Helmut Seidl
Reinhard Wilhelm
Sebastian Hack

Compiler Design

Analysis and Transformation

Springer

Compiler Design

Helmut Seidl · Reinhard Wilhelm
Sebastian Hack

Compiler Design

Analysis and Transformation

Springer

Helmut Seidl
Fakultät für Informatik
Technische Universität München
Garching, Germany

Sebastian Hack
Programming Group
Universität des Saarlandes
Saarbrücken, Germany

Reinhard Wilhelm
Compiler Research Group
Universität des Saarlandes
Saarbrücken, Germany

ISBN 978-3-642-17547-3 ISBN 978-3-642-17548-0 (eBook)
DOI 10.1007/978-3-642-17548-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012940955

ACM Codes: D.1, D.3, D.2

Preface

Compilers for programming languages should translate source-language programs correctly into target-language programs, often programs of a machine language. But not only that; they should often generate target-machine *code* that is as efficient as possible. This book deals with this problem, namely the methods to improve the efficiency of target programs by a compiler.

The history of this particular subarea of compilation dates back to the early days of computer science. In the 1950s, a team at IBM led by John Backus implemented a first compiler for the programming language FORTRAN. The target machine was the IBM 704, which was, according to today's standards, an incredibly small and incredibly slow machine. This motivated the team to think about a translation that would efficiently exploit the very modest machine resources. This was the birth of "optimizing compilers".

FORTRAN is an imperative programming language designed for numerical computations. It offers arrays as data structures to store mathematical objects such as vectors and matrices, and it offers loops to formulate iterative algorithms on these objects. Arrays in FORTRAN, as well as in ALGOL 60, are very close to the mathematical objects that are to be stored in them.

The descriptional comfort enjoyed by the numerical analyst was at odds with the requirement of run-time efficiency of generated target programs. Several sources for this clash were recognized, and methods to deal with them were discovered. Elements of a multidimensional array are selected through sequences of integer-valued expressions, which may lead to complex and expensive computations. Some numerical computations use the same or similar index expressions at different places in the program. Translating them naively may lead to repeatedly computing the same values. Loops often step through arrays with a constant increment or decrement. This may allow us to improve the efficiency by computing the next address using the address used in the last step instead of computing the address anew. By now, it should be clear that arrays and loops represent many challenges if the compiler is to improve a program's efficiency compared to a straightforward translation.

Already the first FORTRAN compiler implemented several efficiency improving program transformations, called *optimizing transformations*. They should, however, be carefully applied. Otherwise, they would change the semantics of the program. Most such transformations have *applicability conditions*, which when satisfied guarantee the preservation of the semantics. These conditions, in general, depend on nonlocal properties of the program, which have to be determined by a *static analysis* of the program performed by the compiler.

This led to the development of *data-flow analysis*. This name was probably chosen to express that it determines the flow of properties of program variables through programs. The underlying theory was developed in the 1970s when the semantics of programming languages had been put on a solid mathematical basis. Two doctoral dissertations had the greatest impact on this field; they were written by Gary A. Kildall (1972) and by Patrick Cousot (1978). Kildall clarified the lattice-theoretic foundations of data-flow analysis. Cousot established the relation between the semantics of a programming language and static analyses of programs written in this language. He therefore called such a semantics-based program analysis *abstract interpretation*. This relation to the language semantics allows for a correctness proof of static analyses and even for the design of analyses that are correct by construction. Static program analysis in this book always means *sound static analysis*. This means that the results of such a static analysis can be trusted. A property of a program determined by a static analysis holds for all executions of the program.

The origins of data-flow analysis and abstract interpretation thus lie in the area of compilation. However, static analysis has emancipated itself from its origins and has become an important *verification method*. Static analyses are routinely used in industry to prove *safety properties* of programs such as the absence of run-time errors. Soundness of the analyses is mandatory here as well. If a sound static analysis determines that a certain run-time error will never occur at a program point, this holds for all executions of the program. However, it may be that a certain run-time error can never happen at a program point, but the analysis is unable to determine this fact. Such analyses thus are *sound*, but may be *incomplete*. This is in contrast with bug-chasing static analysis, which may fail to detect some errors and may warn about errors that will never occur. These analyses may be *unsound* and *incomplete*.

Static analyses are also used to prove partial correctness of programs and to check synchronization properties of concurrent programs. Finally, they are used to determine *execution-time bounds* for embedded real-time systems. Static analyses have become an indispensable tool for the development of reliable software.

This book treats the compilation phase that attempts to improve the efficiency of programs by semantics-preserving transformations. It introduces the necessary theory of static program analysis and describes in a precise way both particular static analyses and program transformations. The basis for both is a simple programming language, for which an operational semantics is presented.

The volume *Wilhelm and Seidl: Compiler Design: Virtual Machines* treats several programming paradigms. This volume, therefore, describes analyses and transformations for imperative and functional programs. Functional languages are based on the λ-calculus and are equipped with a highly developed theory of program transformation.

Several colleagues and students contributed to the improvement of this book. We would particularly like to mention Jörg Herter and Iskren Chernev, who carefully read a draft of this translation and pointed out quite a number of problems.

We wish the reader an enjoyable and profitable reading.

München and Saarbrücken, November 2011

Helmut Seidl
Reinhard Wilhelm
Sebastian Hack

General literature

The list of monographs that give an overview of static program analysis and abstract interpretation is surprisingly short. The book by Matthew S. Hecht [Hec77], summarizing the classical knowledge about data-flow analysis is still worth reading. The anthology edited by Steven S. Muchnick and Neil D. Jones [MJ81], which was published only a few years later, contains many original and influential articles about the foundations of static program analysis and, in particular, the static analysis of recursive procedures and dynamically allocated data structures. A similar collection of articles about the static analysis of declarative programs was edited by Samson Abramsky and Chris Hankin [AH87]. A comprehensive and modern introduction is offered by Flemming Nielson, Hanne Riis Nielson and Chris Hankin [NNH99].

Several comprehensive treatments of compilation contain chapters about static analysis [AG04, CT04, ALSU07]. Steven S. Muchnick's monograph "Advanced Compiler Design and Implementation" [Muc97] contains an extensive treatment.The *Compiler Design Handbook*, edited by Y.N. Srikant and Priti Shankar [SS03], offers a chapter about shape analysis and about techniques to analyze object-oriented programs.

Ongoing attempts to prove compiler correctness [Ler09, TL09] have led to an increased interest in the correctness proofs of optimizing program transformations. Techniques for the systematic derivation of correct program transformations are described by Patrick and Radhia Cousot [CC02]. Automated correctness proofs of optimizing program transformations are described by Sorin Lerner [LMC03, LMRC05, KTL09].

