable code. The analysis has been concerned with policing the programmer: making sure that the source program conforms to the definition of the programming language in which the program is written. After the compiler has verified that the source program conforms, synthesis takes over to translate the program. The target of this translation is typically an interpreatable or executable instruction set. Thus, code generation consists of translating a portion of a program into a sequence of instructions that mean the same thing.

As is true of most languages, there are many ways to say the same thing. In Chapter Chapter:global:fifteen, *instruction selection* is presented as a mechanism for choosing an efficient sequence of instructions for the target machine. In this chapter, we examine more aggressive techniques for improving a program's performance. Section 16.1 introduces program optimization—its role in a compiler, its organization, and its potential for improving program performance. Section 16.2 presents **data flow analysis**—a technique for determining useful properties of a program at compile-time. Section 16.3 considers some advanced analysis and optimizations.

16.1 Introduction

When compilers were first pioneered, they were considered successful if programs written in a high-level language attained performance that rivaled hand-coded efforts. By today's standards, the programming languages of those days may seem primitive. However, the technology that achieved the requisite performance is very impressive—such techniques are successfully applied to modern programming languages. The scale of today's software projects would be impossible without the advent of advanced programming paradigms and languages. As a result, the goal of hand-coded performance has yielded to the goal of obtaining a *reasonable* fraction of a target machine's potential speed.

Meanwhile, the trend in **reduced instruction set computer** (RISC) architecture points toward comparatively low-level instruction sets. Such architectures feature shorter instruction times with correspondingly faster clock rates. Other developments include *liquid* architectures, whose operations, registers, and data paths can be reconfigured. Special-purpose instructions have also been introduced, such as the MMX instructions for Intel machines. These instructions facilitate graphics and numerical computations. Architectural innovations cannot succeed unless compilers can make effective use of both RISC and the more specialized instructions. Thus, collaboration between computer architects, language designers, and compiler writers continues to be strong.

The programming language community has defined the **semantic gap** as a (subjective) measure of the distance between a compiler's source and target languages. As this gap continues to widen, the compiler community is challenged to build efficient bridges. These challenges come from many sources—examples include object-orientation, mobile code, active network components, and distributed object systems. Compilers must produce excellent code quickly, quietly, and—of course—correctly.

16.1.1 Why Optimize?

Although its given name is a misnomer, it is certainly the goal of program optimization to improve a program's performance. Truly *optimal* performance cannot be achieved automatically, as the task subsumes problems that are known to be *undecidable* [?]. The four main areas in which program optimizers strive for improvement are as follows:

- High-level language features.
- Targeting of machine-specific features.
- The compiler itself.
- A better algorithm.

```
$A \leftarrow B \times C$
function $\times(Y, Z)$ : $Matrix$
    if $Y.cols \neq Z.rows$                                        1
    then /* Throw an exception */
    else
        for $i = 1$ to $Y.rows$ do
            for $j = 1$ to $Z.cols$ do
                $Result[i, j] \leftarrow 0$
                for $k = 1$ to $Y.cols$ do
                    $Result[i, j] \leftarrow Result[i, j] + Y[i, k] \times Z[k, j]$
    return $(Result)$
end
procedure $=(To, From)$
    if $To.cols \neq From.cols$ or $To.rows \neq From.rows$        2
    then /* Throw an exception */
    else
        for $i = 1$ to $To.rows$ do
            for $j = 1$ to $To.cols$ do
                $To[i, j] \leftarrow From[i, j]$
end
```

Figure 16.1: Matrix multiplication using overloaded operators.

Each of these is considered next in some detail, using the example shown in Figure 16.1. In this program, variables A, B, and C are of type *Matrix*. For simplicity, we assume all matrices are of size $N \times N$. The $\times$ and $=$ operators are overloaded to perform matrix multiplication and assignment, respectively, using the function and procedure provided in Figure 16.1.

High-Level Language Features

High-level languages contain features that offer flexibility and generality at the cost of some runtime efficiency. Optimizing compilers attempt to recover performance for such features in the following ways.

- Perhaps it can be shown that the feature is not used by some portion of a program.

  In the example of Figure 16.1, suppose that the *Matrix* type is subtyped into *SymMatrix*—with definitions of the *Matrix* methods optimized for symmetric matrices. If A and B are actually of type *SymMatrix*, then languages that offer **virtual function dispatch** are obligated to call the most specialized method for an actual type. However, if compiler can show that $\times$ and $=$ are not redefined in any subclass of *Matrix*, then the result of a virtual function dispatch on

```
for $i = 1$ to $N$ do                                                3
    for $j = 1$ to $N$ do
        $Result[i, j] \leftarrow 0$
        for $k = 1$ to $N$ do
            $Result[i, j] \leftarrow Result[i, j] + B[i, k] \times C[k, j]$  4
for $i = 1$ to $N$ do                                                5
    for $j = 1$ to $N$ do
        $A[i, j] \leftarrow Result[i, j]$
```

Figure 16.2: Inlining the overloaded operators.

these methods can be predicted for any object whose compile-time type is at least *Matrix*.

Based on such analysis, **method inlining** expands the method definitions in Figure 16.1 at their call sites, substituting the appropriate parameter values. Shown in Figure 16.2, the resulting program avoids the overhead of function calls. Moreover, the code is now specially tailored for its arguments, whose row and column sizes are N. Program optimization can then eliminate the tests at Steps **1** and **2**.

- Perhaps it can be shown that a language-mandated operation is not necessary. For example, Java insists on subscript checks for array references and type-checks for **narrowing casts**. Such checks are unnecessary if an optimizing compiler can determine the outcome of the result at compile-time. When code is generated for Figure 16.2, **induction variable analysis** can show that i, j, and k all stay within the declared range of matrices A, B, and C. Thus, subscript tests can be eliminated for those arrays.

  Even if the outcome is uncertain at compile-time, a test can be eliminated if its outcome is already computed. Suppose the compiler is required to check the subscript expressions for the *Result* matrix at Step **4** in Figure 16.2. Most likely, the code generator would insert a test for each reference of $Result[i, j]$. An optimization pass could determine that the second test is redundant.

Modern software-construction practices dictate that large software systems should be comprised of small, easily written, readily reusable components. As a result, the size of a compiler's **compilation unit**—the text directly presented in one run of the compiler—has has been steadily dwindling. Optimizing compilers therefore consider the problem of **whole-program optimization** (WPO), which requires analyzing the interaction of a program's compilation units. Method inlining—successful in our example—is one example of the benefits of WPO. Even if a method cannot be inlined, WPO can generate a version of the invoked method that is customized to its calling context. In other words, the trend toward reusable code can result in systems that are general but not as efficient as they could be. Optimizing compilers step in to regain some of the lost performance.

