Helmut Seidl, Reinhard Wilhelm, Sebastian Hack

Compiler Design: Analysis and Transformation

Springer

1

Foundations and Intraprocedural Optimization

1.1 Introduction

This section presents some basic techniques to improve the quality of compiler-generated code. The quality metric needs not be a priori fixed. It could be the execution time, the required space, or the consumed energy. This book, however, is primarily concerned with methods to improve the *execution time* of programs.

We now give several examples of how to improve the execution time of programs. One strategy to improve the efficiency of programs is to avoid *superfluous* computations. A computation may be *superfluous* when it has already been performed, and when a repetition would provably always produce the same result. The compiler can avoid this recomputation of the same result if it takes care to store the result of the first computation. The recomputation can then be avoided by accessing this stored value.

The execution time of a program can be also reduced if some of the computations can already be done at compile time. *Constant folding* replaces expressions whose value is already known at compile time by this value. This optimization supports the development of generic programs, often called *program families*. These are parametrized in a number of variables and thus can be instantiated to many different variants by supplying different combinations of parameter values. This is good and effective development practice, for instance, in the embedded-systems industry. One generic power-train control program may be instantiated to many different versions for different car engines. Constant folding eliminates the loss in efficiency that could result from such a programming style.

Checks for run-time errors can be eliminated if it is clear that they would always fail, that is, if these errors would provably never happen. A good example is the check for index out of bounds. It checks the indices of arrays against their lower und upper bounds. These checks can be avoided if the indices provably always lie within these bounds.

Another idea to improve the efficiency of programs is to move computations from more frequently executed program parts into less frequently executed parts. An example of this kind of optimization is to move loop-invariant computations out of loops.

Some operations are more costly in execution time than others. For example, multiplication is more expensive than addition. Multiplication can be defined, and this means also replaced by, repeated addition. An optimization, called *reduction in operator strength* would, under certain conditions, replace a multiplication occurring in a loop by an addition.

Finally, *procedure inlining*, i.e., replacing a procedure call by an appropriately instantiated body of the procedure, eliminates the procedure-call overhead and often opens the way to new optimizations.

The following example shows how big the impact of optimizations on the quality of generated code can be:

Example 1.1.1 Consider a program that should sort an array a written in an imperative programming language. This program would use the following function swap:

```
void swap ( int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t ← a[j];
        a[j] ← a[i];
        a[i] ← t;
    }
}
```

The inefficiencies of this implementation are apparent. The addresses of $a[i]$ and $a[j]$ are computed three times. This leads to 6 address computations altogether. However, two should be sufficient. In addition, the values of $a[i]$ and $a[j]$ are loaded twice, resulting in four memory accesses where two should be sufficient.

These inefficiencies can be removed by an implementation as suggested by the array concept of the C programming language. The idea is to access array elements through pointers. Another idea is to store addresses that are used multiple times.

```
void swap (int *p, int *q) {
    int t, ai, aj;
    ai ← *p; aj ← *q;
    if (ai > aj) {
        t ← aj;
        *q ← ai;
        *p ← t;
    }
}
```

Looking more closely at this new code reveals that the temporary variable t can be eliminated.

This second version is apparently more efficient. However, the original version was much more intuitive. High-level programming languages are designed to allow intuitive formulations of algorithms. It is then the duty of the compiler to generate efficient target programs. □

Optimizing program transformations ought to preserve the semantics of the program, which is defined through the semantics of the programming language in which the program is written.

Example 1.1.2 Consider the transformation:

$$y \leftarrow \mathsf{f}() + \mathsf{f}(); \qquad \Longrightarrow \qquad y \leftarrow 2 * \mathsf{f}();$$

The idea behind the "optimization" is to save the evaluation of the second call of the function f. However, the program resulting from this transformation is only equivalent to the original program if the second call to f is guaranteed to produce the same result and if the call does not produce a side effect. This last condition is not immediately clear for functions written in an imperative language. □

This means that so-called program optimizations are not correct if they change the semantics of the program. Therefore, most optimizing transformations have an associated *applicability condition*. This is a sufficient condition for the preservation of the semantics of programs. Checking the satisfaction of these applicability conditions is the duty of static program analysis. Such analyses need to be automatic, that is, run without user intervention, as they will have to be performed by the compiler.

A careful treatment of the issue of semantics preservation needs several proofs. First, a proof is needed that the applicability condition is, in fact, a sufficient condition for semantics preservation. A second proof is needed that the analysis that is to determine the applicability is correct, that is, will never give wrong answers to the question posed by the applicability condition. Both proofs use an *operational semantics* as their basis.

Several of the optimizations mentioned above are effective across several classes of programming languages. However, each programming language and also each class of programming languages require specific optimizations, designed to improve the efficiency of particular language constructs. One such example is the compile-time removal of dynamic method invocations in object-oriented programs. A static method call, which replaces a dynamic call, can be inlined and thus opens the door for further optimizations. This is very effective since methods in object-oriented programs are often rather small. In FORTRAN, on the other hand, inlining doesn't play a comparably large role. For FORTRAN, the parallelization or vectorization of nested loops has greater impact.

The programming language, in particular its semantics, may have a strong influence on the efficiency and the effectiveness of program analyses. The programming language may enforce restrictions whose validation would otherwise require an enormous effort. A major problem in the analysis of imperative programs is the determination of dependencies between the statements in programs. Such dependencies restrict the compiler's possibility to reorder statements to better exploit the resources of the target machine. The unrestricted use of pointers, as in the C programming language, makes this analysis of dependencies very difficult due to the severe alias-problem created through pointers. The more restricted use of pointers in JAVA eases the corresponding analysis.

Example 1.1.3 Let us look at the programming language JAVA. Inherently inefficient language constructs are the mandatory checks for indices out of array bounds, dynamic method invocation, and storage management for objects. The absence of pointer arithmetic and of pointers into the stack increases the analyzability of JAVA programs. On the other hand, dynamic loading of classes may ruin the precision of JAVA analyses due to the lack of information about their semantics and their implementation. Further tough challenges for an automatic static analysis are offered by language constructs such as exceptions, concurrency, and reflection, which may be very useful for the JAVA programmer.

We have stressed in the preface that sound static program analysis has become a verification technology. It is therefore interesting to draw the connection to the problem of proving the correctness of JAVA programs. Any correctness proof needs a formally specified semantics of the programming language. Quite some effort went into the development of such a semantics for JAVA. Still, JAVA programs with a formal correctness proof are rather rare, not because of a principal impossibility, but due to the sheer size of the necessary effort. JAVA just has too many language constructs, each with its non-trivial semantics. □

For this reason, we will not use JAVA as our example language. Instead we use a small subset of an imperative programming language. This subset is, on the one hand, simple enough to limit the descriptional effort, and is, on the other hand, realistic enough to include essential problems of actual compilers. This programming-language fragment can be seen as an intermediate language into which source programs are translated. The *int* variables of the program can be seen as *virtual registers*. The compiler backend will, during register allocation, assign physical registers to them as far as such physical registers are available. Such variables can also be used to store addresses for indirect memory accesses. Arithemtic expressions represent computations of, in our fragment, *int* values. Finally, the fragment contains an abitrarily large array M, into which *int* values can be stored and from which they can be retrieved. This array can be imagined as the whole (virtual) memory allocated to a program by the operating system.

The separation between variables and memory may, at first glance, look somewhat artificial. It is motivated by the wish to avoid the *alias problem*. Both a variable x and a memory-access expression $M[\cdot]$ denote containers for values. The identity of a memory cell denoted by $M[e]$ is not directly visible because it depends on the value of the expression e. In general, it is even undecidable whether $M[e_1]$ and $M[e_2]$ denote the same memory cell. This is different for variables: A variable name x is the only name by which the container associated with x can be accessed. This is important for many program analyses: If the analysis is unable to derive the identity of the memory cell denoted by $M[e]$ in a write access then no assumptions can be made about the contents of the rest of memory. The analysis looses much precision. The derivation of assumptions about the contents of containers associated with variables is easier since no indirect access to their containers is possible.

Our small language fragment has the following constructs:

- variables: x
- arithmetic expressions: e
- assignments: $x \leftarrow e$
- reading access to memory: $x \leftarrow M[e]$
- writing access to memory: $M[e_1] \leftarrow e_2$
- conditional statement: **if** (e) s_1 **else** s_2
- unconditional jump: **goto** L

Note that we have not included explicit loop constructs. These can be substituted by conditional and unconditional jumps to labeled program points. Also missing so far are functions and procedures. This chapter is therefore restricted to the analysis and optimization of single functions.

Example 1.1.4 Let us again consider the function swap() of Example 1.1.1. How would a compiler translate the body of this function into our language fragment? The array a would be allocated into some section of the memory M. Accesses to array components need to be translated into explicit address calculations. The result of a schematic, nonoptimized translation could be:

$$
\begin{array}{lllll}
0: & A_1 \leftarrow A_0 + 1 * i; & & // & A_0 = \&a[0] \\
1: & R_1 \leftarrow M[A_1]; & & // & R_1 = a[i] \\
2: & A_2 \leftarrow A_0 + 1 * j; & & & \\
3: & R_2 \leftarrow M[A_2]; & & // & R_2 = a[j] \\
4: & \textbf{if } (R_1 > R_2) \; \{ & & & \\
5: & \quad A_3 & \leftarrow A_0 + 1 * j; & & \\
6: & \quad t & \leftarrow M[A_3]; & & \\
7: & \quad A_4 & \leftarrow A_0 + 1 * j; & & \\
8: & \quad A_5 & \leftarrow A_0 + 1 * i; & & \\
9: & \quad R_3 & \leftarrow M[A_5]; & & \\
10: & \quad M[A_4] & \leftarrow R_3; & & \\
11: & \quad A_6 & \leftarrow A_0 + 1 * i; & & \\
12: & \quad M[A_6] & \leftarrow t; & & \\
13: & \} & & // &
\end{array}
$$

We assume that variable A_0 holds the start address of the array a. Note that this code makes explicit the inherent inefficiencies discussed in Example 1.1.1. Which optimizations are applicable?