Contents

Chapter 1
Foundations and Intraprocedural Optimization

1.1 Introduction

This section presents basic techniques to improve the quality of compiler-generated code. The quality metric need 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 and 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 ($a[i] > a[j]$) {
        $t \leftarrow a[j]$;
        $a[j] \leftarrow a[i]$;
        $a[i] \leftarrow t$;
    }
}
```

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 $t, ai, aj$;
    $ai \leftarrow *p$; $aj \leftarrow *q$;
    if ($ai > aj$) {
        $t \leftarrow aj$;
        $*q \leftarrow ai$;
        $*p \leftarrow t$;
    }
}
```

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

This second version is apparently more efficient, while 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, as 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. □

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, will never give wrong answers to the question posed by the applicability condition. Both proofs refer to an *operational semantics* as their basis.

Several optimizations are effective across several classes of programming languages. However, each programming language and also each class of programming languages additionally 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 does not play a comparably large role. For FORTRAN, the parallelization or vectorization of nested loops has greater impact.

The programming language, in particular its semantics, also has 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 difficult due to the 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 still may be 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 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 realized 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 can 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:

```
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 :   if $(R_1 > R_2)$ {
5 :           $A_3$      $\leftarrow A_0 + 1 * j;$
6 :           $t$        $\leftarrow M[A_3];$
7 :           $A_4$      $\leftarrow A_0 + 1 * j;$
8 :           $A_5$      $\leftarrow A_0 + 1 * i;$
9 :           $R_3$      $\leftarrow M[A_5];$
10 :          $M[A_4]$ $\leftarrow R_3;$
11 :          $A_6$      $\leftarrow A_0 + 1 * i;$
12 :          $M[A_6]$ $\leftarrow t;$
13 :  }                                   //
```

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 to this code?

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 required then x's value can be used instead of reevaluating the expression e. The program can be greatly simplified by exploiting all this information:

```
$A_1 \leftarrow A_0 + i;$
$R_1 \leftarrow M[A_1];$
$A_2 \leftarrow A_0 + j;$
$R_2 \leftarrow M[A_2];$
**if** $(R_1 > R_2)$ {
        $M[A_2] \leftarrow R_1;$
        $M[A_1] \leftarrow R_2;$
}
```

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 Redundant 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. Instead, we 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:

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

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 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 Background: 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:	NonZero (e) or Zero (e)
assignment:	$x \leftarrow e$
load:	$x \leftarrow M[e]$
store:	$M[e_1] \leftarrow e_2$
empty statement:	;

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 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

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

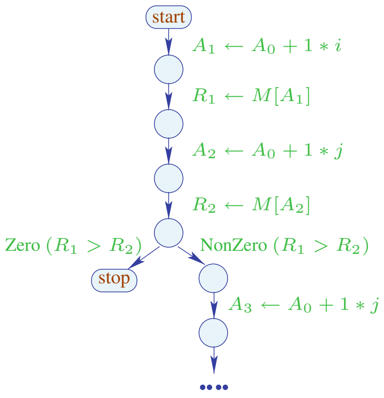


Fig. 1.1 A section of the control-flow graph for swap()

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:

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

An edge $k = (u, \mathit{lab}, v)$ with source vertex u, target vertex v and label lab defines a transformation $[\![k]\!]$ 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 $[\![k]\!]$ of the edge $k = (u, \mathit{lab}, v)$ only depends on its label lab:

$$[\![k]\!] = [\![\mathit{lab}]\!]$$

The edge effects $[\![\mathit{lab}]\!]$ are defined as follows:

$$\begin{array}{rcll}
[\![;]\!]\,(\rho, \mu) & = & (\rho, \mu) & \\
[\![\mathsf{NonZero}(e)]\!]\,(\rho, \mu) & = & (\rho, \mu) & \text{if } [\![e]\!]\,\rho \neq 0 \\
[\![\mathsf{Zero}(e)]\!]\,(\rho, \mu) & = & (\rho, \mu) & \text{if } [\![e]\!]\,\rho = 0 \\
[\![x \leftarrow e]\!]\,(\rho, \mu) & = & (\boxed{\rho \oplus \{x \mapsto [\![e]\!]\,\rho\}}, \mu) & \\
[\![x \leftarrow M[e]]\!]\,(\rho, \mu) & = & (\boxed{\rho \oplus \{x \mapsto \mu([\![e]\!]\rho)\}}, \mu) & \\
[\![M[e_1] \leftarrow e_2]\!]\,(\rho, \mu) & = & (\rho, \boxed{\mu \oplus \{[\![e_1]\!]\rho \mapsto [\![e_2]\!]\rho\}}) &
\end{array}$$

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 do not 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:

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

□

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 $\pi = k_1 \dots k_n$ of edges $k_i = (u_i, lab_i, u_{i+1})$ of the control-flow graph ($i = 1, \dots, n-1$), 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:

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

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 $\pi = k_1 \dots k_n$ 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 $k_{i+1}, \dots, k_n$ 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 $x \notin \mathsf{Vars}(e)$. An assignment violating this constraint cannot be considered as available at the subsequent program point and therefore are excluded from the set Ass. Let us assume that $A \subseteq Ass$ is 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:

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

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 are removed from the available-set. Following this, the actual assignment is 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 $\pi = k_1 \ldots k_n$ 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:

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

where $\mathit{start} \rightarrow^* v$ denotes the set of all paths from the entry point *start* of the program to the program point v. The sets $\mathcal{A}[v]$ are called the *merge-over-all-paths* (MOP) solution of the analysis problem. We temporarily postpone the question of

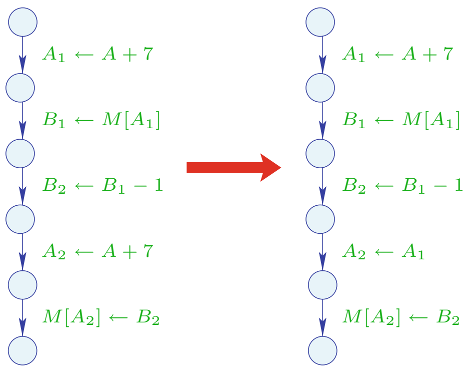
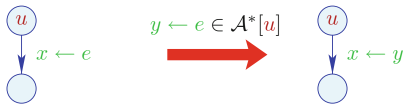


Fig. 1.2 Transformation RE applied to the code for $a[7]--;$

how to compute these sets. Instead, we discuss how the analysis information can be used for optimizing 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:



Analogous rules describe the replacement of expressions by variable accesses in conditions, 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 Consider the C statement a[7]--; — as implemented 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 $A_2 \leftarrow A + 7$. □

According to transformation RE, the evaluation of an expression is not always replaced by a variable look-up, when the evaluation is definitely repeated. Additionally, the result of the last evaluation still should be available in a variable, see Example 1.4.1. In order to increase applicability of the transformation, a compiler therefore could introduce a dedicated variable for each expression occurring in the program. To develop a corresponding transformation is the task of Exercise 5.