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $A[i,j] \leftarrow 0$
        for $k = 1$ to $N$ do                                   6
            $A[i,j] \leftarrow A[i,j] + B[i,k] \times C[k,j]$   7
```

Figure 16.3: Fusing the loop nests.

Target-Specific Optimization

Portability ranks high among the goals of today's programming languages. Ideally, once a program is written in a high-level language, it should be movable without modification to any computing system that supports the language. Architected interpretive languages such as **Java virtual machine** (JVM) support portability nicely—any computer that sports a JVM interpreter can run any Java program. But the question remains—how fast? Although most modern computers are based on RISC principles, the details of their instruction sets vary widely. Moreover, various models for a given architecture can also differ greatly with regard to their storage hierarchies, their instruction timings, and their degree of concurrency.

Continuing with our example, the program shown in Figure 16.2 is an improvement over the version shown in Figure 16.1, but it is possible to obtain better performance. Consider the behavior of the matrix *Result*. The values of *Result* are computed in the loop nest at Step **3**—one element at a time. Each of these elements is then copied from *Result* to *A* by the loop nest at Step **5**. Poor performance can be expected on any **non-uniform memory access** (NUMA) system that cannot accommodate *Result* at its fastest storage level. Better performance can be obtained if the data is stored directly in *A*. Optimizing compilers that feature **loop fusion** can identify that the outer two loop nests at Steps **3** and **5** are structurally equivalent. **Dependence analysis** can show that each element of *Result* is computed independently. The loops can then be fused to obtain the program shown in Figure 16.3, in which the *Result* matrix is eliminated.

Artifacts of program translation

In the process of translating a program from a source to target language, a compiler pass can introduce spurious computations. As discussed in Chapter Chapter:global:fifteen, compilers try to keep frequently accessed variables in fast registers. Thus, it is likely that the iteration variables in Figure 16.3 are kept in registers. Figure 16.4 shows the results of straightforward code generation for the loops in Figure 16.3.

- The loops contain instructions at Steps **10**, **11**, and **12** that save the iteration variable register in the named variable. However, this particular program does not require a value for the iteration variables when the loops are finished. Thus, such saves are unnecessary. In Chapter Chapter:global:fifteen,

```
$r_i \leftarrow 1$
while $r_i \leq N$ do
    $r_j \leftarrow 1$
    while $r_j \leq N$ do
        $r_A \leftarrow \star(Addr(A) + (((r_i - 1) \times N + (r_j - 1))) \times 4)$
        $\star(r_a) \leftarrow 0$
        $r_k \leftarrow 1$
        while $r_k \leq N$ do
            $r_A \leftarrow \star(Addr(A) + (((r_i - 1) \times N + (r_j - 1))) \times 4)$      8
            $r_B \leftarrow \star(Addr(B) + (((r_i - 1) \times N + (r_k - 1))) \times 4)$
            $r_C \leftarrow \star(Addr(C) + (((r_k - 1) \times N + (r_j - 1))) \times 4)$
            $r_{sum} \leftarrow r_A$
            $r_{prod} \leftarrow r_B \times r_C$
            $r_{sum} \leftarrow r_{sum} + r_{prod}$
            $r_A \leftarrow \star(Addr(A) + (((i - 1) \times N + (j - 1))) \times 4)$      9
            $\star(r_a) \leftarrow r_{sum}$
            $r_k \leftarrow r_k + 1$
        $k \leftarrow r_k$      10
        $r_j \leftarrow r_j + 1$
    $j \leftarrow r_j$      11
    $r_i \leftarrow r_i + 1$
$i \leftarrow r_i$      12
```

Figure 16.4: Low-level code sequence.

register allocation can avoid such saves if the iteration variable can be allocated a register without spilling.

- Because code generation is mechanically invoked for each program element, it is easy to generate redundant computations. For example, Steps **8** and **9** compute the address of $A[i, j]$. Only one such computation is necessary.

Conceivably, the code generator could be written to account for these conditions as it generates code. Superficially, this may seem desirable, but modern compiler design practices dictate otherwise.

- Such concerns can greatly complicate the task of writing the code generator. Issues such as instruction selection and register allocation are the primary concerns of the code generator. Generally, it is wiser to craft a compiler by combining simple, single-purpose transformations that are easily understood, programmed, and maintained. Each such transformation is often called a **pass** of the compiler.

- There are typically many portions of a compiler that can generate superfluous code. It is more efficient to write one compiler pass that is dedicated to the removal of unnecessary computations than to duplicate that functionality throughout the compiler.

In this chapter, we study two program optimizations that remove unnecessary code: **dead-code elimination** and **unreachable code elimination**. Even the most basic compiler typically includes these passes, if only to clean up after itself.

Continuing with our example, consider code generation for Figure 16.3. Let us assume that each array element occupies 4 bytes, and that subscripts are specified in the range of $1..N$. The code for indexing the array element $A[i, j]$ becomes

$$Addr(A) + (((i-1) \times \ N + (j-1))) \times 4$$

which takes

4	integer "+" and "−"
2	integer "×"
6	integer operations

Since Step **7** has 4 such array references, each execution of this statement takes

16	integer "+" and "−"
8	integer "×"
3	loads
1	floating "+"
1	floating "×"
1	store
30	instructions

Moreover, the loop contains 2 floating-point instructions and 24 fixed-point instructions. On superscalar architectures that support concurrent fixed- and floating-point instructions, this loop can pose a severe bottleneck for the fixed-point unit.

The computation can be greatly improved by optimizations that are described in this chapter.

- **Loop-invariant detection** can deterine that the (address) expression $A[i, j]$ does not change at Step **7**.

- **Reduction in strength** can replace the address computations for the matrices with simple increments of an index variable. The iteration variables themselves can disappear, with loop termination based on the subscript addresses.

The result of applying these optimizations on the innermost loop is shown in Figure 16.5. The inner loop now contains 2 floating-point and 2 fixed-point operations—a balance that greatly improves the loop's performance on modern processors.