Optimization 1: $1 * R \implies R$

The scaling factor generated by an automatic (and schematic) translation of array indexing can be dispensed with if this factor is 1 as is the case in the example.

Optimization 2: Reuse of values calculated for (sub)expressions

A closer look at the example shows that the variables A_1, A_5, and A_6 have the same values as is the case for the variables A_2, A_3, and A_4:

$$A_1 = A_5 = A_6 \qquad A_2 = A_3 = A_4$$

In addition, the memory accesses $M[A_1]$ and $M[A_5]$ as well as the accesses $M[A_2]$ and $M[A_3]$ will deliver the same values:

$$M[A_1] = M[A_5] \qquad M[A_2] = M[A_3]$$

Therefore, the variables R_1 and R_3, as well as the variables R_2 and t also contain the same values:

$$R_1 = R_3 \qquad R_2 = t$$

If a variable x already contains the value of an expression e whose value is needed then x's value can be used instead of reevaluating the expression e. The program can be greatly simplified by exploiting all this information:

$$
\begin{array}{l}
A_1 \leftarrow A_0 + i; \\
R_1 \leftarrow M[A_1]; \\
A_2 \leftarrow A_0 + j; \\
R_2 \leftarrow M[A_2]; \\
\textbf{if } (R_1 > R_2)\ \{ \\
\qquad M[A_2] \leftarrow R_1; \\
\qquad M[A_1] \leftarrow R_2; \\
\}
\end{array}
$$

One can observe that the temporary variable t as well as the variables A_3, A_4, A_5, and R_3 are now superfluous and can be eliminated from the program.

The following table lists the achieved savings:

	Before	**After**
$+$	6	2
$*$	6	0
load	4	2
store	2	2
$>$	1	1
$\leftarrow$	6	2

□

The optimizations applied to the function swap "by hand" should, of course, be done in an automated way. The following sections will introduce the necessary analyses and transformations.

1.2 Avoiding Superfluous Computations

This chapter presents a number of techniques to save computations that the program would otherwise needlessly perform. We start with an optimization that avoids *redundant computations*, that is, multiple evaluations of the same expression guaranteed to produce the same result. This first example is also used to exemplify fundamentals of the approach. In particular, an operational semantics of our language fragment is introduced in a concise way, and the necessary lattice-theoretic foundations are discussed.

A frequently used trick to speed up algorithms is to trade time against space, more precisely, invest some additional space in order to speed up the program's execution. The additional space is used to save some computed values. These values are then later retrieved instead of recomputed. This technique is often called *memoization*.

Let us consider the profitability of such a transformation replacing a recomputation by an access to a stored value. Additional space is needed for the storage of this value. The recomputation does not disappear completely, but is replaced by an access to the stored value. This access is cheap if the value is kept in a register, but it can also be expensive if the value has to be kept in memory. In the latter case, recomputing the value may, in fact, be cheaper. To keep things simple, we will ignore such considerations of the costs and benefits, which are highly architecture-dependent. We will assume that accessing a stored value is always cheaper than recomputing it.

The computations we consider here are evaluations of expressions. The first problem is to recognize potential recomputations.

Example 1.2.1 Consider the following program fragment:

$$
\begin{array}{lll}
 & z & \leftarrow 1; \\
 & y & \leftarrow M[5]; \\
A: & x_1 & \leftarrow \boxed{y+z}; \\
 & & \quad \ldots \\
B: & x_2 & \leftarrow \boxed{y+z};
\end{array}
$$

It seems like at program point B, the expression $y+z$ will be evaluated a second time yielding the same value. This is true under the following conditions: The occurrence of $y+z$ at program point B is always evaluated *after* the one at program point A, and the values of the variables y and z have the same values before B that they had before A. □

Our conclusion from the example above is that for a systematic treatment of this optimization we need to be able to answer the following questions:

- Will one evaluation of an expression always be executed before another one?
- Does a variable always have the same value at a given program point that it had at another program point?

To answer these types of questions, we need several things: an *operational semantics*, which defines what happens when a program is executed, and a method that identifies redundant computations in programs. Note that we are not so ambitious as to attempt to identify *all* redundant computations. This problem is undecidable. In practice, the method to be developed should at least find some redundant computations and should never classify as redundant a computation that, in fact, is not redundant.

1.3 Digression: An Operational Semantics

Small-step operational semantics have been found to be quite adequate for correctness proofs of program analyses and transformations. Such a semantics formalizes what a step in a computation is. A *computation* is then a sequence of such steps.

We start by choosing a suitable program representation, *control-flow graphs*. The vertices of these graphs correspond to program points; we will therefore refer to these vertices as program points. Program execution traverses these vertices. The edges of the graph correspond to steps of the computation. They are labeled with the corresponding program actions, that is, with conditions, assignments, loads and stores from and to memory, or with the empty statement, ";". Program point *start* represents the entry point of the program, and *stop* the exit point.
Possible edge labels are:

test:	$\mathsf{NonZero}(e)$ or $\mathsf{Zero}(e)$
assignment:	$x \leftarrow e$
load:	$x \leftarrow M[e]$
store:	$M[e_1] \leftarrow e_2$
empty statment:	;

A section of the control-flow graph for the body of the function swap is shown in Fig. 1.1. Sometimes, we omit an edge label ;. A conditional statement with condition e in a program has two corresponding edges in the control-flow graph. The one labeled with $\mathsf{NonZero}(e)$ is taken if the condition e is satisfied. This is the case when e evaluates to some value not equal to 0. The edge labeled with Zero is taken if the condition is not satisfied, i.e., when e evaluates to 0.

Computations are performed when paths of the control-flow graph are traversed. They transform the *program state*. Program states can be represented as pairs

$$s = (\rho, \mu)$$

The function ρ maps each program variable to its actual value, and the function μ maps each memory address to the actual contents of the corresponding memory cell. For simplicity, we assume that the values of variables and memory cells are integers. The types of the functions ρ and μ, thus, are:

$\rho : \mathit{Vars} \to \mathbf{int}$	value of variables
$\mu : \mathbb{N} \to \mathbf{int}$	memory contents

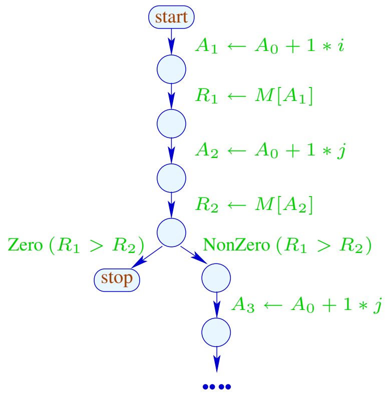


Fig. 1.1. A section of the control-flow graph for swap()

An edge $k = (u, lab, v)$ with source vertex u, target vertex v and label lab defines a transformation $\llbracket k \rrbracket$ of the state before the execution of the action labeling the edge to a state after the execution of the action. We call this transformation the *effect* of the edge. The edge effect need not be a total function. It could also be a *partial* function. Program execution in a state s will not execute the action associated with an edge if the edge effect is undefined for s. There may be two reasons for an edge being undefined: The edge may be labeled with a condition that is not satsified in all states, or the action labeling the edge may cause a memory access outside of a legal range.

The edge effect $\llbracket k \rrbracket$ of the edge $k = (u, lab, v)$ only depends on its label lab:

$$\llbracket k \rrbracket = \llbracket lab \rrbracket$$

The edge effects $\llbracket lab \rrbracket$ are defined as follows:

$$\llbracket ; \rrbracket \, (\rho, \mu) = (\rho, \mu)$$

$$\begin{array}{rcll} \llbracket \mathsf{NonZero}(e) \rrbracket \, (\rho, \mu) &=& (\rho, \mu) & \text{if } \llbracket e \rrbracket \, \rho \neq 0 \\ \llbracket \mathsf{Zero}(e) \rrbracket \, (\rho, \mu) &=& (\rho, \mu) & \text{if } \llbracket e \rrbracket \, \rho = 0 \end{array}$$

$$\llbracket x \leftarrow e \rrbracket \, (\rho, \mu) = (\boxed{\rho \oplus \{x \mapsto \llbracket e \rrbracket \, \rho\}}, \mu)$$

$$\llbracket x \leftarrow M[e] \rrbracket \, (\rho, \mu) = (\boxed{\rho \oplus \{x \mapsto \mu(\llbracket e \rrbracket \rho)\}}, \mu)$$

$$\llbracket M[e_1] \leftarrow e_2 \rrbracket \, (\rho, \mu) = (\rho, \boxed{\mu \oplus \{\llbracket e_1 \rrbracket \rho \mapsto \llbracket e_2 \rrbracket \rho\}})$$

An empty statement does not change the state. Conditions $\mathsf{NonZero}(e)$ and $\mathsf{Zero}(e)$, represent partial identities; the associated edge effects are only defined if the conditions are satisfied, that is if the expression e evaluated to a value not equal to or equal to 0, resp. They do, however, not change the state. Expressions e are evaluated by an auxiliary function $[\![e]\!]$, which takes a variable binding ρ of the program's variables and calculates e's value in the valuation ρ. As usual, this function is defined by induction over the structure of expressions. This is now shown for some examples:

$$\begin{array}{ll} [\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = & 6 \\ [\![\neg(x=4)]\!]\,\{x \mapsto 5\} & = \neg 0 = 1 \end{array}$$

The operator $\neg$ denotes *logical negation*.