To decide when the application of the transformation RE is profitable can be non-trivial. Storing values of subexpressions costs storage resources. Access to stored values will be fast if the compiler succeeds to keep the values in registers. However, registers are scarce. Spending one of them for temporary storage may cause more costs somewhere else. Storing the value in memory, on the other hand, will result in long access times in which case it may be cheaper to reevaluate the expression.

Let us turn to the correctness proof of the described transformation. It can be split into two parts:

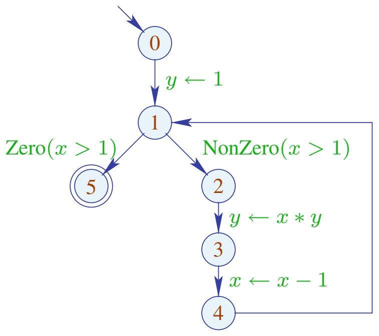
1. The proof of correctness of the abstract edge effects $[\![k]\!]^\sharp$ with respect to the definition of availability;
2. The proof of correctness of the replacement of definitely available expressions by accesses to variables.

We only treat the second part. Note that availability of expressions has been introduced by means of *semantic* terms, namely the *evaluation* of expressions and the assignment of their *values* to variables. In order to formulate the analysis, we then secretly switched to *syntactic* terms namely, *labeled* 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 thus 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 $y \not\in \mathsf{Vars}(e)$ holds and that $y \leftarrow e$ is available at u. It can be 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 guarantees the correctness of the analysis and the associated transformation. But 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 have 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.



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

Fig. 1.3 The system of inequalities for the factorial function

The abstract edge effects of our analysis, however, are total functions. They do not 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 assignment $x \leftarrow y + z$ 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:

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

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 Fig. 1.3. We see that the system of inequalities can be produced from the control-flow graph and the abstract edge transformers in a straightforward way. 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.

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

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

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

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 general treatment will provide us universal algorithms for solving virtually all program-analysis problems in this book.

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 preserve 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 Background: 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. In the case of available assignments, we therefore take the liberty to set $\sqsubseteq \; = \; \supseteq$. We start with definitions of partial orders and complete lattices.

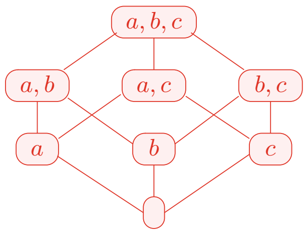
A set $\mathbb{D}$ together with a relation $\sqsubseteq$ on $\mathbb{D} \times \mathbb{D}$ is called a *partial order* if for all $a, b, c \in \mathbb{D}$ it holds that:

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

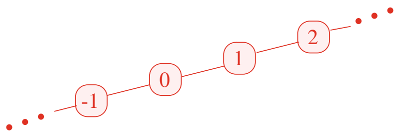
The sets we consider in this book consist of information at program points about potential or definite program behaviors. In our running example, such a piece of information at a program point is a set of available assignments. The ordering relation indicates *precision*. By convention, *less* means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

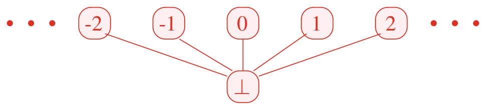
1. The set $2^{\{a,b,c\}}$ of all subsets of the set $\{a, b, c\}$ together with the relation $\subseteq$:



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* of X if

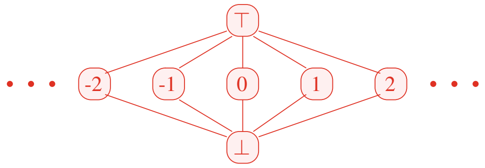
1. d is an upper bound of X, and
2. $d \sqsubseteq y$ holds for each upper bound y of X.

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 v. The abstract edge effects associated with these edges propagate different information towards v. The least upper bound operator then can be applied to combine the incoming information in a sound way to a value at v.

Each element is an upper bound of the empty set of elements of $\mathbb{D}$. The *least* upper bound $\perp$ of the empty set, therefore, is less than or equal to any other element of the complete lattice. This least element 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 base 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 lattice.
3. The set $\mathbb{Z}$ together with the equality relation $=$ is also not a clomplete lattice. A complete lattice, however, is obtained if an extra least element, $\perp$, *and* an extra greatest element, $\top$, is added:



This lattice $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$ contains only a minimum of pairs in the ordering relation. Such lattices are called *flat*.

In analogy to upper and least upper bounds, one can define lower and greatest lower bounds for subsets of partially ordered sets. 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.

We first show that g is 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 therefore

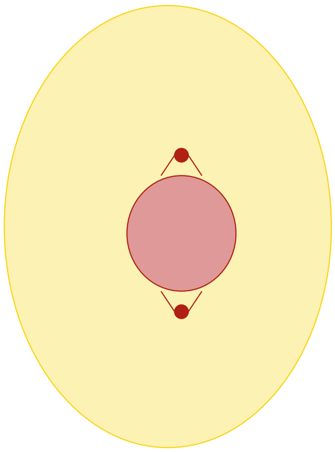


Fig. 1.5 The least *upper* bound and the greatest *lower* bound for a subset X

greater than or equal to the least upper bound of U, i.e., $g \sqsubseteq x$. Since x was an arbitrary element, g is in deed a lower bound of X.

Since g is an upper bound of U and therefore greater than or equal to each element in U, i.e., $u \sqsubseteq g$ for all $u \in U$, g is the greatest lower bound of X, which completes the proof. □

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 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! Recall that the unknowns in the inequalities for the analysis of available assignments are 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 $\supseteq$.

All inequalities for the same unknown v can 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:

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

This reformulation 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. This property guarantees that an increase of values on right-hand sides, may have no impact or increase also the values on the left-hand sides. 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:

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

Functions of this form 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 the 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. □

We convince ourselves of the fact that this is the case for any complete lattice given that right sides of equations are monotonic. More precisely, we 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

$$d = f^{n-1}\,\bot = f^n\,\bot$$

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

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

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 $f^i\,\bot$ 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 iterative method is the least solution not only of the inequality $x \sqsupseteq f\,x$, but is also the least solution of the equality $x = f\,x$, i.e., it is the least *fixed point* of f. What happens if not all ascending chains of the complete lattice eventually stabilize? Then the iteration may not always terminate. Nontheless, a least solution is guaranteed to 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 d_0 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., $f\;d_0 \in P$. Since d_0 is a lower bound of P, the inequality $d_0 \sqsubseteq f\;d_0$ follows. □

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, namely, ∞. This fixed point will not be reached after finitely many iteration steps. □

Theorem 1.5.3 can be applied to the complete lattice with the *dual* partial order $\sqsupseteq$ (instead of $\sqsubseteq$). Thus, we 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 our application, which is to solve systems of inequalities