```
$FourN \leftarrow 4 \times N$
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $a \leftarrow \&(A[i,j])$
        $b \leftarrow \&(B[i,1])$
        $c \leftarrow \&(C[1,j])$
        while $b < \&(B[i,1]) + FourN$ do
            $\star a \leftarrow \star a + \star b \times \star c$
            $b \leftarrow b + 4$
            $c \leftarrow c + FourN$
```

Figure 16.5: Optimized matrix multiply.

Program Restructuring

Some optimizing compilers attempt to improve a program by prescribing better algorithms and data structures.

- Some programming languages contain constructs for which diverse implementations are possible. For example, Pascal and SETL offer the *set* type-constructor. An optimizing compiler could choose a particular implementation for a set, based on the the predicted construction and use of the set.
- If the example in Figure 16.3 can be recognized as matrix-multiply, then an optimizing compiler could replace the code with a better algorithm for matrix-multiply. Except in the most idiomatic of circumstances, it is difficult for a compiler to recognize an algorithm at work.

16.1.2 Organization

In this section, we briefly examine the organization of an optimizing compiler—how it represents and processes programs.

Use of an intermediate language

The task of creating an optimizing compiler can be substantial, especially given the extent of automation available for a compiler's other parts. Ideally, the design, cost, and benefits of an optimizing compiler can be shared by multiple source languages and target architectures. An **intermediate language** (IL) such as JVM can serve as a focal point for the optimizer, as shown in Figure 16.6. The optimizer operates on the IL until the point of code generation, when target code is finally generated. Attaining this level of machine-independent optimization is difficult, as each target machine and source language can possess features that demand special treatment. Nonetheless, most compiler designs resemble the structure shown in Figure 16.6.

The techniques we present in this chapter are sufficiently general to accommodate any program representation, given that the following information is available.

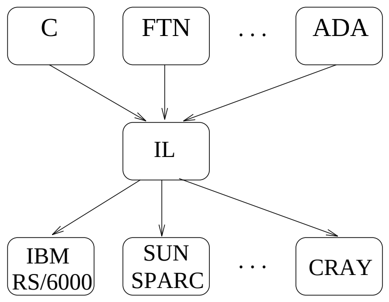

Figure 16.6: Program optimization for multiple sources and targets.

- The operations of interest must be known and explicitly represented. For example, if the program contains a code sequence that computes $a + b$, the compiler might also be called upon to perform this computation.
- The effects of the program's operations must be known or approximated. For example, the optimizer may require knowing the set of global variables that can be affected by a given method. If this set is not known, it can be approximated by the set of all global variables.
- The potential ordering of operations may be of interest. Within a given method, its operations can usually be regarded as the nodes of a **control flow graph**, whose edges show the potential flow from one operation to the next. Any path taken by a program at runtime is a path of this control flow graph. Of course, the graph may contain paths that are rarely taken, but these are necessary to understand the program's potential behavior. Moreover, the graph can contain paths that can never be taken—consider a predicate whose outcome is always false.

As discussed in Section 16.2, each optimization pass has its own view of the important aspects of the program's operations. We stress here that the potential run-time behavior of a program is approximated at compile-time, both in terms of the operations' effects and order.

A series of small passes

As discussed in Section 16.1.1, it is expedient to organize an optimizing compiler as a series of *passes*. Each pass should have a narrowly defined goal, so that its scope is easily understood. This facilitates development of each pass as well as the integration of the separate passes into an optimizing compiler. For example, it should be possible to run the *dead-code elimination* pass at any point to remove useless code. Certainly this pass would be run prior to code generation. However, the effectiveness of an individual pass as well as the overall speed of the optimizing compiler can be improved by sweeping away unnecessary code.

To facilitate development and interoperability, it is useful for each pass to accept and produce programs in the IL of the compiler. Thus, the program transformations shown in Figures 16.1, 16.2, 16.3, and 16.5 could be carried out on the compiler's intermediate form. In research compilers, a source language (such as C) can serve as a nice IL, because a C compiler can take the optimized IL form to native code when the optimizing compiler is finished. Alternatively JVM can serve as an IL for high-level program optimization, although JVM is certainly biased toward a Java view of what programs can do.

Unfortunately, the passes of a compiler can interact in subtle ways. For example, **code motion** can rearrange a program's computations to make better use of a given architecture. However, as the distance between a variable's definition and its uses grows, so does the pressure on the register allocator. Thus, it is often the case that a compiler's optimizations are at odds with each other. Ideally, the passes of an optimizing compiler should be fairly independent, so that they can be reordered and retried as situations require.