An assignment $x \leftarrow e$ modifies the ρ-component of the state. The resulting ρ holds the value $[\![e]\!]\,\rho$ for variable x, that is, the value obtained by evaluating e in the old variable binding ρ. The memory M remains unchanged by this assignment. The formal definition of the change to ρ uses an operator $\oplus$. This operator modifies a given function such that it maps a given argument to a given new value:

$$\rho \oplus \{x \mapsto d\}(y) = \begin{cases} d & \text{if } y \equiv x \\ \rho(y) & \text{otherwise} \end{cases}$$

A load action, $x \leftarrow M[e]$, is similar to an assignment with the difference that the new value of variable x is determined by first calculating a memory address and then loading the value stored at this address from memory.

The store operation, $M[e_1] \leftarrow e_2$, has the most complex semantics. Values of variables don't change. The following sequence of steps is performed: The values of the expressions e_1, e_2 are computed. e_1's value is the address of a memory cell at which the value of e_2 is stored.

We assume for both load and store operations that the address expressions deliver legal addresses, i.e., values > 0.

Example 1.3.1 An assignment $x \leftarrow x+1$ in a variable binding $\{x \mapsto 5\}$ results in:

$$[\![x \leftarrow x+1]\!]\,(\{x \mapsto 5\}, \mu) = (\rho, \mu)$$

where:

$$\begin{array}{lll} \rho & = & \{x \mapsto 5\} \oplus \{x \mapsto [\![x+1]\!]\,\{x \mapsto 5\}\} \\ & = & \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\ & = & \{x \mapsto 6\} \end{array}$$

□

We have now established what happens when edges of the control-flow graph are traversed. A *computation* is (the traversal of) a path in the control-flow graph leading from a starting point u to an endpoint v. Such a path is a sequence $\pi = k_1 \ldots k_n$ of edges $k_i = (u_i, lab_i, u_{i+1})$ of the control-flow graph ($i = 1, \ldots, n-1$), where $u_1 = u$ and $u_n = v$. The state transformation $[\![\pi]\!]$ corresponding to π is obtained as the *composition* of the edge effects of the edges of π:

$$[\![\pi]\!] = [\![k_n]\!] \circ \ldots \circ [\![k_1]\!]$$

Note that, again, the function $[\![\pi]\!]$ need not be defined for all states. A computation along π starting in state s is only possible if $[\![\pi]\!]$ is defined for s.

1.4 Elimination of Redundant Computations

Let us return to our starting point, the attempt to find an analysis that determines for each program point whether an expression has to be newly evaluated or whether it has an already computed value. The method to do this is to identify *expressions available in variables*. An expression e is available in variable x at some program point if it has been evaluated before, the resulting value has been assigned to x, and neither x nor any of the variables in e have been modified in between. Consider an assignment $x \leftarrow e$ such that $x \notin \mathsf{Vars}(e)$, that is, x does not occur in e. Let $\pi = k_1 \ldots k_n$ be a path from the entry point of the program to a program point v. The expression e is *available* in x at v if the two following conditions hold:

- The path π contains an edge k_i, labeled with an assignment $x \leftarrow e$.
- No edge $k_{i+1}, \ldots, k_n$ is labeled with an assignment to one of the variables in $\mathsf{Vars}(e) \cup \{x\}$.

For simplicity, we say in this case that the *assignment* $x \leftarrow e$ is *available* at v Otherwise, we call e or $x \leftarrow e$, resp., *not available in* x at v. We assume that no assignment is available at the entry point of the program. So, none are available at the end of an empty path $\pi = \epsilon$.

Regard an edge $k = (u, lab, v)$ and assume we knew the set A of assignments available at u, i.e., at the source of k. The action labeling this edge determines which assignments are added to or removed from the availability set A. We look for a function $[\![k]\!]^\sharp$ such that the set of assignments available at v, i.e., at the target of k, is obtained by applying $[\![k]\!]^\sharp$ to A. This function $[\![k]\!]^\sharp$ should only depend on the label of k. It is called the *abstract edge effect* in contrast to the concrete edge effect of the operational semantics. We now define the abstract edge effects $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ for different types of actions.

Let *Ass* be the set of all assignments of the form $x \leftarrow e$ in the program and with the constraint that $x \notin \mathsf{Vars}(e)$.[1] Let us assume that $A \subseteq Ass$ were available at the source u of the edge $k = (u, lab, v)$. The set of assignments available at the target of k is determined according to:

[1] An assignment violating this constraint is not considered to be available at the subsequent program point.

$$
\begin{aligned}
[\![;]\!]^\sharp\, A &= A \\
[\![\mathsf{NonZero}(e)]\!]^\sharp\, A &= [\![\mathsf{Zero}(e)]\!]^\sharp\, A = A \\
[\![x \leftarrow e]\!]^\sharp\, A &= \begin{cases} (A \backslash \mathsf{Occ}(x)) \cup \{x \leftarrow e\} & \text{if } x \not\in \mathsf{Vars}(e) \\ A \backslash \mathsf{Occ}(x) & \text{otherwise} \end{cases} \\
[\![x \leftarrow M[e]]\!]^\sharp\, A &= A \backslash \mathsf{Occ}(x) \\
[\![M[e_1] \leftarrow e_2]\!]^\sharp\, A &= A
\end{aligned}
$$

where $\mathsf{Occ}(x)$ denotes the set of all assignments in which x occurs either on the left or in the expression on the right side. An empty statement and a condition do not change the set of available assignments. Executing an assignment to x means evaluating the expression on the right side and assigning the resulting value to x. Therefore, all assignments that contain an occurrence of x have to be removed from the available-set. Following this, the actual assignment has to be added to the available-set provided x does not occur in the right side. The abstract edge effect for loads from memory looks similar. Storing into memory does not change the value of any variable, hence, A remains unchanged.

The abstract effects, which were just defined for each type of label, are composed to an abstract effect $[\![\pi]\!]^\sharp$ for a path $\pi = k_1 \dots k_n$ in the following way:

$$[\![\pi]\!]^\sharp = [\![k_n]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

The set of assignments available at the end of a path π from the entry point of the program to program point v is therefore obtained as:

$$[\![\pi]\!]^\sharp \emptyset = [\![k_n]\!]^\sharp (\ldots ([\![k_1]\!]^\sharp\, \emptyset) \ldots)$$

Applying such a function associated with a path π can be used to determine which assignments are available along the path. However, a program will typically have several paths leading to a program point v. Which of these paths will actually be taken at program execution may depend on program input and is therefore unknown at analysis time. We define an assignment $x \leftarrow e$ to be *definitely available* at a program point v if it is available along *all* paths leading from the entry node of the program to v. Otherwise, $x \leftarrow e$ is possibly not available at v. Thus, the set of assignments definitely available at a program point v is:

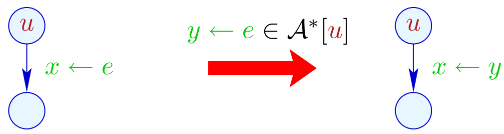
$$\mathcal{A}^*[v] \quad = \quad \bigcap \{ [\![\pi]\!]^\sharp \emptyset \mid \pi : start \rightarrow^* v \}$$

where $start \rightarrow^* v$ denotes the set of all paths from the entry point $start$ of the program to the program point v.

We temporarily postpone the treatment of how to compute the sets $\mathcal{A}^*[v]$. We will later call this the merge-over-all-paths (MOP) solution of the analysis problem. The general approach to compute it will only succeed under some conditions for the abstract edge effects. We now discuss how the analysis information can be used for an optimizing transformation of the program.

Transformation RE:

An assignment $x \leftarrow e$ is replaced by an assignment $x \leftarrow y$, if $y \leftarrow e$ is definitely available at program point u just before this assignment, i.e., $y \leftarrow e$ is contained in the set $\mathcal{A}^*[u]$. This is formally described by the following graph rewrite rule:



Analog rules describe the replacement of expressions by variable accesses in conditions and in loads from and in stores into memory.

The transformation RE is called *redundancy elimination*. The transformation appears quite simple. It may, however, require quite some effort to compute the program properties necessary to ascertain the applicability of the transformation.

Example 1.4.1 Regard the following program fragment:

$$\begin{aligned} & x \leftarrow y + 3; \\ & x \leftarrow 7; \\ & z \leftarrow y + 3; \end{aligned}$$

The assignment $x \leftarrow y + 3$ is not available before, but it is available after the first statement. The second assignment overwrites the value of x. So, the third assignment can not be simplified using rule RE. □

Example 1.4.2 Look at the implementation of the C statement $a[7]$--; in our language fragment. Assume that the start address of the array a is contained in variable A. Figure 1.2 shows the original control-flow graph of the program fragment together with the application of transformation rule RE. The right side, $A + 7$, of the assignment $A_2 \leftarrow A + 7$ can be replaced by the variable A_1 since the assignment $A_1 \leftarrow A + 7$ is definitely available just before the assignment $A_2 \leftarrow A + 7$. □

The compiler could provide a dedicated variable for each expression of interest to increase the chances to apply this transformation. This would eliminate the problem that an expression could have been evaluated, but its value was no longer available since the variable that contained the value had meanwhile received a different value, see Example 1.4.1. To develop a corresponding transformation is the task of Exercise 5.