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

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

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

In the instances of static program analysis considered in this book, we will frequently meet complete lattices where ascending chains eventually stabilize. In these cases, the iterative procedure of repeated evaluation of right-hand sides according to Theorem 1.5.2, is able to compute the required solution. This *naive* fixed-point iteration, however, is often quite *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 Fig. 1.6. The values for the unknowns stabilize only after four iterations. □

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

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

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

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

How can naive fixed-point iteration be improved? A significant improvement is already achieved by *round-robin iteration*. In round-robin iteration, the computation of a value in a new round does not use the values computed in the last round, but for each variable x_i the *last* value which has been computed for x_i. In the description of the algorithm, we must distinguish between the *unknowns* x_i and their *values*. For that purpose we introduce an array D that is indexed with the unknowns. The array component $D[x_i]$ always holds the value of the unknown x_i. The array D is successively updated until it finally contains the resulting variable assignment.

```
**for** $(i \leftarrow 1; i \leq n; i{+}{+})\ D[x_i] \leftarrow \bot;$
**do** {
    $\mathit{finished} \leftarrow$ **true**;
    **for** $(i \leftarrow 1; i \leq n; i{+}{+})$ {
        $\mathit{new} \leftarrow f_i(D[x_1], \ldots, D[x_n]);$
        **if** $(\neg(D[x_i] \sqsupseteq \mathit{new}))$ {
            $\mathit{finished} \leftarrow$ **false**;
            $D[x_i] \leftarrow D[x_i] \sqcup \mathit{new};$
        }
    }
} **while** $(\neg \mathit{finished})$
```

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 $D[x_i] \leftarrow D[x_i] \sqcup$ *new*; in our implementation does not 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 even in the non-monotonic case. Thus, it still returns *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

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

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 is the following: 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 at least in the monotonic case, 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 $D[x_i]$ 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 implies 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.

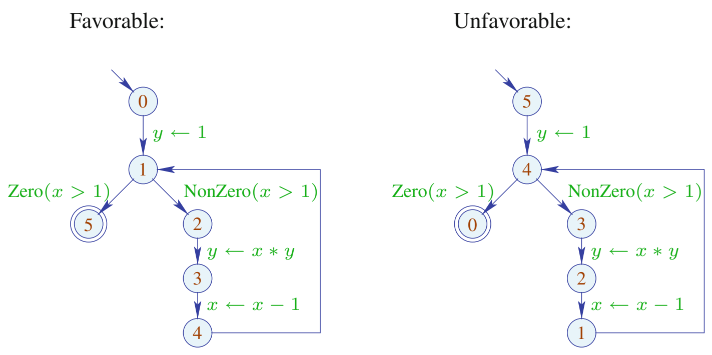


Fig. 1.8 A favorable and an unfavorable order of unknowns

	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 Fig. 1.8

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

1.6 Least Solution or MOP Solution?

Section 1.5 presented methods to determine least solutions of systems of inequalities. Let us now apply these techniques for solving program analysis problems such as availability of expressions in variables. Assume we are given a control-flow graph. The analysis problem consists in computing one information for each program point, i.e., each node v in the control-flow graph. A specifation of the analysis then consists of the following items:

- a complete lattice $\mathbb{D}$ of possible results for the program points;
- a start value $d_0 \in \mathbb{D}$ for the entry point *start* of the program; together with
- a function $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ for each edge k of the control-flow graph, which is monotonic. These functions are also called the *abstract edge effects* for the control-flow graph.

Each such specification constitutes an instance of the *monotonic analysis framework*. For availability of expressions in variables we provided such a specification, and we will see more instances of this framework in the coming sections.

Given an instance of the monotonic analysis framework, we can define for each program point v, the value

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

The mapping $\mathcal{I}^*$ is called the *merge over all paths* solution (in short: MOP solution) of the analysis problem. On the other hand, we can put up a system of inequalities which locally describes how information is propagated between nodes along the edges of the control-flow graph:

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

According to the theorems of the last section, this system has a least solution. And if the complete lattice $\mathbb{D}$ has finite height, this least solution can be computed by means, e.g., of round-robin iteration. The following theorem clarifies the relation between the least solution of the inequalities and the MOP solution of the analysis.

Theorem 1.6.1 (Kam and Ullman 1975) *Let $\mathcal{I}^*$ denote the MOP solution of an instance of the monotonic framework and $\mathcal{I}$ the least solution of the corresponding system of inequalities. Then 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:

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

This proves the claim. □

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 the least solution is only an upper bound of the MOP solution. This means that, in general, the least solution may be not as precise as the MOP solution and thus exhibit less opportunities for optimization as the MOP. Still, in many practical cases 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:

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

□

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 $\bot$ 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 $f(\bot) = g(\bot)$ 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 $\bot$ and the atomic elements of $\mathbb{A}$. For the opposite direction, we regard an arbitrary element $x \in \mathbb{A}$. For $x = \bot$ holds $f(x) = g(x)$ according to our assumption. For $x \neq \bot$, the set $A_x = \{a \in A \mid a \sqsubseteq x\}$ is not empty. It follows that:

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

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:

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

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

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

The next to last equality holds since the set $\{\pi' \mid \pi' : \mathit{start} \to^* u\}$ 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 choose $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ with the canonical order $\leq$. As single edge effect we choose

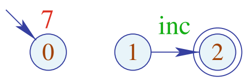


Fig. 1.10 A control-flow graph showing the consequences of unreachability

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 monotonic analysis frameworks.

- The MOP solution of a monotonic analysis framework is always less or equal to the least solution of the corresponding system of inequalities.
- If all edge effects are distributive and every program point is reachable from the entry point of the program, the MOP solution coincides with teh least solution.
- Round-robin iteration can be used to determine the least solution if all ascending chains in the complete lattice have finite lengths.

Let us apply these observations to the analysis of the availability of expressions in variables. In this analysis, the complete lattice $\mathbb{D} = 2^{Ass}$ is a finite powerset lattice with the order $\supseteq$. The value for the entry point of the program is $d_0 = \emptyset$, and the abstract edge effects $[\![k]\!]^\sharp$ are functions f of the form

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

for $\bar{b} = Ass\backslash b$. Example 1.6.2 shows 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 entry point of the program. □

We conclude the section about the removal of redundant computations. The transformation we presented has several disadvantages:

The analysis of availability of expressions in variables may fail to discover a redundant computation because it requires an available expression, i.e., an expression whose reevaluation could be avoided, to be available in the *same* variable along all paths. It also misses to identify expressions as available which occur in conditions or index expressions, because their values are not available in variables. At the expense of introducing extra auxiliary variables, the compiler could transform the program before-hand in order to make this program analysis more effective.