```
$u \leftarrow 5$
repeat
    if $r$
    then
        $v \leftarrow 9$
        if $p$
        then  $u \leftarrow 6$
        else  $w \leftarrow 5$
        $x \leftarrow v + w$
    else  $y \leftarrow v + w$
    $u \leftarrow 7$
    repeat
        if $q$
        then
            $z \leftarrow v + w$     13
    until $r$
    $v \leftarrow 2$
until $s$
```

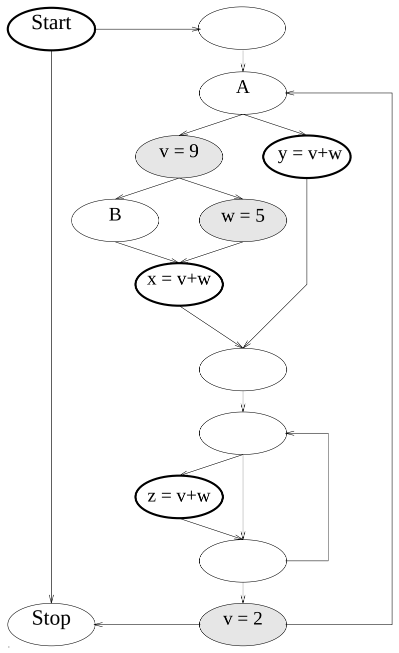

Figure 16.7: A program and its control flow graph.

16.2 Data Flow Analysis

As discussed in Section 16.1, an optimizing compiler is typically organized as a series of **passes**. Each pass may require approximate information about the program's run-time behavior. **Data flow frameworks** offer a structure for such analysis. We begin in Section 16.2.1 by describing several **data flow problems**—their specification and their use in program optimization. In Section 16.2.2, we formalize the notion of a data flow framework. Section 16.2.3 describes how to evaluate a data flow framework. In this section, we use the program shown in Figure 16.7(a) as an example. The same program is represented in Figure 16.7(b) by its **control flow graph**—the instructions are contained in the nodes, and the edges indicate potential branching within the program.

16.2.1 Introduction and Examples

In Section 16.2.2, we offer a more formal presentation of data flow frameworks. Here, we discuss data flow problems informally, examining a few popular optimization problems and reasoning about their data flow formulation. In each problem, we are interested in the following.

- What is the effect of a code sequence on the solution to the problem?

- When branching in a program converges, how do we summarize the solution so that we need not keep track of branch-specific behavior?
- What are the best and worst possible solutions?

Local answers to the above questions are combined by data flow analysis to arrive at a global solution.

Available Expressions

Figure 16.7 contains several computations of the expression $v + w$. If we can show that the particular value of $v + w$ computed at Step **13** is already *available*, then there is no need to recompute the expression at Step **13**. More specifically, an expression *expr* is **available** at edge e of a flow graph if the past behavior of the program necessarily includes a computation of the value of *expr* at edge e. The **available expressions** data flow problem analyzes programs to determine such information.

To solve the **available expressions** problem, a compiler must examine a program to determine that expression *expr* is available at edge e, regardless of how the program arrives at edge e. Returning to our example, $v + w$ is **available** at Step **13** if every path arriving at Step **13** computes $v + w$ without a subsequent change to v or w.

In this problem, an instruction affects the solution if it computes $v + w$ or if it changes the value of v or w, as follows.

- The Start node of the program is assumed to contain an implicit computation of $v + w$. For programs that initialize their variables, $v + w$ is certainly available after node Start. Otherwise, although $v + w$ is uninitialized, the compiler is free to assume the expression has *any* value it chooses.
- A node of the flow graph that computes $v + w$ makes $v + w$ available.
- A node of the flow graph that assigns v or w makes $v + w$ not available.
- All other nodes have no effect on the availability of $v + w$.

In Figure 16.7(b), the shaded nodes make $v + w$ unavailable. The nodes with dark circles make $v + w$ available. From the definition of this problem, we summarize two solutions by assuming the worst case. At the input to node A, the path from Start contains an implicit computation of $v + w$; on the loop edge, $v + w$ is not available. At the input to node A, we must therefore assume that $v + w$ is *not* available.

Based on the above reasoning, information can be pushed through the graph to reach the solution shown on each edge of Figure 16.8. In particular, $v + w$ is avaiable on the edge entering the node that represents Step **13** in Figure 16.7(a). Thus, the program can be optimized by eliminating the recomputation of $v + w$. Similarly, the address computation for $A[i, j]$ need not be performed at Step **9** in Figure 16.4—the expression is available from the computation at Step **8**.

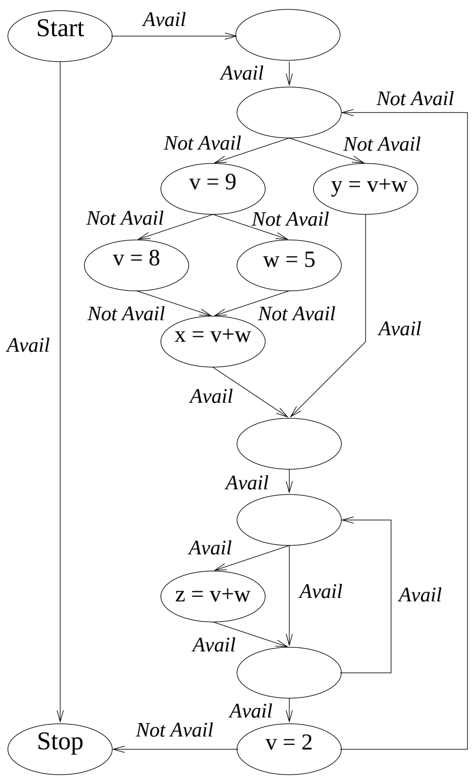

Figure 16.8: Global solution for availability of the expression $v + w$.

In this example, we explored the availability of a single expression $v + w$. Better optimization could obviously result from determining that an expression is available. In an optimizing compiler, one of the following situations usually holds.

- The compiler identifies an expression such as $v + w$ as *important*, in the sense that eliminating its computation can significantly improve the program's performance. In this situation, the optimizing compiler may selectively evaluate the availability of a single expression.
- The compiler may compute availability of *all* expressions, without regard to the importance of the results. In this situation, it is common to formulate a *set* of expressions and compute the availability of its members. Section 16.2.2 and Exercise 8 considers this in greater detail.

Live Variables

We next examine an optimization problem related to register allocation. As discussed in Chapter Chapter:global:fifteen, k registers suffice for a program whose

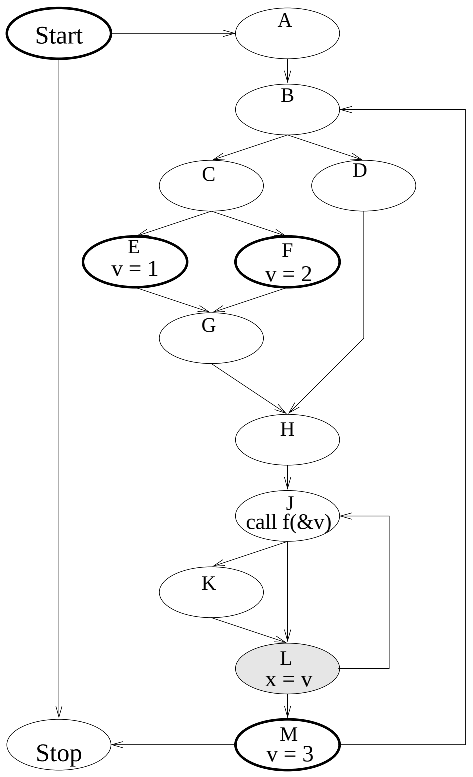

Figure 16.9: Example flow graph for liveness. The function `f` potentially assigns v but does not read its value.

interference graph is k-colorable. In this graph, each node represents one of the program's variables. An edge is placed between two nodes if their associated variables are simultaneously *live*. A variable v is **live** at control flow graph edge e if the future behavior of the program can reference the value of v that is present at edge e. In other words, the value of a live variable is potentially of future use in the program. Thus, register allocation relies on **live variable analysis** to build the interference graph.

In the available-expressions problem, information was pushed forward through the control flow graph. In the live-variables problem, the potential behavior of a program is pushed backward through the control flow graph. In the control flow graph shown in Figure 16.9, consider the liveness of variable v. The dark-circled nodes contain uses of v—they represent future behavior that makes v live. On the other hand, the shaded nodes destroy the current value of v. Such nodes represent future behavior that makes v not live. At the `Stop` node, we may assume v is dead since the program is over. If the value of v were to survive an execution, then its

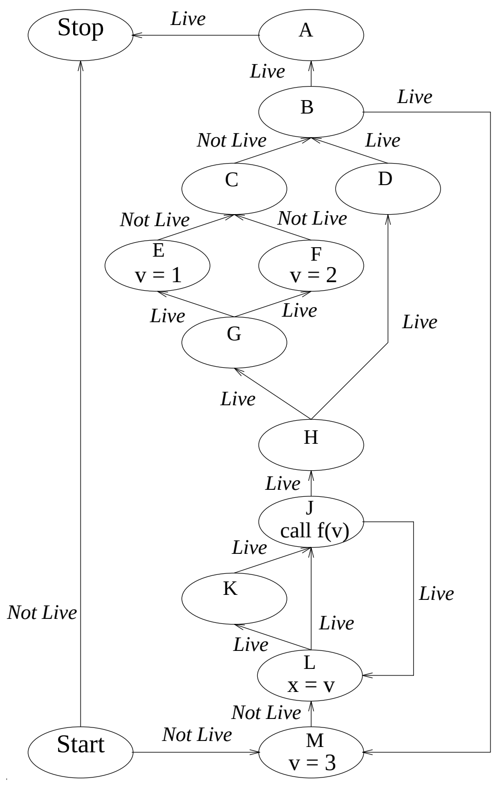

Figure 16.10: Solution for liveness of v.

value should be printed by the program.

Figure 16.9 shows a node with a `call` instruction. How does this node affect the liveness of v? In this example, we assume that the function `f` potentially assigns v but does not use its value. Thus, the invoked function does not make v live. However, since we cannot be certain that `f` assigns v, the invoked function does not make v dead. This particular node has no effect on the solution to live variables.