To decide when the application of the transformation RE is profitable is very difficult. Storing the value of a subexpression costs storage resources. Access to the value will be fast if the compiler succeeds to keep the value in a register. However, registers are a scarce resource, and spending one of them for temporary storage may cause more costs somewhere else. Storing the value in memory will result in long access times

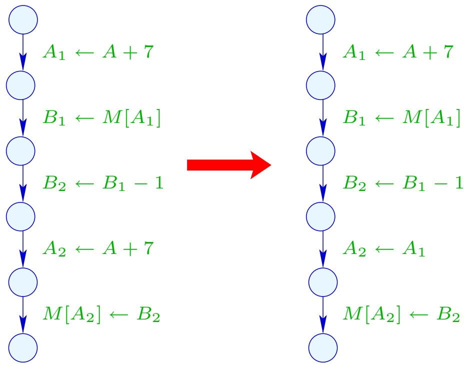


Fig. 1.2. Transformation RE applied to the code for $a[7]--;$.

such that it may be cheaper to reevaluate the expression if access to its operands is cheap.
Let us now turn to the correctness proof of the described transformation. It can be split into two parts:

1. The proof of correctness of the abstract edge effects $[\![k]\!]^\sharp$ with respect to the definition of availability
2. The proof of correctness of the replacement of definitely available expressions by accesses to variables

We only treat the second part. Note that we initially talked about the availability of expressions in semantic terms, namely the evaluation of expressions and the assignment of their values to variables. We then secretly switched to syntactic terms to formulate the analysis, namely actions labeling edges of the control-flow graph, paths in this graph, and occurrences of variables on the left and right side of assignments or in conditions. The proof has to connect syntax with semantics.

Let π be a path leading from the entry point of the program to a program point u, and let $s = (\rho, \mu)$ be the state after the execution of the path π. Let $y \leftarrow e$ be an assignment such that $y \notin \mathsf{Vars}(e)$ holds and that $y \leftarrow e$ is available at u. It is easily shown by induction over the length of executed paths π that the value of y in state s is equal to the value of the expression e when evaluated in the valuation ρ, i.e., $\rho(y) = [\![e]\!]\,\rho$.

Assume that program point u has an outgoing edge k labeled with assignment $x \leftarrow e$, and that $y \leftarrow e$ is contained in $\mathcal{A}^*[u]$, i.e., definitely available. $y \leftarrow e$ is in particular available at the end of path π. Therefore, $\rho(y) = [\![e]\!]\,\rho$ holds. Under this condition, the assignment $x \leftarrow e$ can be replaced by $x \leftarrow y$.

The proof when thoroughly done convinces one of the correctness of the analysis and the associated transformation. What about the *precision* of the analysis? Does a compiler realizing this analysis miss some opportunity to remove redundant computations, and if so, why? There are, in fact, several reasons why this can happen. The first reason is caused by *infeasible* paths. We saw seen in Sect. 1.3 that a path may be not executable in all states or even in not any states at all. In the latter case, such a path is called *infeasible*. The composition of the concrete edge effects of such a path is not defined anywhere. The abstract edge effects of our analysis, however, are total functions. They don't know about infeasibility. Such a path would be considered in forming the intersection in the definition of definite availability and may "pollute" the information if this path does not contain an assignment available on all other paths.

A second reason is the following: Assume that the assigment $x \leftarrow y + z$ is available at program point u, and that there exists an edge $k = (u, y \leftarrow e, v)$ leaving u. Assume further that the value of e is always the one that y has at u. In this case, the transformation replacing $y + z$ by x would still be correct although $x \leftarrow y + z$ would no longer be recognized as available at v.

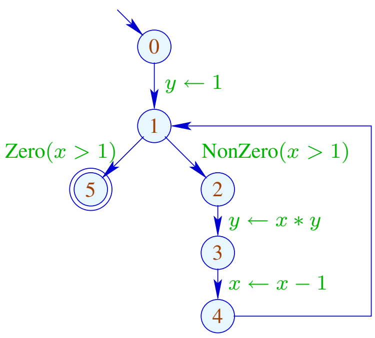
An important question remains: How are the sets $\mathcal{A}^*[u]$ computed?

The main idea is to derive from the program a *system of inequalities* that characterizes these values. It consists of all the constraints that the values must satisfy.

$$\begin{array}{lll} \mathcal{A}[start] & \subseteq \emptyset & \\ \mathcal{A}[v] & \subseteq [\![k]\!]^\sharp\,(\mathcal{A}[u]) & \text{for an edge} \quad k = (u, lab, v) \end{array}$$

The first inequality expresses the assumption that no assignments are available at the entry of the program. Further, each edge k leading from a node u to a node v generates an inequality of the second kind. $[\![k]\!]^\sharp\,(\mathcal{A}[u])$ are the assignments that are propagated as available along the edge $k = (u, lab, v)$, either since they were available at u and "survived" $[\![k]\!]$ or since they were made available by $[\![k]\!]$. This set is *at most* available at v since other edges may target v along which these assignments might not be available.

Example 1.4.3 Let us consider the program implementing the factorial function as in Figure 1.3. We see that the system of inequalities can be produced from the



$$\begin{array}{l} \mathcal{A}[0] \subseteq \emptyset \\ \mathcal{A}[1] \subseteq (\mathcal{A}[0] \backslash \mathsf{Occ}(y)) \cup \{y \leftarrow 1\} \\ \mathcal{A}[1] \subseteq \mathcal{A}[4] \\ \mathcal{A}[2] \subseteq \mathcal{A}[1] \\ \mathcal{A}[3] \subseteq \mathcal{A}[2] \backslash \mathsf{Occ}(y) \\ \mathcal{A}[4] \subseteq \mathcal{A}[3] \backslash \mathsf{Occ}(x) \\ \mathcal{A}[5] \subseteq \mathcal{A}[1] \end{array}$$

Fig. 1.3. The system of inequalities for the factorial function

control-flow graph and the abstract edge transformers in a straightforward way. The system of inequalities of the example can be largely simplified. The only assignment whose left-side variable does not also occur on the right side is $y \leftarrow 1$. The complete lattice for the analysis of available assignments therefore consists of only two elements, $\emptyset$ and $\{y \leftarrow 1\}$. Correspondingly, $\mathsf{Occ}(y) = \{y \leftarrow 1\}$ and $\mathsf{Occ}(x) = \emptyset$ hold.

Figure 1.4 shows a trivial solution of this system of inequalities. In this case, this

$$\mathcal{A}[0] = \mathcal{A}[1] = \mathcal{A}[2] = \mathcal{A}[3] = \mathcal{A}[4] = \mathcal{A}[5] = \emptyset$$

Fig. 1.4. A trivial solution of the system of inequalities of Example 1.4.3

is the only solution. In general, there could be several solutions. In the available-assignment analysis, we are interested in *largest* sets. The larger the sets, the more assignments have been shown to be available, and the more optimizations can be performed. In consequence, we consider an analysis more precise that identifies more assignments as available.

So, in this case, the largest solution is the best solution. The question is, does a best solution always exist? If yes, can it be efficiently computed? We generalize the problem a bit to be able to systematically answer the question as to the existence of best solutions of systems of inequalities and as to their efficient computation. This generalization will allow us to apply the treatment to all our program-analysis problems.

The first observation is that the set of possible values for the unknowns $\mathcal{A}[v]$ forms a *partial order* with respect to the subset relation $\subseteq$. The same holds for the superset relation $\supseteq$. These partial orders have the additional property that each subset of X has a *least upper bound* and a *greatest lower bound*, namely the union and the intersection, respectively, of the sets in X. Such a partial order is called a *complete lattice*.

A further observation is that the abstract edge transformers $[\![k]\!]^\sharp$ are *monotonic* functions, that is, they maintain the ordering relation between values:

$$[\![k]\!]^\sharp(B_1) \supseteq [\![k]\!]^\sharp(B_2) \quad \text{if} \quad B_1 \supseteq B_2$$

1.5 Digression: Complete Lattices

This section presents fundamental notions and theorems about complete lattices, solutions of systems of inequalities, and the essentials of methods to compute least solutions. The reader should not be confused about best solutions being least solutions, although in the available-assignments analysis the largest solution was claimed to be the best solution. The following treatment is in terms of partial orders, $\sqsubseteq$, where less is, by convention, always more precise. We take the liberty to set $\sqsubseteq = \supseteq$ as in the case of available assignments. We start with definitions of partial orders and complete lattices.

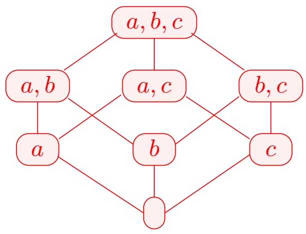
A set $\mathbb{D}$ together with a relation $\sqsubseteq$ on $\mathbb{D} \times \mathbb{D}$ is called a *partial order* if for all $a, b, c \in \mathbb{D}$ it holds that:

$$\begin{array}{ll} a \sqsubseteq a & \textit{reflexivity} \\ a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b & \textit{antisymmetry} \\ a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c & \textit{transitivity} \end{array}$$

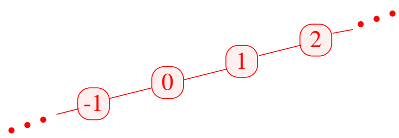
The sets we consider in this book consist of information at program points about potential or definite program behaviors. In our running example, such a piece of information at a program point is a set of available assignments. The ordering relation indicates *precision*. By convention, "less" means "more precise". More precise in the context of program optimizations should mean "enabling more optimizations". For the available-assigments analysis, more available assignments means potentially more enabled optimizations. So, the ordering relation $\sqsubseteq$ is the superset relation $\supseteq$.