This transformation introduces unique temporary variables T_e for selected expressions e and insert an assignments of e into T_e at each occurrence of e (see Exercise 5). An assignment $x \leftarrow e$ thus is decomposed into the sequence

$$T_e \leftarrow e; x \leftarrow T_e$$

which is the evaluation of the right-hand side, followed by a variable-to-variable assignment. Most of these variable-to-variable assignments, though, turn out to be superfluous, and thus should better be removed. Transformations doing that are provided in Sect. 1.8.

1.7 Removal of Assignments to Dead Variables

So far, we have only met a single optimizing transformation. It replaces the recomputation of an expression by accessing a previously computed value, provided that this value is 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 language, complete lattices as domains of analysis information, and abstractions of the operational semantics to statically analyze programs. This foundational background enables us now to introduce more optimizing transformations and their associated program analyses quite easily.

Example 1.7.1 Let us regard the following example:

$$\begin{array}{ll} 0: & x \leftarrow y+2; \\ 1: & y \leftarrow 5; \\ 2: & x \leftarrow y+3; \end{array}$$

The value of program variable x at program points 0 and 1 is not of any importance. It is overwritten before being used. We therefore call variable x *dead* at this program point. The first assignment to x can be removed because the value of x before the second assignment is irrelevant for the semantics of the program. We also call this assignment *dead*. These notions are now made precise. □

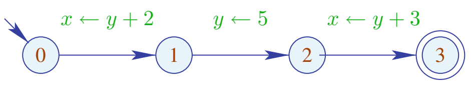


Fig. 1.11 Example for liveness of variables

Let us assume that, after program execution, the values of the variables from some set $X \subseteq$ *Vars* are still needed. This set X can be empty, in case all variables are only used within the program under analysis. However, the analysis to be presented can also be applied to individual procedure bodies. Returning from a procedure does not necessarily mean leaving the whole program. This means that accesses to *globally visible* variables may still happen. The set X should, in this case, be defined as the set of global variables.

The following definitions use the terms *definition* and *use*, well known in the compiler literature. A *definition* of a variable x is a statement which may change the value of x. In our small example language, the only definitions are assignments and loads where the left sides are x. A *use* of a variable x is an occurrence where the value of x is read. The sets of variables used and defined at an edge in the control-flow graph can be derived from the statement labeling the edge. For a label *lab*, they are determined by:

Lab	Used	Defined
;	$\emptyset$	$\emptyset$
NonZero(e)	Vars(e)	$\emptyset$
Zero(e)	Vars(e)	$\emptyset$
$x \leftarrow e$	Vars(e)	$\{x\}$
$x \leftarrow M[e]$	Vars(e)	$\{x\}$
$M[e_1] \leftarrow e_2$	Vars(e_1) $\cup$ Vars(e_2)	$\emptyset$

where Vars(e) denotes the set of program variables that occur in e.

We call a variable x *live* (relative to X) along path π to program exit, if $x \in X$ and π contains no definition of x, or if there exists at least one *use* of x in π, and the first use of x does not follow a definition of x. π can, in this case, be decomposed into $\pi = \pi_1\, k\, \pi_2$ such that the edge k contains a use of variable x and the prefix π_1 contains no definition of x. We will in the future omit the restriction, "relative to X" and tacitly assume a set X being given.

A variable x that is not live along π is called *dead* along π. A variable x is called (possibly) *live* at a program point v if x is live along at least one path from v to the program exit *stop*. Otherwise, we call x *dead* at program point v.

Whether a variable is possibly live or (definitely) dead at a program point depends on the possible continuations of program execution, this is the *future*. This is in contrast to the availability of assignments at a program point, which depends on the *history* of program execution before this program point is reached.

Example 1.7.2 Let us regard the simple program of Fig. 1.11. In this example, we assume that all variables are dead at the end of the program. There is only one path from each program point to the program exit. So, the sets of live and dead variables at each program point are easily determined. For the program points of the example, they are:

	Live	Dead
0	$\{y\}$	$\{x\}$
1	$\emptyset$	$\{x, y\}$
2	$\{y\}$	$\{x\}$
3	$\emptyset$	$\{x, y\}$

□

How can at each program point the set of live variables be computed? In principle, we proceed in the same way as we did for available assignments. The domain of possible values is $\mathbb{L} = 2^{Vars}$. Instead of providing a value for the entry point of the program, however, we now provide a value for the exit point, namely, the set X of variables which are live when exiting the program. Also, we provide for each edge an abstract edge effect. Since the liveness of variables at a program point does not depend on the history but on the future, the abstract effect of the edge $k = (u, lab, v)$ is a function $[\![k]\!]^\sharp$ that determines the set of variables possibly live at u, given a set of variables possibly live at v. Again, the abstract edge effect only depends on the label lab of the edge. this means that $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$, where

$$\begin{array}{ll}
[\![;]\!]^\sharp\, L & = L \\
[\![\mathsf{NonZero}(e)]\!]^\sharp\, L & = [\![\mathsf{Zero}(e)]\!]^\sharp\, L = L \cup \mathsf{Vars}(e) \\
[\![x \leftarrow e]\!]^\sharp\, L & = (L\backslash\{x\}) \cup \mathsf{Vars}(e) \\
[\![x \leftarrow M[e]]\!]^\sharp\, L & = (L\backslash\{x\}) \cup \mathsf{Vars}(e) \\
[\![M[e_1] \leftarrow e_2]\!]^\sharp\, L & = L \cup \mathsf{Vars}(e_1) \cup \mathsf{Vars}(e_2)
\end{array}$$

The abstract effects $[\![k]\!]^\sharp$ of edges k on a path $\pi = k_1 \ldots k_r$ can be composed to form the abstract effect of this path. We define:

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

The sequence of the edges is maintained by this function composition (and not reverted as for the analysis of expressions available in variables). The reason is that the abstract effect $[\![\pi]\!]^\sharp$ of the path π is to describe how a set of variables L live at the end of π is propagated through the path to compute the set of variables possibly live at the beginning of π.

The set of program variables possibly live at a program point v is obtained as the *union* of the sets of variables that are live along at least one program path π from v to the program exit, that is, as the union of the sets $[\![\pi]\!]^\sharp\, X$. Correspondingly, we define:

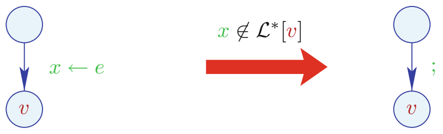
$$\mathcal{L}^*[v] \;=\; \bigcup\{[\![\pi]\!]^\sharp\, X \mid \pi : v \rightarrow^* stop\}$$

where $v \rightarrow^*$ *stop* denotes the set of all paths from v to the program exit *stop*. As partial order on the set $\mathbb{L}$ we choose the subset relation $\subseteq$. Intuitively, smaller sets of live variables mean larger sets of dead variables, which means more opportunities for optimization. The function $\mathcal{L}^*$ represents the MOP solution of our analysis problem.