Based on the definition of this problem, common points of control flow cause v to be live if *any* future behavior causes v to be live. The solution for liveness of v is shown in Figure 16.10—the control flow edges are reversed to show how the computation is performed. It is clearly to an optimizing compiler's advantage to show that a variable is *not* live. Any resources associated with a dead variable can be reclaimed by the compiler, including the variable's register or local JVM slot.

An optimizing compiler may seek liveness information for one variable or for a set of variables. Exercise 9 considers the computation for a set.

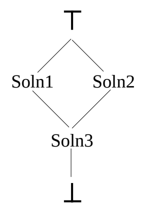

Figure 16.11: A meet lattice.

16.2.2 Formal Specification

We have introduced the notion of a data flow framework informally, relying on examples drawn from optimizing compilers. In this section, we formalize the notion of a **data flow framework**. As we examine these details, it will be helpful to refer to the problems discussed in Section 16.2.1.

A data flow framework has the following components.

- A **flow graph**. This directed graph's nodes typically represent some aspect of a program's behavior. For example, a node may represent a nonbranching sequence of instructions, or an entire procedure. The graph's edges represent a relation over the nodes. For example, the edges may indicate potential transfer of control by branching or by procedure call. We assume the graph's edges are oriented in the "direction" of the data flow problem.
- A **meet lattice**. This is a mathematical structure that describes the solution space of the data flow problem and designates how to combine multiple solutions in a safe (conservative) way. It is convenient to present such lattices as Hasse diagrams: to find the meet of two elements, you put one finger on each element, and travel down the page until your fingers first meet. For example, the lattice in Figure 16.11 indicates that when *Soln*1 and *Soln*2 must be combined, then their solutions are to be approximated by *Soln*3.
- A set of **transfer functions**. These model the behavior of a node with respect to the optimization problem under study. Figure 16.12 depicts a generic transfer function. A transfer function's input is the solution that holds on entry to the node, so that the function's output specifies how the node behaves given the input solution.

We next examine each of these components in detail.

Data Flow Graph

The **data flow graph** is constructed for an optimization problem, so that evaluation of this graph produces a solution to the problem.

- Transfer functions are associated with each node;

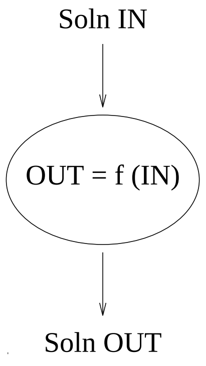

Figure 16.12: A node's transfer function.

- Information converging at a node is combined as directed by the meet lattice;
- Information is propagated through the data flow graph to obtain a solution.

For the problems considered here, a flow graph's nodes represent some component of a program's behavior and its edges represent potential transfer of control between nodes. In posing an optimization problem as a data flow framework, the resulting framework is said to be

- **forward**, if the solution at a node can depend only on the program's past behavior. Evaluating such problems involves propagating information *forward* through the flow graph. Thus, the control flow graph in Figure 16.7(b) served as the data flow graph for availability analysis of $v+w$ in the program of Figure 16.7.
- **backward**, if the solution at a node can depend only on the program's future behavior. The live variables problem introduced in Section 16.2.1 is such a problem. For live variables, a suitable data flow graph is the reverse control flow graph, as shown in Figure 16.10.
- **bidirectional**, if both past and future behavior is relevant.

In this chapter, we discuss only forward or backward problems; moreover, we assume that edges in the data flow graph are oriented in the direction of the data flow problem. With this assumption, information always propagates in the direction of the data flow graph's edges. It is convenient to augment data flow graphs with a Start and Stop node, and an edge from Start to Stop, as shown in Figure 16.7(b).

In compilers where space is at a premium, nodes of a control flow graph typically correspond to the *maximal* straight-line sequences—the **basic blocks**—of a program. While this design conserves space, program analysis and optimization must then occur at two levels: *within* and *between* the basic blocks. These two levels are called **local** and **global** data flow analysis, respectively. An extra level of analysis complicates and increases the expense of writing, maintaining, and documenting an optimizing compiler. We therefore formulate data flow graphs whose nodes model the effects of perhaps a single JVM or MIPS instruction. In a production compiler, the choice of node composition may dictate otherwise.

Meet Lattice

As with all lattices, the **meet lattice** represents a partial order imposed on a set. Formally, the meet lattice is defined by the tuple

$$(A, \top, \bot, \preceq, \wedge)$$

which has the following components.

- A **solution space** A. In a data flow framework, the relevant set is the space of all possible solutions to the data flow problem. Exercise 9 considers the live-variables problem, posed over a set of n variables. Since each variable is either live or not live, the set of possible solutions contains 2^n elements. Fortunately, we need not enumerate or represent all elements in this large set. In fact, some data flow problems have an infinite solution space.

- The **meet operator** $\wedge$. The partial order present in the lattice directs how to combine (summarize) multiple solutions to a data flow problem. In Figure 16.8, the first node of the outer loop receives two edges—$v+w$ is available on one edge and not available on the other. The meet operation ($\wedge$) serves to summarize the two solutions. Mathematically, $\wedge$ is associative, so multiple solutions are easily summarized by applying $\wedge$ pairwise in any order.

- Distinguished elements $\top$ and $\bot$. Lattices associated with data flow frameworks always include the following distinguished elements of A.

  - $\top$ intuitively represents the solution that allows the most optimization.
  - $\bot$ intuitively represents the solution that prevents or inhibits optimization.

- The comparison operator $\preceq$. The meet lattice includes a reflexive partial order, denoted by $\preceq$. Given two solutions a and b—both from set A—it must be true that that $a \preceq b$ or $a \not\preceq b$. If $a \preceq b$, then solution a is no better than solution b. Further, if $a \prec b$, then solution a is strictly worse than b—optimization based on solution a will not be as good as with solution b. If $a \not\preceq b$, then solutions a and b are incomparable.