We give some examples of partial orders, representing lattices graphically as directed graphs. Vertices are the lattice elements. Edges are directed upwards and represent the $\sqsubseteq$ relation. Vertices not connected by a sequence of edges are incomparable by $\sqsubseteq$.

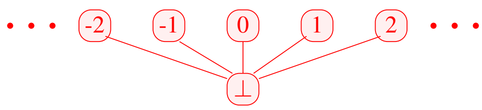
1. $\mathbb{D} = 2^{\{a,b,c\}}$, the set of all subsets of the finite set $\{a, b, c\}$ together with the relation $\subseteq$:



2. The set of all integer numbers $\mathbb{Z}$ together with the relation $\leq$:



3. The set of all integer numbers $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$, extended by an additional element $\perp$ together with the order:



An element $d \in \mathbb{D}$ is called an *upper bound* for a subset $X \subseteq \mathbb{D}$ if

$$x \sqsubseteq d \qquad \text{for all } x \in X$$

An element d is called a *least upper bound* if

1. d is an upper bound, and
2. $d \sqsubseteq y$ holds for each upper bound y of X.

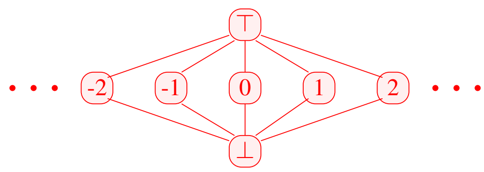
Not every subset of a partially ordered set has an upper bound, let alone a least upper bound. The set $\{0, 2, 4\}$ has the upper bounds $4, 5, \ldots$ in the partially ordered set $\mathbb{Z}$ of integer numbers, with the natural order $\leq$, while the set $\{0, 2, 4, \ldots\}$ of all even numbers has no upper bound.

A partial order $\mathbb{D}$ is a *complete lattice* if each subset $X \subseteq \mathbb{D}$ possesses a least upper bound. This least upper bound is represented as $\bigsqcup X$.

Forming the least upper bound of a set of elements is an important operation in program analysis. Let us consider the situation that several edges of the control-flow graph have the same target node u. The abstract edge effects associated with these edges propagate different information towards u. How should we combine this information? Well, the least upper bound operator combines the incoming information in a sound and the most precise way.

Each element is an upper bound of the empty set. The empty set, on the other hand, also possesses a least upper bound, $\perp$, which is less than or equal to any other element of the complete lattice. This least element, $\perp$, is called the *bottom* element of the lattice. The set of all elements of a complete lattice also possesses an upper bound. Each complete lattice therefore also has a greatest element, $\top$, called the *Top* element. Let us consider the partial orders of our examples. We have:

1. The set $\mathbb{D} = 2^{\{a,b,c\}}$ of all subsets of the basic set $\{a, b, c\}$ and, in general, of each basic set together with the subset relation is a complete lattice.
2. The set $\mathbb{Z}$ of the integer numbers with the partial order $\leq$ is not a complete partial order.
3. The addition of a least element, $\perp$, *and* a greatest element, $\top$, is necessary to obtain a complete lattice, namely the so-called *flat lattice* $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$:



In analogy to upper and least upper bounds, one can define lower and greatest lower bounds for subsets of a partially ordered set. For a warm-up, we prove the following theorem:

Theorem 1.5.1 *Each subset X of a complete lattice $\mathbb{D}$ has a greatest lower bound $\bigsqcap X$.*

Proof. Let $U = \{u \in \mathbb{D} \mid \forall x \in X : \; u \sqsubseteq x\}$ the set of all lower bounds of the set X. The set U has a least upper bound $g := \bigsqcup U$ since $\mathbb{D}$ is a complete lattice. We claim that g is the desired greatest lower bound of X.

To prove this claim, we show that g is also a lower bound of the set X. For this, we take an arbitrary element $x \in X$. It holds $u \sqsubseteq x$ for each $u \in U$, since each $u \in U$ is even a lower bound for the whole set X. Therefore, x is an upper bound of the set U, and in particular greater than or equal to the least upper bound of U, i.e., $g \sqsubseteq x$. The claim follows since x was an arbitrary element.

It remains to show that g is also the greatest lower bound of X. This is easily shown. g is an upper bound of U, therefore in particular greater than or equal to each element in U, i.e., $u \sqsubseteq g$ for all $u \in U$. $\square$

Figure 1.5 shows a complete lattice, a subset, and its greatest lower and least upper bounds. That each of its subsets has a least upper bound makes a complete lattice

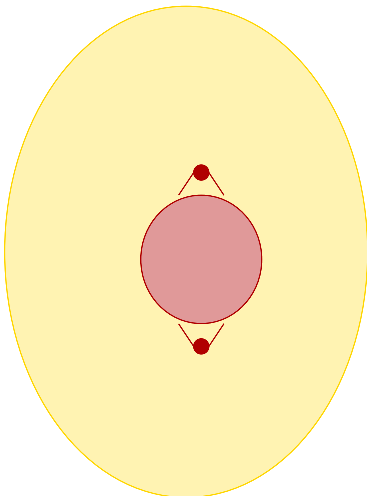


Fig. 1.5. The least upper bound and the greatest lower bound for a subset X

out of a partially ordered set. Theorem 1.5.1 says that each subset also has a greatest lower bound.

Back to our search for ways to determine *solutions* for systems of inequalities! The unknowns in the inequalities for the analysis of available assignments were the sets $\mathcal{A}[u]$ for all program points u. The complete lattice $\mathbb{D}$ of values for these unknowns is the powerset lattice 2^{Ass}, where the partial order is the superset relation $\supseteq$ as mentioned above.

All inequalities, $\mathcal{A}[v] \subseteq \{[\![k]\!]^\sharp\,(\mathcal{A}[u])$, for the same unknown v will now be combined into one inequality by applying the least upper bound operator to the right sides of the original inequalities. This leads to the form:

$$\begin{array}{ll}\mathcal{A}[\mathit{start}] & \subseteq \emptyset \\ \mathcal{A}[v] & \subseteq \bigcap\{[\![k]\!]^\sharp\,(\mathcal{A}[u]) \mid k = (u, \mathit{lab}, v) \;\; \text{edge}\} \text{ for } v \neq \mathit{start}\end{array}$$

The reformulation used above does not change the set of solutions due to

$$x \sqsupseteq d_1 \wedge \ldots \wedge x \sqsupseteq d_k \quad \text{iff} \quad x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\}$$

As a result, we obtain the generic form of a system of inequalities specifying a program-analysis problem:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad i = 1, \ldots, n$$

The functions $f_i : \mathbb{D}^n \to \mathbb{D}$ describe how the unknowns x_i depend on other unknowns.

One essential property of the functions f_i that define the right sides of the inequalities is their monotonicity, which guarantees that information, once computed, does not get lost. A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ between the two partial orders $\mathbb{D}_1, \mathbb{D}_2$ is *monotonic*, if $a \sqsubseteq b$ implies $f(a) \sqsubseteq f(b)$. For simplicity, the two partial orders in $\mathbb{D}_1$ and in $\mathbb{D}_2$ have been represented by the same symbol, $\sqsubseteq$.

Example 1.5.1 For a set U, let $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ be the powerset lattice with the partial order $\subseteq$. Each function f defined through $f\,x = (x \cap a) \cup b$ for $a, b \subseteq U$ is monotonic. A function g defined through $g\,x = a \setminus x$ for $a \neq \emptyset$, however, is not monotonic.

The functions inc and dec defined as $\mathsf{inc}\,x = x + 1$ and $\mathsf{dec}\,x = x - 1$ are monotonic on $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ together with the partial order "$\leq$".

The function inv defined through $\mathsf{inv}\,x = -x$ is not monotonic. □

If the functions $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ and $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ are monotonic so is their composition $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$.

If $\mathbb{D}_2$ is a complete lattice then the set $[\mathbb{D}_1 \to \mathbb{D}_2]$ of monotonic functions $f : \mathbb{D}_1 \to \mathbb{D}_2$ forms a complete lattice, where

$$f \sqsubseteq g \quad \text{iff} \quad f\,x \sqsubseteq g\,x \quad \text{for all } x \in \mathbb{D}_1$$

holds. In particular, for $F \subseteq [\mathbb{D}_1 \to \mathbb{D}_2]$ the function f defined by $f\,x = \bigsqcup\{g\,x \mid g \in F\}$ is again monotonic, and it is the least upper bound of the set F.

Let us consider the case $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$. For functions $f_i\,x = a_i \cap x \cup b_i$, where $a_i, b_i \subseteq U$, the operations "$\circ$", "$\sqcup$" and "$\sqcap$" can be described by operations on the sets a_i, b_i:

$$\begin{array}{lll}(f_2 \circ f_1)\,x = \boxed{a_1 \cap a_2} \cap x \cup \boxed{a_2 \cap b_1 \cup b_2} & & \textit{composition} \\ (f_1 \sqcup f_2)\,x = \boxed{(a_1 \cup a_2)} \cap x \cup \boxed{b_1 \cup b_2} & & \textit{union} \\ (f_1 \sqcap f_2)\,x = \boxed{(a_1 \cup b_1) \cap (a_2 \cup b_2)} \cap x \cup \boxed{b_1 \cap b_2} & & \textit{intersection}\end{array}$$

Functions of this shape occur often in so-called *bit-vector frameworks*.