Program analyses are called *forward analyses* if the value at a program point depends on the paths reaching this program point from program entry. Program analyses are called *backward analyses* if the value at a program point depends on the paths leaving that program point and reaching program exit. Liveness of variables therefore is a backward analysis. Available assignments, in contrast, is a forward analysis.

Transformation DE:

Let us assume that we are given the MOP solution $\mathcal{L}^*$. At each program point v, this is the set $\mathcal{L}^*[v]$. Its complement contains only variables that are definitely dead at v, i.e., dead along all program paths starting at v. Assignments to these variables are superfluous and can be removed by the following transformation DE, for Dead variable assignment Elimination:



Memory accesses whose results are not needed could also be removed in analogy to the removal of assignments. This could, however, change the semantics of the program if it were to remove an illegal access to memory, which—depending on the semantics of the programming language—may produce a side effect such as raising an exception. Useless accesses to memory are therefore not removed.

Transformation DE is called *dead-code elimination*. Correctness of this transformation is again shown in two steps:

1. The abstract edge effects are shown to correctly implement the definition of liveness.
2. The transformation is shown to preserve the semantics of programs.

We again consider the second step only. An important insight is that to show semantics preservation it is not necessary to show that the value of *each* variable at *each* program point remains invariant under the transformation. In fact, this is not the case here. For the applicability of the transformation, however, it suffices that the *observable behavior* of the original and the transformed program are the same. The only question is: what is potentially observable? Here, we demand that the program points traversed by program execution are the same, and that in each step the contents of memory coincide as well as the values of the variables in the set X at the end of program execution. Claim 2 then is that the value of a variable, dead at some program point v, does not influence the observable behavior of the program. To prove this, we consider the state s at v and the computation of a path π starting at v and reaching program exit. Remember, the *state* is a pair consisting of a concrete variable binding ρ and a memory. We show by induction over the length of program paths π:

(L) Let s' be a state that differs from s only in the values of dead variables. The state transformation $[\![\pi]\!]$ is also defined for s', and the states $[\![\pi']\!]\, s$ and $[\![\pi']\!]\, s'$ agree up to the values of dead variables for all prefixes π' of π.

The invariant (L) entails that two states at a program point v definitely lead to the same program behavior if they only disagree in the values of variable that are dead at v. To prove the correctness of the transformation it suffices to show that only the values of dead variables may be different during the execution of the original and the transformed program.

The computation of the set $\mathcal{L}^*[u]$ of variables possibly live at program point u works analogously to the way sets of definitely available assignments were computed. We set up an appropriate system of inequalities. Recall that, opposed to the available-assignments analysis where we fixed a start value at program entry *start*, we now fix a value X at program exit *stop* which consists of all variables which are live at program exit. Also, each edge $k = (u, lab, v)$ has an associated inequality that delivers the set of possibly live variables at u, given the set of possible live variables at v. Remember that the inequalities in the available-assignment analysis were oriented the other way around. The resulting system of inequalities is:

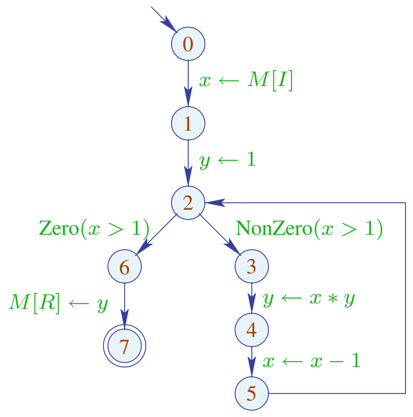
$$\begin{array}{lll} \mathcal{L}[stop] & \supseteq X & \\ \mathcal{L}[u] & \supseteq [\![k]\!]^\sharp\,(\mathcal{L}[v]) & \text{for an edge} \quad k = (u, lab, v) \end{array}$$

Thus, the difference in the systems of inequalities for forward and backward analyses only consists in the exchange of *start* and *stop* and in the reverted orientation of the edges.

The complete lattice in which the system of inequalities will be solved is finite. This means that all ascending chains will eventually stabilize. The abstract edge effects are monotonic. Round-robin iteration can thus be used to determine the least solution $\mathcal{L}$. In fact, the abstract edge effects are even distributive, which means that this least solution is the same as the MOP solution $\mathcal{L}^*$ provided that the program exit *stop* is reachable from each program point (cf. Theorem 1.6.3).

Example 1.7.3 We consider again the program for the factorial function assuming that it obtains its input and returns its output through memory cells, more precisely through the cells $M[I]$ and $M[R]$. No variables are assumed to be live at program exit. The control-flow graph and the system of inequalities derived from it are shown in Fig. 1.12. The system of inequalities closely corresponds to the control-flow graph. After all, the system of equations was extracted from the control-flow graph. However, it does not need to be explicitly constructed. Instead, fixed-point iteration could traverse the control-flow graph, executing the abstract edge effects associated with the traversed edges. The fixed-point iterator would be something like a *driver*. Another analysis can be conducted by executing it with the edges effects of this new analysis.

Round-robin iteration delivers the solution after only one round, given the right ordering of the unknowns:



$$\begin{aligned}
\mathcal{L}[0] &\supseteq (\mathcal{L}[1]\backslash\{x\}) \cup \{I\}\\
\mathcal{L}[1] &\supseteq \mathcal{L}[2]\backslash\{y\}\\
\mathcal{L}[2] &\supseteq (\mathcal{L}[6] \setminus \{x\}) \cup (\mathcal{L}[3] \cup \{x\})\\
\mathcal{L}[3] &\supseteq (\mathcal{L}[4]\backslash\{y\}) \cup \{x, y\}\\
\mathcal{L}[4] &\supseteq (\mathcal{L}[5]\backslash\{x\}) \cup \{x\}\\
\mathcal{L}[5] &\supseteq \mathcal{L}[2]\\
\mathcal{L}[6] &\supseteq \mathcal{L}[7] \setminus \{y, R\}\\
\mathcal{L}[7] &\supseteq \emptyset
\end{aligned}$$

Fig. 1.12 The system of inequalities for possibly live variables for the factorial program

	1	2
7	$\emptyset$	
6	$\{y, R\}$	
2	$\{x, y, R\}$	ditto
5	$\{x, y, R\}$	
4	$\{x, y, R\}$	
3	$\{x, y, R\}$	
1	$\{x, R\}$	
0	$\{I, R\}$	

We notice that no assignment in the factorial program has a dead left side. Therefore, transformation DE does not modify this program. □

The removal of assignments to dead variables can make other variables dead. This is witnessed in Fig. 1.13. This example shows a weakness of the analysis for dead variables: It may classify variables as live due to later uses in assignments to dead variables. A removal of such an assignment and a subsequent reanalysis would discover new dead variables. This iterative application of transformation and analysis is rather inefficient. In the example of live-variable analysis the repeated analysis can be avoided by *strengthening* the analysis. Strengthening leads to possibly smaller sets of possibly live variables. The new analysis works with a more restricted condition for liveness. The new notion, *true liveness*, uses the notion, *true use* of a variable on a path starting at a program point. A use in an assignment to a dead variable is not considered a true use. This renders the definition of true liveness recursive: true liveness depends on true use, which depends on true liveness.