  For example, consider the problem of live variables, computed for the set of variables $\{ v, w \}$. As discussed in Section 16.2.1, the storage associated with variables found *not* to be live can be reused. Thus, optimization is improved when fewer variables are found to be live. Thus, the set $\{ v, w \}$ is worse than the set $\{ v \}$ or the set $\{ w \}$. However, the solution $\{ v \}$ cannot be compared with the set $\{ w \}$. In both cases, one variable is live, and data flow analysis cannot prefer one to the other for live variables.

Property	Explanation
$a \wedge a = a$	The combination of two identical solutions is trivial.
$a \preceq b \Longleftrightarrow a \wedge b = a$	If a is worse than b, then combining a and b must yield a; if $a = b$, then the combination is simply a, as above.
$a \wedge b \preceq a$ $a \wedge b \preceq b$	The combination of a and b can be no better than a or b.
$a \wedge \top = a$	Since $\top$ is the best solution, combining with $\top$ changes nothing.
$a \wedge \bot = \bot$	Since $\top$ is the worst solution, any combination that includes $\bot$ will be $\bot$.

Figure 16.13: Meet lattice properties.

At this point, it is important to develop an intuitive understanding of the lattice—especially its distinguished elements $\top$ and $\bot$. For each analysis problem, there is some solution that admits the greatest amount of optimization. This solution is always $\top$ in the lattice. Recalling available expressions, the best solution would make *every* expression available—all recomputations could then be eliminated. Correspondingly, $\bot$ represents the solution that admits the least amount of optimization. For available expressions, $\bot$ implies that *no* expressions can be eliminated.

Mathematically, the meet lattice has the properties shown in Figure 16.13.

Transfer Functions

Finally, our data flow framework needs a mechanism to describe the effects of a fragment of program code—specifically, the code represented by a path through the flow graph. Consider a single node of the data flow graph. A solution is present on entry to the node, and the transfer function is responsible for converting this input solution to a value that holds after the node executes. Mathematically, the node's behavior can be modeled as a *function* whose domain and range are the meet lattice A. This function must be **total**—defined on every possible input. Moreover, we shall require the function to behave **monotonically**—the function cannot produce a better result when presented with a worse input. Consider the available expressions problem, posed for the expressions $v+w$ and $a+b$. Figure 16.14 contains fragments of a program and explains the transfer function that models their effects.

Fragment	Transfer Function	Explanation
y = v+w	$f(in) = in \cup \{$ "v+w" $\}$	Regardless of which expressions were available on entry to this node, expression "v+w" becomes available after the node. That status of expression $a+b$ is not affected by this node.
v = 9	$f(in) = in - \{$ "v+w" $\}$	Regardless of which expressions were available on entry to this node, expression "v+w" is *not* available after the node, because the assignment to v potentially changes the value of "v+w", and the node includes no recomputation of this expression. The status of expression $a+b$ is not affected.
printf("hello")	$f(in) = in$	This node affects no expression; thus, the solution on exit from the node is identical to the solution on entry.

Figure 16.14: Data flow transfer functions.

Because a transfer functions are *mathematical*, they can model not only the effects of a single node but also the effects along a *path* through a program. If a node with transfer function f is followed by a node with transfer function g, then the cumulative effect of both nodes on input a is captured by $g(f(a))$. In other words, program behavior—brief or lengthy—is captured by a transfer function. A transfer function can be applied to any lattice value a to obtain the compile-time estimation of the program fragment's behavior given the conditions represented by a.

16.2.3 Evaluation WARNING this subsection is incomplete

Having discussed how to pose an optimization problem in a data flow framework, we next turn to evaluating such frameworks. As shown in Figure 16.12, each node of the data flow graph asserts a transfer function. Once we know the input value to a transfer function, its output value is easily computed. When multiple solutions converge at a node, the meet lattice serves to summarizes the solutions, as shown in Figure 16.11. Thus, we can compute the input to a node as the meet of all the outputs that feed the node.

Figure 16.15: Iterative evaluation of a data flow framework.

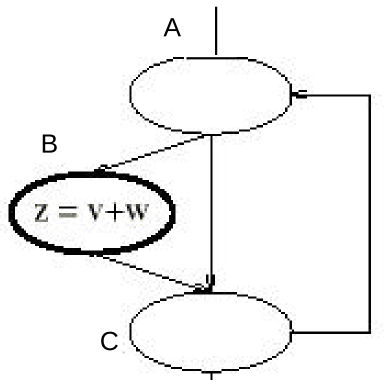

Figure 16.16: Is $v + w$ available throughout this loop?

An algorithm for evaluating a data flow framework is shown in Figure 16.15.

Two issues: initialization – is it OK to do it to top? and termination – how do we know that the loop terminates?

Initialization

The problem is that the solution at a given node depends on the solution at other nodes, as dictated by the edges of the data flow graph. In fact, the solution at a node can depend directly or indirectly *on itself*. Figure 16.7 contains a loop that is shown in Figure 16.16. The solution in Figure 16.8 shows that $v + w$ is available everywhere inside the loop. But how is this determined? The nodes in Figure 16.16 have the following transfer functions.

$$\begin{aligned} f_A(in_A) &= in_A \\ f_B(in_B) &= \top \\ f_c(in_C) &= in_C \end{aligned}$$

Let in_{loop} denote the input to the loop—the input to node A from outside the loop. The inputs to A and C are computed as follows.

$$\begin{aligned} in_C &= f_B(in_B) \wedge f_A(in_A) \\ &= \top \wedge f_A(in_A) \\ &= f_A(in_A) \\ in_A &= f_C(in_C) \wedge in_{loop} \\ &= in_C \wedge in_{loop} \\ &= f_A(in_A) \wedge in_{loop} \end{aligned}$$

In other words, the input to node A depends on the output of node A! This seems a contradiction, unless we reason that the first time we evaluate A's transfer function, we can assume some prior result. Computationally, we have the following choices concerning the prior result.

$$\begin{aligned} f_A(in_A) &= \perp \\ f_A(in_A) &= \top \end{aligned}$$

It seems safe to assume that $v + w$ is *not* available previously—that is, $f_A(in_A) = \perp$. Based on this assumption, $v + w$ is not available anywhere except in node B. It is more daring to assume that $v + w$ *is* available—$f_A(in_A) = \top$. Based on this assumption, we obtain the result shown in Figure 16.8, with $v + w$ available everywhere in the inner loop.

Termination

In a **monotone data flow framework**, each transfer function f obeys the rule

$$a \preceq b \Rightarrow f(a) \preceq f(b)$$

for all pairs of lattice values a and b. Note that if the lattice does not relate a and b, the rule is trivially satisfied.

16.2.4 Application of Data Flow Frameworks

We have thus far studied the formulation and evaluation fo data flow frameworks. We examine next a series of optimization problems and discuss their solution as data flow problems.

Available Expressions