Our goal is to find a least solution in a complete lattice $\mathbb{D}$ for the system of inequalities

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \qquad i = 1, \ldots, n \qquad (*)$$

where the functions $f_i : \mathbb{D}^n \to \mathbb{D}$ that define the right sides of the inequalities are monotonic. We exploit that $\mathbb{D}^n$ is a complete lattice if $\mathbb{D}$ is one. We combine the n functions f_i to one function $f : \mathbb{D}^n \to \mathbb{D}^n$ to simplify the presentation of the underlying problem. This function f is defined through $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, where $y_i = f_i(x_1, \ldots, x_n)$. It turns out that this constructions leads from monotonic component functions to a monotonic combined function. This transformation of the problem has reduced our problem to one of finding a least solution for a single inequality $x \sqsupseteq f\, x$, however in a slightly more complex complete lattice $\mathbb{D}^n$.

The search proceeds in the following way: It starts with an element d that is as small as possible, for instance, with $d = \underline{\perp} = (\perp, \ldots, \perp)$, the least element in $\mathbb{D}^n$. In case $d \sqsupseteq f\, d$ holds, a solution has been found. Otherwise, d is replaced by $f\, d$ and tested for being a solution. If not, f is applied to $f\, d$ and so on.

Example 1.5.2 Consider the complete lattice $\mathbb{D} = 2^{\{a,b,c\}}$ with the partial order $\sqsubseteq = \subseteq$ and the system of inequalities:

$$\begin{aligned} x_1 &\supseteq \{a\} \cup x_3 \\ x_2 &\supseteq x_3 \cap \{a, b\} \\ x_3 &\supseteq x_1 \cup \{c\} \end{aligned}$$

The iterative search for a least solution produces the results for the different iteration steps as they are listed in the following table:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	ditto
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

We observe that at least one value for the unknowns increases in each iteration until finally a solution is found. □

In fact, we can convince ourselves of the fact that this is the case for any complete lattice. More precisely, we can show:

Theorem 1.5.2 *Let $\mathbb{D}$ be a complete lattice and $f : \mathbb{D} \to \mathbb{D}$ be a monotonic function. Then the following two claims hold:*

1. *The sequence $\perp, f\perp, f^2\perp, \ldots$ is an ascending chain, i.e., it holds that $f^{i-1}\perp \sqsubseteq f^i\perp$ for all $i \geq 1$.*
2. *If $d = f^{n-1}\perp = f^n\perp$ then d is the least element d' satisfying $d' \sqsupseteq f(d')$.*

Proof. The first claim is proved by induction: For $i = 1$, the first claim holds since $f^{1-1} \perp = f^0 \perp = \perp$ is the least element of the complete lattice and therefore less than or equal to $f^1 \perp = f \perp$. Assume that the claim holds for $i - 1 \geq 1$, i.e., $f^{i-2} \perp \sqsubseteq f^{i-1} \perp$ holds. The monotonicity of the function f implies:

$$f^{i-1} \perp = f(f^{i-2} \perp) \sqsubseteq f(f^{i-1} \perp) = f^i \perp$$

We conclude that the claim also holds for i. Therefore, the claim holds for all $i \geq 1$.

Let us now regard the second claim. Assume that

$$d = f^{n-1} \perp \sqsupseteq f^n \perp$$

Then d is a solution of the system of inequalities $x \sqsupseteq f\,x$. Let us further assume we had another solution d' of the same system of inequalities. Thus, $d' \sqsupseteq f\,d'$ holds. It suffices to show that $f^i \perp \sqsubseteq d'$ holds for all $i \geq 0$. This is again shown by induction. It is the case for $i = 0$. Let $i > 0$ and $f^{i-1} \perp \sqsubseteq d'$. The monotonicity of f implies

$$f^i \perp = f(f^{i-1} \perp) \sqsubseteq f\,d' \sqsubseteq d'$$

since d' is a solution. This proves the claim for all i. □

Theorem 1.5.2 supplies us with a method to determine not only a solution, but even the least solution of an inequality, assuming that the ascending chain $f^i \perp$ eventually stabilizes, i.e., becomes constant at some i. It is therefore sufficient for the termination of our search for a fixed point that *all* ascending chains in $\mathbb{D}$ eventually stabilize. This is always the case in *finite* lattices.

The solution found by the described iterative method is a solution not only of the inequality $x \sqsupseteq f\,x$, but is even a solution of the equality $x = f\,x$, i.e., it is a *fixed point* of f.

What happens if not all ascending chains of the complete lattice eventually stabilize? Well, the iteration will not always terminate, although a least solution does exist.

Theorem 1.5.3 (Knaster–Tarski) *Each monotonic function $f : \mathbb{D} \to \mathbb{D}$ on a complete lattice $\mathbb{D}$ has a least fixed point d_0, which is also the least solution of the inequality $x \sqsupseteq f\,x$.*

Proof. A solution of the inequality $x \sqsupseteq f\,x$ is also called a *post-fixed point* of f. Let $P = \{d \in \mathbb{D} \mid d \sqsupseteq f\,d\}$ be the set of *post-fixed points* of f. We claim that the greatest lower bound d_0 of the set P is the least fixed point of f.

We first prove that d_0 is an element of P, i.e., is a post-fixed point of f. It is clear that $f\,d_0 \sqsubseteq f\,d \sqsubseteq d$ for each post-fixed point $d \in P$. Thus $f\,d_0$ is a lower bound of P and is therefore less than or equal to the greatest lower bound, i.e., $f\,d_0 \sqsubseteq d_0$.

d_0 is a lower bound of P, and it is an element of P. It is thus the *least* post-fixed point of f. It remains to prove that f also is a fixed point of f and therefore the least fixed point of f.

We know already that $f\,d_0 \sqsubseteq d_0$ holds. Let us consider the other direction: The monotonicity of f implies $f(f\,d_0) \sqsubseteq f\,d_0$. Therefore, $f\,d_0$ is a post-fixed point of f, i.e., $f\,d_0 \in P$. d_0 is a lower bound of P, which implies that $d_0 \sqsubseteq f\,d_0$ holds. □

Theorem 1.5.3 guarantees that each monotonic function f on a complete lattice has a least fixed point, which conicides with the least solution of the inequality $x \sqsupseteq f\,x$.

Example 1.5.3 Let us consider the complete lattice of the natural numbers augmented by ∞, i.e., $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ together with the partial order $\leq$. The function inc defined by $\mathsf{inc}\,x = x + 1$ is monotonic. We have:

$$inc^i \perp = inc^i\, 0 = i \quad \sqsubset \quad i + 1 = inc^{i+1} \perp$$

Therefore, this function has a least fixed point. However, this fixed point will not be reached after finitely many iteration steps. □

We can apply Theorem 1.5.3 to the complete lattice with the *dual* partial order $\sqsupseteq$ (instead of $\sqsubseteq$), and obtain that each monotonic function not only has a least, but also a *greatest* fixed point.

Example 1.5.4 Let us consider again the powerset lattice $\mathbb{D} = 2^U$ for a base set U and a function f with $f\,x = x \cap a \cup b$. This function is monotonic. It therefore has a least and a greatest fixed point. Fixed-point iteration delivers for f:

f	$f^k \perp$	$f^k \top$
0	$\emptyset$	U
1	b	$a \cup b$
2	b	$a \cup b$

□

With this newly acquired knowledge, we return to the application we had in mind, solving systems of inequalities

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \qquad i = 1, \ldots, n \qquad (*)$$

over a complete lattice $\mathbb{D}$ for monotonic functions $f_i : \mathbb{D}^n \to \mathbb{D}$. We know now that such a system of inequalities always has a least solution, which coincides with the least solution of the associated system of equations

$$x_i = f_i(x_1, \ldots, x_n), \qquad i = 1, \ldots, n$$

We will frequently meet complete lattices where ascending chains eventually stabilize in the instances of static program analysis considered in this book. In these cases, the described iterative procedure of repeated subsitution is able to compute the required solution. However, the "naive" fixed-point iteration according to Theorem 1.5.2 is often *inefficient.*

Example 1.5.5 Let us consider again the factorial program in Example 1.4.3. The fixed-point iteration to compute the least solution of the system of inequalities for available assignments is shown in Figure 1.6. The values for the unknowns stabilize only after five iterations. □

	1	2	3	4	5
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	$\emptyset$

Fig. 1.6. Naive fixed-point iteration for the program in Example 1.4.3

How could one improve on the naive fixed-point iteration? A significant improvement is achieved by so-called *round-robin iteration*. It is easily realized: The computation of a value in a new round does not use the values computed in the last round, but those that have already been computed in the current round.

```
for (i ← 1; i ≤ n; i++) x_i ← ⊥;
do {
        finished ← true;
        for (i ← 1; i ≤ n; i++) {
                new ← f_i(x_1, ..., x_n);
                if (¬(x_i ⊒ new)) {
                        finished ← false;
                        x_i ← x_i ⊔ new;
                }
        }
} while (¬finished);
```

Example 1.5.6 Let us consider again the system of inequalities for available assignments for the factorial program in Example 1.4.3. Figure 1.7 shows the corresponding round-robin iteration. It appears that three iteration rounds suffice. □

Let us have a closer look at round-robin iteration. The assignment $x_i \leftarrow x_i \sqcup new;$ in our implementation doesn't just overwrite the old value of x_i, but replaces it by the least upper bound of the old and the new value. We say that the algorithm *accumulates* the solution for x_i during the iteration. In the case of a monotonic function f_i, the least upper bound of old and new values for x_i is equal to the new value. For a non-monotonic function f_i, this need not be the case. The algorithm is robust enough to compute an ascending chain of values for each unknown x_i. It even computes at least some solution of the system of inequalities whenever it terminates.