Let us assume again that the values of variables in a set X are still used at program exit. We call a variable x *truly live* along a path π to program exit if $x \in X$ and π contains no definition of x or if π contains a *true use* of x, which occurs before any

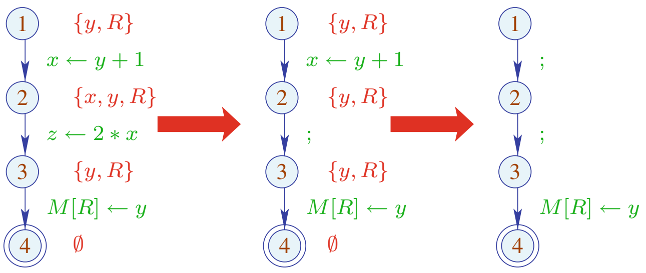


Fig. 1.13 Repeated application of transformation DE

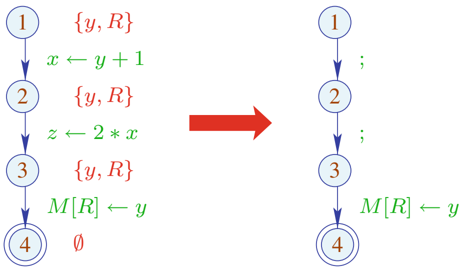


Fig. 1.14 Truly live variables

definition of x, i.e., π can be decomposed into $\pi = \pi_1\, k\, \pi_2$, such that π_1 contains no definition of x, and k contains a true use of x relative to π_2. The true use of variables at edge $k = (u, lab, v)$ is defined by:

Lab	y truly used
;	$false$
NonZero(e)	$y \in \mathsf{Vars}(e)$
Zero(e)	$y \in \mathsf{Vars}(e)$
$x \leftarrow e$	$y \in \mathsf{Vars}(e) \wedge x$ is truly live at v
$x \leftarrow M[e]$	$y \in \mathsf{Vars}(e) \wedge x$ is truly live at v
$M[e_1] \leftarrow e_2$	$y \in \mathsf{Vars}(e_1) \vee y \in \mathsf{Vars}(e_2)$

The additional condition that the assignment's left side must be truly live makes up the only difference to normal liveness.

Example 1.7.4 Consider the program in Fig. 1.14. Variable z is not live (nor truly live) at program point 2. Therefore, the variables on the right side of the corresponding assignment, i.e. x, are not truly used. Thus, x is not truly live at program point 1 since x is not truly used at the edge to program point 2. □

The abstract edge effects for true liveness are as follows:

$$\begin{array}{ll}
[\![;]\!]^\sharp\, L & = L \\
[\![\mathsf{NonZero}(e)]\!]^\sharp\, L = [\![\mathrm{Zero}(e)]\!]^\sharp\, L & = L \cup \mathsf{Vars}(e) \\
[\![x \leftarrow e]\!]^\sharp\, L & = (L\backslash\{x\}) \cup \ ((x \in L)\ ?\, \mathsf{Vars}(e)\ :\ \emptyset) \\
[\![x \leftarrow M[e]]\!]^\sharp\, L & = (L\backslash\{x\}) \cup \ ((x \in L)\ ?\, \mathsf{Vars}(e)\ :\ \emptyset) \\
[\![M[e_1] \leftarrow e_2]\!]^\sharp\, L & = L \cup \mathsf{Vars}(e_1) \cup \mathsf{Vars}(e_2)
\end{array}$$

For an element x and sets a, b, c, the conditional expression $(x \in a)\ ?\, b\ :\ c$ denotes the set:

$$(x \in a)\ ?\, b\ :\ c\ =\ \begin{cases} b \text{ if } & x \in a \\ c \text{ if } & x \notin a \end{cases}$$

The abstract edge effects for true liveness are thus more complex than those for plain liveness. However, they are still distributive! This follows from the fact that the new conditional operator is distributive provided that $c \subseteq b$ holds. To convince ourselves of this property, we consider an arbitrary powerset domain $\mathbb{D} = 2^U$ together with the partial order $\subseteq$ and the function:

$$f\ y = (x \in y)\ ?\, b :\ c$$

For an arbitrary nonempty set $Y \subseteq 2^U$, we calculate:

$$\begin{aligned}
f\,(\bigcup Y) &= (x \in \bigcup Y)\ ?\, b\ :\ c \\
&= (\bigvee\{x \in y \mid y \in Y\})\ ?\, b\ :\ c \\
&= c \cup \bigcup\{(x \in y)\ ?\, b\ :\ c \mid y \in Y\} \\
&= c \cup \bigcup\{f\ y \mid y \in Y\}
\end{aligned}$$

Theorem 1.6.2 has a more general implication:

Theorem 1.7.4 *Let U be a finite set and $f : 2^U \to 2^U$ be a function.*

1. *$f(x_1 \cup x_2) = f(x_1) \cup f(x_2)$ for all $x_1, x_2 \subseteq U$ holds if and only if f can be represented in the following form:*

$$f(x) = b_0 \cup ((u_1 \in x)\ ?\, b_1 : \emptyset) \cup \cdots \cup ((u_r \in x)\ ?\, b_r : \emptyset)$$

   for appropriate $u_i \in U$ and $b_i \subseteq U$.
2. *$f(x_1 \cap x_2) = f(x_1) \cap f(x_2)$ for all $x_1, x_2 \subseteq U$ holds if and only if f can be represented in the form:*

$$f(x) = b_0 \cap ((u_1 \in x)\ ?\, U : b_1) \cap \cdots \cap ((u_r \in x)\ ?\, U : b_r)$$

   for appropriate $u_i \in U$ and $b_i \subseteq U$. □

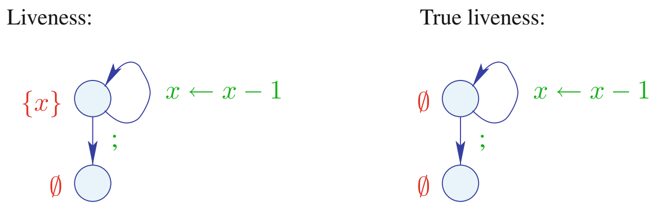


Fig. 1.15 True liveness in loops

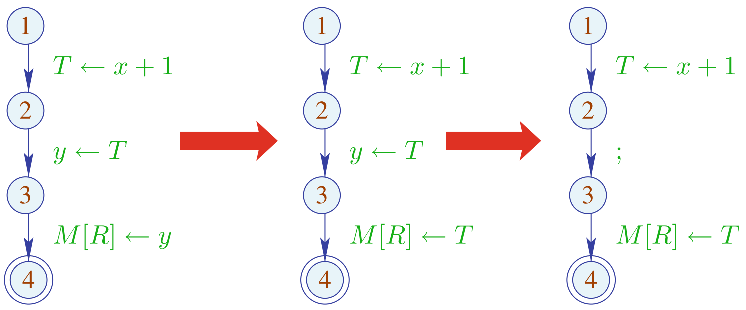


Fig. 1.16 A program with copying instructions

Note that the functions of Theorem 1.7.1 are closed under composition, least upper bounds, and greatest lower bounds (Exercise 11).