Section 16.2.1 contained an informal description of the available expressions problem for the expression $v + w$. Suppose a compiler is interested in computing the availability of a *set* of expressions, where the elements of the set are chosen by examining the expressions of a program. For example, a program with the quadratic formula

$$-b + \sqrt{\frac{b^2 - 4ac}{2a}}$$

contains the variables a, b, and c. Each variable is itself a simple expression. However, each variable is trivially available by loading its value. The more ambitious expressions in this formula include $-b$, b^2, ac, $4ac$, $b^2 - 4ac$. We might prefer to show that $b^2 - 4ac$ is available, because this expression is costly to compute. However, the availability of any of these expressions can improve the program's performance.

We can generalize the available expressions problem to accommodate a *set* of expressions, as follows.

- This is a forward problem. The data flow graph is simply the control flow graph.
- If S is the set of interesting expressions, then the solution space A for this problem is 2^S—the **power set** of S. In other words, a given expression is or is not present in any particular solution.
- The most favorable solution is S—every expression is available. Thus, $\top = S$ and $\bot = \emptyset$.
- The transfer function at node Y can be formulated as

  $$f_Y(in) = (in - Kill_Y) \cup Gen_Y$$

  where

  - $Kill_Y$ is the set of expressions containing a variable that is (potentially) modified by node Y.
  - Gen_Y is the set of expressions computed by node Y.

  If the flow graph's nodes are sufficiently small—say, a few instructions—then it is possible to limit each node's effects to obtain $Kill_Y \cap Gen_Y = \emptyset$ at each node Y. In other words, none of these small nodes both generates an expression and kills it. Exercise 6 explores this issue in greater detail.
- The meet operation is accomplished by set intersection, based on the following reasoning.
  - An expression is available only if all paths reaching a node compute the expression.
  - Formally, we require that $\top \wedge a = a$. If $\top$ is the set of all expressions, then meet must be set intersection.

As discussed in Exercise 7, some simplfications can be made when computing available expressions for a single expression. Available expressions is one of the so-called **bit-vectoring data flow problems**. Other such problems are described in Exercises 9, 10, and 11.

Constant Propagation

In this optimization, a compiler attempts to determine expressions whose value is constant over all executions of a program. Most programmers do not intentionally introduce constant expressions. Instead, such expressions arise as artifacts of program translation, as discussed in Section 16.1. Interprocedurally, constants develop when a general method is invoked with arguments that are constant.

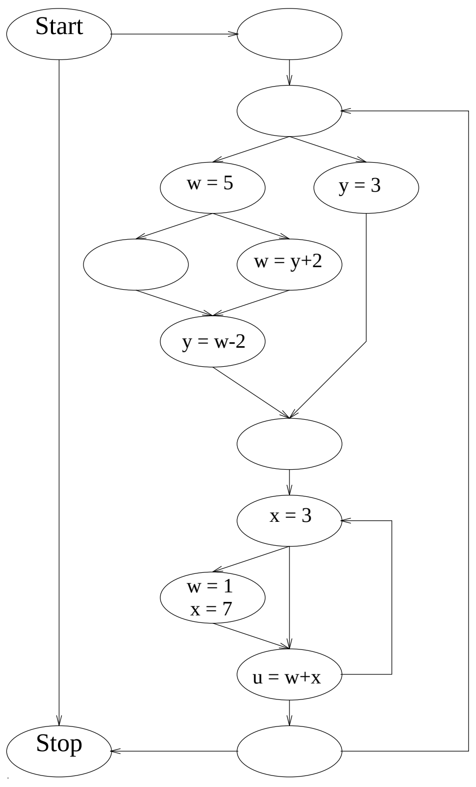

Figure 16.17: A program for constant propagation.

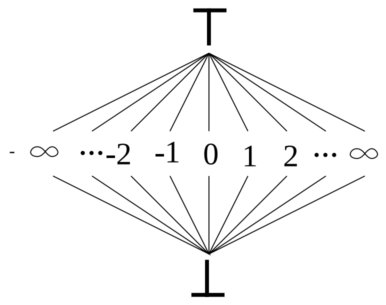

Figure 16.18: Lattice for constant propagation---one expression only.

Consider the program whose control flow graph is shown in Figure 16.17. Some nodes assign constant values to variables. In other nodes, these constant values may combine to create other constant values. We can describe constant propagation as a data flow problem as follows.

- Without loss of generality, we pose constant propagation over a program's variables. If the program contains an expression or subexpression of interest, this can be assigned to a temporary variable. Constant propagation can then try to discover a constant value for the temporary.
- For a single variable, we formulate the three-tiered lattice shown in Figure 16.18.
  - $\top$ means that the variable is considered a constant of some (as yet) undetermined value.
  - $\perp$ means that the expression is not constant.
  - Otherwise, the expression has a constant value found in the middle layer.
- As shown in Figure 16.19, each edge in the data flow graph carries a solution for each variable of interest.
- The meet operator is applied using the lattice shown in Figure 16.18. The lattice is applied separately for each variable of interest.
- The transfer function at node Y interprets the node by substituting the node's incoming solution for each variable used in the node's expression. Suppose node v is computed using the variables in set U. The solution for variable v after node Y is computed as follows.
  - If any variable in U has value $\perp$, then v has value $\perp$.
  - Otherwise, if any variable in U has value $\top$, then v has value $\top$.
  - Otherwise, all variables in U have constant value. The expression is evaluated and the constant value is assigned to v.

Figure 16.19■[1]hows the solution for constant propagation on the program of Figure 16.17.

This program demonstrates some interesting properties of constant propagation. Although x is uninitialized prior to executing the node that assigns its value, constant propagation is free to assume it has the value 3 throughout the program. This occurs because an uninitialized variable has the solution $\top$. When summarized by the meet operator, $\top \wedge 3 = 3$. Although the uninitialized variable may indicate a programming error, the optimizer can assume an uninitialized variable has any value it likes without fear of contradiction. If the programming language at hand has semantics that insist on initialization of all variables—say to 0—then this must be represented by an assignment to these variables at the Start node.

[1]*Production note: s*

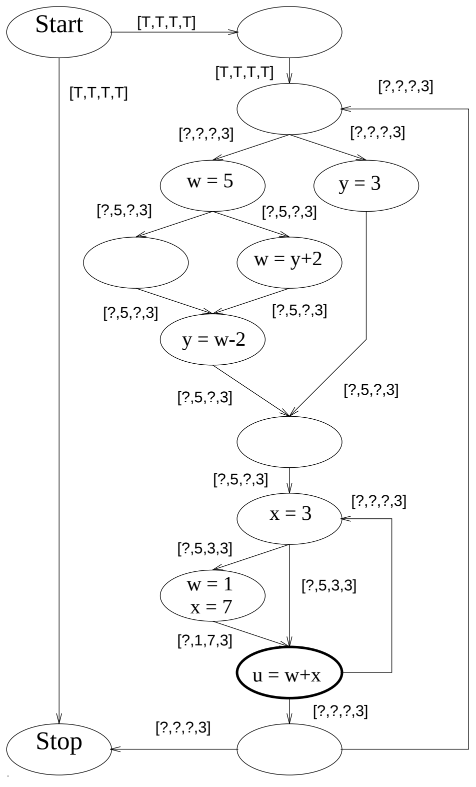

Figure 16.19: Constant propagation. Each edge is labeled by a tuple that shows the solution for $[u, w, x, y]$.

Another observation is that constant propagation failed to determine that `u` has the value 8 at the dark-circled node in Figure 16.19. Prior to this node, both `w` and `x` have constant values. Although their sum is the same, the individual values for these variables differ. Thus, when a meet is taken at the dark-circled node, `w` and `x` both appear to be $\perp$. For efficiency, data flow analysis computes its solution based on edges and not paths into a node. For some data flow problems, this approach provides the best solution possible—such is the case for the bit-vectoring data flow problems discussed in Exercises 12 and 18. Unfortunately, the example in Figures 16.17 and 16.19 shows that constant propagation does not compute the best solution.

16.3 Advanced Optimizations

16.3.1 SSA Form

Definition

Construction

16.3.2 SSA-based Transformations

Constant Propagation

Value Numbering

Code Motion

16.3.3 Loop Transformations

Should we even go here?

Summary

hello

Exercises

1. Recall the transformation experienced by the inner loops as the program in Figure 16.1 was optimized into the program shown in Figure 16.5. Apply these same transformations to the outer loops of Figure 16.5.

2. In Section 16.1.2, JVM is proposed as an intermediate form. Compare and contrast your rendering of the program shown in Figure 16.5 in the following intermediate forms.

   (a) The C programming language
   (b) JVM
   (c) The MIPS instruction set