The run time of the algorithm depends on the number of times the *do-while* loop is executed. Let h be the maximum of the lengths of all proper ascending chains, i.e., one with no repetitions

$$\bot \sqsubset d_1 \sqsubset d_2 \sqsubset \ldots \sqsubset d_h$$

	1	2	3
0	$\emptyset$	$\emptyset$	
1	$\{y \leftarrow 1\}$	$\emptyset$	
2	$\{y \leftarrow 1\}$	$\emptyset$	
3	$\emptyset$	$\emptyset$	dito
4	$\emptyset$	$\emptyset$	
5	$\emptyset$	$\emptyset$	

Fig. 1.7. Round-robin iteration for the program in Example 1.4.3

in the complete lattice $\mathbb{D}$. This number is called the *height* of the complete lattice $\mathbb{D}$. Let n be the number of unknowns in the system of inequalities. Round-robin iteration needs at most $h \cdot n$ rounds of the *do-while* loop until the values of all unknowns for the least solution are determined and possibly one more round to detect termination.

The bound $h \cdot n$ can be improved to n if the complete lattice is of the form 2^U for some base set U, if all functions f_i are constructed from constant sets and variables using only the operations $\cup$ and $\cap$. The reason for this follows: whether an element $u \in U$ is in the result set for the unknowns x_i is independent of whether any other element u' is contained in these sets. For which variables x_i a given element u is in the result sets for x_i can be determined in n iterations over the complete lattice $2^{\{u\}}$ of height 1. Round-robin iteration for all $u \in U$ is performed "in parallel" by using the complete lattice 2^U instead of the lattice $2^{\{u\}}$. These bounds concern the worst case. The least solution is often found in far fewer iterations if the variables are ordered appropriately.

Will this new iteration strategy also find the least solution if naive fixed-point iteration would have found the least solution? To answer this question, we assume again that all functions f_i are monotonic. Let $y_i^{(d)}$ be the ith component of $F^d \perp$ and $x_i^{(d)}$ be the value of x_i after the dth execution of the *do-while* loop of round-robin iteration. For all $i = 1, \ldots, n$ and $d \geq 0$ we prove the following claims:

1. $y_i^{(d)} \sqsubseteq x_i^{(d)} \sqsubseteq z_i$ for each solution $(z_1, \ldots, z_n)$ of the system of inequalities;
2. if the round-robin iteration terminates then the variables $x_1, \ldots, x_n$ will, after termination, contain the least solution of the system of inequalities;
3. $y_i^{(d)} \sqsubseteq x_i^{(d)}$.

Claim 1 is shown by induction. It imples that all approximations $x_i^{(d)}$ lie below the value of the unknown x_i in the least solution. Let us assume that the round-robin iteration terminates after round d. The values $x_i^{(d)}$ therefore satisfy the system of inequalities and thereby are a solution. Because of claim 1, they also form a least solution. This implies claim 2.

Claim 1 also entails that after d rounds the round-robin iteration computes values at least as large as the naive fixed-point iteration. If the naive fixed-point iteration terminates after round d, then the round-robin iteration terminates after at most d rounds.

We conclude that round-robin iteration is never slower than naive fixed-point iteration. Nevertheless, round-robin iteration can be performed more or less cleverly. Its efficiency substantially depends on the *order* in which the variables are reevaluated. It is *favorable* to reevaluate a variable x_i on which another variable x_j depends before this variable. This strategy leads to termination with a least solution after one execution of the *do-while* loop for an acyclic system of inequalities.

Example 1.5.7 Let us consider again the system of inequalities for the determination of available assignments for the factorial program in Example 1.4.3. Figure 1.8 shows a favorable and an unfavorable order of unknowns. In the unfavorable case, iteration

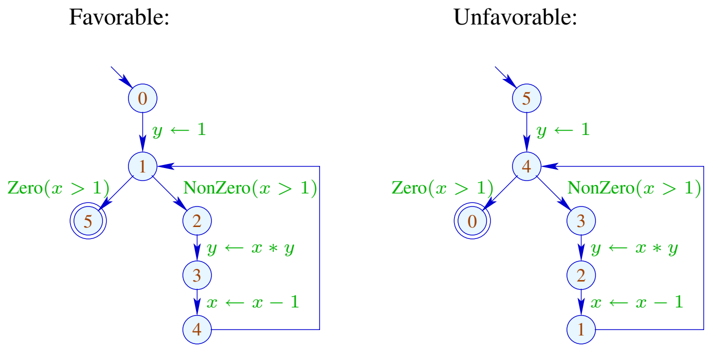


Fig. 1.8. A favorable and an unfavorable order of unknowns

needs four rounds for this program, as shown in Figure 1.9. □

	1	2	3	4	5
0	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	
1	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\emptyset$	
2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	ditto
3	$\{y \leftarrow 1\}$	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	
4	$\{y \leftarrow 1\}$	$\emptyset$	$\emptyset$	$\emptyset$	
5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	

Fig. 1.9. Round-robin iteration for the unfavorable order of Figure 1.8

1.6 Least Solution or MOP Solution?

Section 1.5 presented methods to determine least solutions of systems of inequalities. We will now see what the use of such least solutions is.

Consider again a complete lattice $\mathbb{D}$, as they appear in program analysis, and a system of inequalities of the form

$$\begin{array}{lll} \mathcal{I}[start] \sqsupseteq d_0 & & \\ \mathcal{I}[v] \quad \sqsupseteq [\![k]\!]^\sharp\,(\mathcal{I}[u]) & \text{for each edge} & k = (u, lab, v) \end{array}$$

where $d_0 \in \mathbb{D}$ is the value for the entry point of the program, and all abstract edge effects $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ are monotonic. There is a generic approach to solve an analysis problem, namely to select a complete lattice, fix a start value, and define the abstract edge effects by monotonic functions. This is often called a *monotonic analysis framework*. We will see more instances of this generic approach in the coming sections.

For each program point v, the value

$$\mathcal{I}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \to^* v\}$$

is to be determined. The mapping $\mathcal{I}^*$ is called the *merge over all paths* solution (in short: MOP solution) of the analysis problem. The following theorem clarifies the relation between the least solution and the MOP solution of a system of inequalities.

Theorem 1.6.1 (Kam, Ullman 1975) *Let $\mathcal{I}$ be the least solution of the system of inequalities. For each program point v*

$$\mathcal{I}[v] \;\sqsupseteq\; \mathcal{I}^*[v]$$

holds. This means that for each path π from program entry to v, we have:

$$\mathcal{I}[v] \;\sqsupseteq\; [\![\pi]\!]^\sharp\, d_0\ . \qquad (*)$$

Proof. We prove the claim $(*)$ by induction over the length of π. For the empty path π, i.e., $\pi = \epsilon$, we have:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[start]$$

Otherwise π is of the form $\pi = \pi' k$ for an edge $k = (u, lab, v)$. According to the induction hypothesis, the claim holds for the shorter path π', that is, $[\![\pi']\!]^\sharp\, d_0 \sqsubseteq \mathcal{I}[u]$. It follows that:

$$\begin{array}{rll} [\![\pi]\!]^\sharp\, d_0 & = [\![k]\!]^\sharp\,([\![\pi']\!]^\sharp\, d_0) & \\ & \sqsubseteq [\![k]\!]^\sharp\,(\mathcal{I}[u]) & \text{since } [\![k]\!]^\sharp \text{ is monotonic} \\ & \sqsubseteq \mathcal{I}[v] & \text{since } \mathcal{I} \text{ is a solution} \end{array}$$

This proves the claim. $\square$

Theorem 1.6.1 is somewhat disappointing. We would have hoped that the least solution was the same as the MOP solution. Instead, the theorem tells us that, in general, the least solution is only an upper bound of the MOP solution. So, the least solution may be less precise than the MOP solution. In many practical cases, however, the two solutions agree. This is, in particular, the case if all functions $[\![k]\!]^\sharp$ are *distributive*. A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- *distributive*, if $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ holds for all nonempty subsets $X \subseteq \mathbb{D}$;
- *strict*, if $f\,\bot = \bot$;
- *totally distributive*, if f is distributive and strict.

Example 1.6.1 Let us consider the complete lattice $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ with the canonical order $\leq$. The function inc defined by $\mathsf{inc}\,x = x + 1$ is distributive, but not strict.