3. Compile—automatically or by-hand—the program shown in Figure 16.5 to generate the intermediate forms of Exercise 2. Compare the sizes of the representations.

4. Consider each of the following properties of a proposed **intermediate language** (IL). Explain how each property is or is not found in each IL of Exercise 2.

   (a) The IL should be a *bona fide* language—it should have a precise syntactic and semantic specification. It should not exist simply as an aggregation of data structures.
   (b) The size of programs expressed in the IL should be as compact as possible.
   (c) The IL should have an expanded, human-readable form.
   (d) The IL should be sufficiently general to represent the important aspects of a wide variety of source languages.
   (e) The IL should be easily and cleanly extensible.
   (f) The IL should be sufficiently general to support generation of efficient code for multiple target architectures.

5. Using a common programming language, construct a program whose control flow graph is the one shown in Figure 16.9.

6. Section 16.2.4 describes how to apply data flow frameworks. The section includes the following formulation for transfer functions in the data flow problem of available expressions.

$$f_Y(in) = (in - Kill_Y) \cup Gen_Y$$

   where

   - $Kill_Y$ is the set of expressions containing a variable that is (potentially) modified by node Y.

- Gen_Y is the set of expressions computed by node Y.

Suppose a flow graph node represents the following code.

$v \leftarrow z$
$u \leftarrow v + w$
$v \leftarrow x$

For the expression $v + w$, the node kills the expression, generates it, and then kills it again. The cumulative effect is to kill the expression, but this requires analyzing the order in which the operations occur inside the node. Describe how to formulate a data flow graph with *smaller* nodes so that the order of operations within the nodes need not be examined.

7. Section 16.2.4 presents the formal definition of a data flow framework to determine the availability of expressions in the set S. Describe the simplifications that result when computing available expressions for a single expression—when $|S| = 1$.

   (a) What is the lattice?
   (b) What are $\top$ and $\bot$?
   (c) What are the transfer functions?
   (d) How is meet performed?

8. The **bit-vectoring data flow problems** earn their name from a common representation for finite sets—the **bit vector**. In this representation, a slot is reserved for each element in the set. If $e \in S$, then e's slot is **true** in the bit vector that represents set S.

   Describe how bit vectors can be applied to the available expressions problem for a set of n expressions. In particular, describe how a bit vector is affected by

   (a) the transfer function at a node
   (b) application of the meet operator
   (c) assignment to $\top$ or $\bot$

9. Explain how liveness of a set of n variables can be computed as a data flow problem.

   (a) Define the formal framework, using the notation in Section 16.2.2. The transfer function at node Y is defined by the following formula (from Section 16.2.4).

   $$f_Y(in) = (in - Kill_Y) \cup Gen_Y$$

   Explain how $Kill_Y$ and Gen_Y are determined for live variables at node Y.

(b) Now consider the use of bit vectors to solve live variables. The transfer function can be implemented as described in Exercise 8. How is the meet operation performed? What are $\top$ and $\perp$?

10. Liveness shows that a variable is potentially of future use in a program. The **very busy expressions** problem if an expression's value is *certainly* of future use.

    (a) Is this a forward or backward problem?

    (b) What is the best solution?

    (c) Describe the effects of a node on an expression.

    (d) How are solutions summarized at common control flow points?

    (e) How would you determine liveness for a *set* of expressions?

11. Reaching defs

12. Four of the data flow problems presented in Section 16.2 and in Exercises 10 and 11 are:

    - Available expressions
    - Live variables
    - Very busy expressions
    - Reaching definitions

    These problems are known as the **bit-vectoring data flow problems**. Summarize these problems by entering each into its proper position in the followoing table.

	Forward	Backward
Any		
All		

    The columns refer to whether information is pushed forward or backward to achieve a solution to the problem. The rows refer to whether information should hold on all paths or any path.

13. A data flow framework is **monotone** if the following formula holds for every transfer function f and lattice values a and b.

    $$x \preceq y \Rightarrow f(x) \preceq f(y).$$

    In other words, a transfer function cannot produce a better answer given a worse input. Prove that the following formula must hold for monotone frameworks.

    $$f(a \wedge b) \preceq f(a) \wedge f(b).$$

14. A data flow framework is **rapid** if defines rapid..then prove that available expressions is rapid.

15. Generalize the proof from Exercise 14 to prove or disprove that all four bit-vectoring data flow problems in Exercise 12 are rapid.

16. Prove or disprove that constant propagation is a rapid data flow problem.

17. A data flow problem is **distributive** if the following formula holds for every transfer function f and lattice values a and b.

$$f(a \wedge b) = f(a) \wedge f(b).$$

Prove or disprove that available expressions (Exercise 8) is a distributive data flow problem.

18. Generalize the proof from Exercise 17 to prove or disprove that all four bit-vectoring data flow problems in Exercise 12 are distributive.

19. Prove or disprove that constant propagation is a distributive data flow problem.

20. Consider generalizing the problem of constant propagation to that of *range analysis*. For each variable, we wish to associate a minimum and maximum value, such that the actual value of the variable (at that site in the program) at runtime is guaranteed to fall between the two values. For example, consider the following program.

```
$x \leftarrow 5$
$y \leftarrow 3$
if $p$
then
    $z \leftarrow x + y$                                  14
else
    $z \leftarrow x - y$                                  15
$w \leftarrow z$
```

After their assignment, variable x has range $5 \ldots 5$ and variable y has range $3 \ldots 3$. The effect of Step **14** gives z the range $8 \ldots 8$. The effect of Step **15** gives z the range $2 \ldots 2$. The assignment for w therefore gets the range $2 \ldots 8$.

(a) Sketch the data flow lattice for a single variable. Be specific about the values for $\top$ and $\bot$.

(b) Is this a forwards or backwards propagation problem?

(c) If the variable v could have range r_1 or r_2, describe how to compute the meet of these two ranges.

21. As defined in Exercise 20, prove or disprove that range analysis is a rapid data flow problem.

22. As defined in Exercise 20, prove or disprove that range analysis is a distribute data flow problem.

23. The number of bits problem

    (a) Prove or disprove that this data flow problem is distribute.

    (b) Prove or disprove that this data flow problem is rapid.