As another example, let us look at the function

$$\mathsf{add} : (\mathbb{N} \cup \{\infty\})^2 \to (\mathbb{N} \cup \{\infty\})$$

where $\mathsf{add}\,(x_1, x_2) = x_1 + x_2$, and where the complete lattice $(\mathbb{N} \cup \{\infty\})^2$ is component-wise ordered. We have:

$$\mathsf{add}\,\bot = \mathsf{add}\,(0,0) = 0 + 0 = 0$$

Therefore, this function is strict. But it is not distributive, as the following counter-example shows:

$$\begin{aligned} \mathsf{add}\,((1,4) \sqcup (4,1)) = \mathsf{add}\,(4,4) &= 8 \\ \neq 5 &= \mathsf{add}\,(1,4) \sqcup \mathsf{add}\,(4,1) \end{aligned}$$

□

Example 1.6.2 Let us again consider the powerset lattice $\mathbb{D} = 2^U$ with the partial order $\subseteq$. For all $a, b \subseteq U$ the function f defined by $f\,x = x \cap a \cup b$ is distributive since

$$\begin{aligned} (\bigcup X) \cap a \cup b &= \bigcup\{x \cap a \mid x \in X\} \cup b \\ &= \bigcup\{x \cap a \cup b \mid x \in X\} \\ &= \bigcup\{f\,x \mid x \in X\} \end{aligned}$$

for each nonempty subset $X \subseteq \mathbb{D}$. The function f is, however, strict only if $b = \emptyset$ holds.

Functions f of the form $f\,x = (x \cup a) \cap b$ have similar properties on the powerset lattice $\mathbb{D} = 2^U$ with the reversed order $\supseteq$. For this partial order, distributivity means that for each nonempty subset $X \subseteq 2^U$ it holds that $f(\bigcap X) = \bigcap\{f\,x \mid x \in X\}$. □

There exists a precise characterization of all distributive functions if their domain is an *atomic* lattice. Let $\mathbb{A}$ be a complete lattice. An element $a \in \mathbb{A}$ is called *atomic* if $a \neq \bot$ holds and the only elements $a' \in \mathbb{A}$ with $a' \sqsubseteq a$ are the elements $a' = \bot$

and $a' = a$. A complete lattice $\mathbb{A}$ is called atomic if each element $d \in \mathbb{A}$ is the least upper bound of all atomic elements $a \sqsubseteq d$ in $\mathbb{A}$.

In the complete lattice $\mathbb{N} \cup \{\infty\}$ of Example 1.6.1, 1 is the only atomic element. Therefore, this lattice is not atomic. In the powerset lattice 2^U, ordered by the subset relation $\subseteq$, the atomic elements are the singleton sets $\{u\}, u \in U$. In the powerset lattice with the same base set, but the reversed order $\supseteq$, the atomic elements are the sets $(U \backslash \{u\}), u \in U$. The next theorem states that for atomic lattices distributive functions are uniquely determined by their values for the least element $\perp$ and for the atomic elements.

Theorem 1.6.2 *Let $\mathbb{A}$ and $\mathbb{D}$ be complete lattices where $\mathbb{A}$ is atomic. Let $A \subseteq \mathbb{A}$ be the set of atomic elements of $\mathbb{A}$. It holds that*

1. *Two distributive functions $f, g : \mathbb{A} \to \mathbb{D}$ are equal if and only if $f(\perp) = g(\perp)$ and $f(a) = g(a)$ for all $a \in A$.*
2. *Each pair (d, h) such that $d \in \mathbb{D}$ and $h : A \to \mathbb{D}$ define a distributive function $f_{d,h} : \mathbb{A} \to \mathbb{D}$ by:*

$$f_{d,h}(x) = d \sqcup \bigsqcup \{h(a) \mid a \in A, a \sqsubseteq x\}, \qquad x \in \mathbb{A}$$

Proof. We only prove the first claim. If the functions f and g are equal they agree on $\perp$ and the atomic elements of $\mathbb{A}$. For the opposite direction, we regard an arbitrary element $x \in \mathbb{A}$. For $x = \perp$ holds $f(x) = g(x)$ according to our assumption. For $x \neq \perp$, the set $A_x = \{a \in A \mid a \sqsubseteq x\}$ is not empty. It follows that:

$$\begin{aligned} f(x) &= f(\textstyle\bigsqcup A_x) \\ &= \textstyle\bigsqcup \{f(a) \mid a \in A, a \sqsubseteq x\} \\ &= \textstyle\bigsqcup \{g(a) \mid a \in A, a \sqsubseteq x\} = g(x) \end{aligned}$$

which was to be proved. □

Note that each distributive function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is also monotonic. $a \sqsubseteq b$ holds if and only if $a \sqcup b = b$ holds. If $a \sqsubseteq b$ holds we have:

$$f\,b = f\,(a \sqcup b) = f\,a \sqcup f\,b$$

Consequently, we have $f\,a \sqsubseteq f\,b$, what was to be shown. □

There is an important theorem for program analyses with distributive edge effects:

Theorem 1.6.3 (Kildall 1972) *Assume that every program point v is reachable from the program's entry point. Assume further that all edge effects $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ are distributive. The least solution $\mathcal{I}$ of the system of inequalities agrees with the MOP solution $\mathcal{I}^*$, i.e.,*

$$\mathcal{I}^*[v] = \mathcal{I}[v]$$

for all program points v.

Proof. Because of Theorem 1.6.1 it suffices to show that $\mathcal{I}[v] \sqsubseteq \mathcal{I}^*[v]$ holds for all v. Since $\mathcal{I}$ is the least solution of the system of inequalities it suffices to show that under the given circumstances, $\mathcal{I}^*$ is a solution, that is, satisfies all inequalities. For the entry point *start* of the program, we have:

$$\mathcal{I}^*[\mathit{start}] = \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \rightarrow^* \mathit{start}\} \sqsupseteq [\![\epsilon]\!]^\sharp\, d_0 \sqsupseteq d_0$$

For each edge $k = (u, \mathit{lab}, v)$ we check that

$$\begin{aligned}
\mathcal{I}^*[v] &= \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \rightarrow^* v\} \\
&\sqsupseteq \bigsqcup\{[\![\pi' k]\!]^\sharp\, d_0 \mid \pi' : \mathit{start} \rightarrow^* u\} \\
&= \bigsqcup\{[\![k]\!]^\sharp\, ([\![\pi']\!]^\sharp\, d_0) \mid \pi' : \mathit{start} \rightarrow^* u\} \\
&= [\![k]\!]^\sharp\, (\bigsqcup\{[\![\pi']\!]^\sharp\, d_0 \mid \pi' : \mathit{start} \rightarrow^* u\}) \\
&= [\![k]\!]^\sharp\, (\mathcal{I}^*[u])
\end{aligned}$$

The next to last equality holds since the set $\{\pi' \mid \pi' : \mathit{start} \rightarrow^* u\}$ of all paths from the entry point of the program *start* to u is not empty, and since the abstract edge effect $[\![k]\!]^\sharp$ is distributive. We conclude that $\mathcal{I}^*$ satisfies all inequalities. This proves the claim. □

The following example shows that in Theorem 1.6.3, the assumption that all program points are reachable is necessary.

Example 1.6.3 Regard the control-flow graph of Fig. 1.10. As complete lattice we

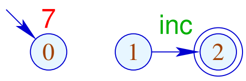


Fig. 1.10. A control-flow graph showing the consequences of unreachability

choose $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ with the canonical order $\leq$. As single edge effect we choose the distributive function inc. For an arbitrary starting value at the entry point, we have

$$\begin{aligned}
\mathcal{I}[2] &= \mathsf{inc}\,(\mathcal{I}[1]) \\
&= \mathsf{inc}\, 0 \\
&= 1
\end{aligned}$$

On the other hand, we have:

$$\mathcal{I}^*[2] = \bigsqcup \emptyset = 0$$

since there is no path from the entry point 0 to program point 2. It follows that the MOP solution is different from the least solution. □

It is not critical to assume that all program points are reachable. Unreachable program points can be easily identified and then removed without changing the semantics of the program.

Conclusion 1.6.1 We gather all the observations about analysis frameworks and apply them to the analysis of the availability of assignments.

- The MOP solution and the least solution of a system of inequalities are the same if all edge effects are distributive.
- Any solution to the system of inequalities is an upper bound for the MOP solution if the edge effects are only monotonic.
- Round-robin iteration can be used to determine the least solution if all ascending chains in the complete lattice have finite lengths.

In the application we have considered so far, the analysis of available assignments, the complete lattice $\mathbb{D} = 2^{Ass}$ is a finite powerset lattice with the order $\supseteq$, and the abstract edge effects $[\![k]\!]^\sharp$ are functions f of the form

$$f\,x = x\backslash b \cup a = (x \cup a) \cap (\bar{b} \cup a)$$

for $\bar{b} = \mathit{Ass}\backslash b$. Example 1.6.2 showed that all such functions are distributive. Thus, round-robin iteration for correpsonding systems of inequalities computes the MOP solution, provided that all program points are reachable from the program entry point. □

We can now conclude the section about the avoidance of redundant computations. The transformation we presented, however, has several disadvantages:

The described analysis for available assignments discovers relatively few redundant computations because it requires an available expression, i.e., one whose reevaluation could be avoided, to be available in the same variable along all paths. It also misses all available expressions in conditions and index expressions, whose value is not available in a variable. The compiler could transform the program in order to make this program analysis more effective. This transformation would introduce a unique temporary variable T_e for each expression e and insert an assignment of e into T_e at each occurrence of e. This transformation is the subject of Exercise 5.

Several redundant occurrences of the same expression would be replaced by reshuffling values between variables. This reshuffling of values is in many cases superfluous. Later sections of the book show how to eliminate such inefficiencies.

1.7 Removal of Assignments to Dead Variables

So far, we have only met a single optimizing transformation. It replaced the recomputation of an expression by accessing a previously computed value, provided that this value was guaranteed to be available in a variable. To present this transformation and, in particular, to prove properties like correctness and termination of the associated static program analysis, we introduced an operational semantics of our tiny