The least solution of systems of inequalities for true liveness agree with the MOP solutions due to the distributivity of the abstract edge effects. We must, however, require that program exit *stop* is reachable from each program point.

It is interesting to note that the analysis of true liveness discovers more superfluous assignments than repeated analysis of plain liveness and dead-code elimination.

Example 1.7.5 Figure 1.15 shows a loop in which a variable is modified that is only used in the loop. Plain liveness analysis cannot determine that this variable is dead, while true-liveness analysis is able to do so. □

1.8 Removal of Assignments Between Variables

Programs often contain assignments that simply copy values from one variable into another variable. These copy instructions may be the result of other optimizations or of conversions of one program representation into a different form.

Example 1.8.1 Consider the program in Fig. 1.16. Storing a value in variable T is useless in the given case, since the value of the expression is used exactly once. Variable T can be used directly instead of variable y since T is guaranteed to contain

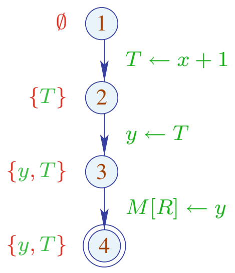


Fig. 1.17 Variables in Example 1.8.1 having the same value as T

the same value. This renders variable y dead at program point 2, such that the compiler can eliminate the assignment to y. The resulting program still contains variable T, but variable y is eliminated. □

For this kind of transformation, the compiler needs to know how the value of an expression is propagated by copy actions between variables. Such an analysis, therefore, is called *copy propagation*. Consider a variable x. The analysis maintains at each program point a set of variables guaranteed to contain the actual value of this variable. The use of a variable containing a copy of x can be replaced by a use of x.

Let $\mathbb{V} = \{V \subseteq \mathit{Vars} \mid x \in V\}$ be the complete lattice of all sets of program variables containing x, ordered by the superset relation $\supseteq$. It is intuitively clear that larger sets of variables guaranteed to contain the value of x will offer a greater chance for this optimization.

At program entry, only variable x is guaranteed to contain its own value. The start value of our analysis at program entry, therefore, is $V_0 = \{x\}$. The abstract edge effects again only depend on the edge labels. We define:

$$\begin{array}{lll}
[\![x \leftarrow e]\!]^\sharp\, V & = \{x\} & \\
[\![x \leftarrow M[e]]\!]^\sharp\, V & = \{x\} & \\
[\![z \leftarrow y]\!]^\sharp\, V & = \begin{cases} V \cup \{z\} & \text{if } \; y \in V \\ V \backslash \{z\} & \text{if } \; y \notin V \end{cases} & \\
[\![z \leftarrow r]\!]^\sharp\, V & = V \backslash \{z\} & \text{if } x \not\equiv z, r \notin \mathit{Vars}
\end{array}$$

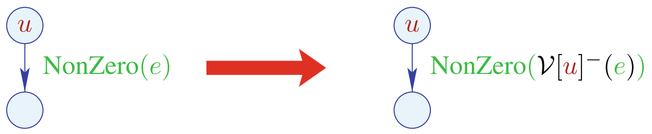
No other variable besides x definitely contains the value of x following an assignment $x \leftarrow e$ or reading from memory $x \leftarrow M[e]$. The other two cases treat assignments to variables z different from x. The abstract edge effects of all other edge labels do not change the incoming analysis information.

The result of the analysis for the program of Example 1.8.1 and the variable T is shown in Fig. 1.17. Note that the information is propagated through the program control-flow graph in a forward direction. Due to Theorem 1.7.1, all abstract edge effects are known to be distributive. This means that also for this problem the least solutions of the corresponding system of inequalities coincides with the MOP solution. Let $\mathcal{V}_x$ be this solution. By construction it follows from $z \in \mathcal{V}_x[u]$ that z contains the same value as x. The compiler may, therefore, replace accesses to z by accesses to x. We introduce the substitution $\mathcal{V}[u]^-$ to define the corresponding transformation:

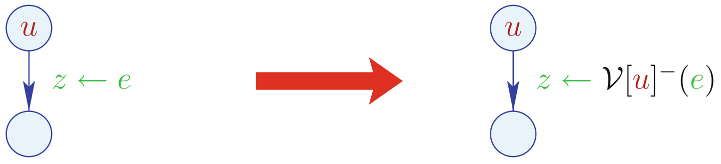
$$\mathcal{V}[u]^-\,z \;=\; \begin{cases} x & \text{if } z \in \mathcal{V}_x[u] \\ z & \text{otherwise} \end{cases}$$

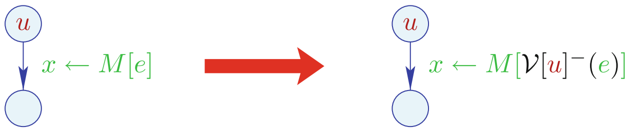
The transformation then is given be the following rules:

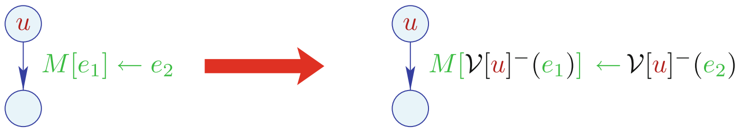
Transformation CE:



An analogous rule is applied to edges labeled by Zero (e).







Here, $\mathcal{V}[u]^-(e)$ denotes the application of the substitution $\mathcal{V}[u]^-$ to the expression e.

Example 1.8.2 It is time to have a look at a slightly bigger example to observe the cooperation of the different transformations.

In Example 1.4.2 we considered the implementation of the statement $a[7]--$; in our example language and showed how the second computation of the expression $A + 7$ could be replaced by an access to variable A_1. Figure 1.18 shows on the left side the results of transformation RE. The application of transformation CE replaces the use of variable A_2 by a use of variable A_1. The result of the transformation is the control-flow graph in the middle. The application of the transformation CE

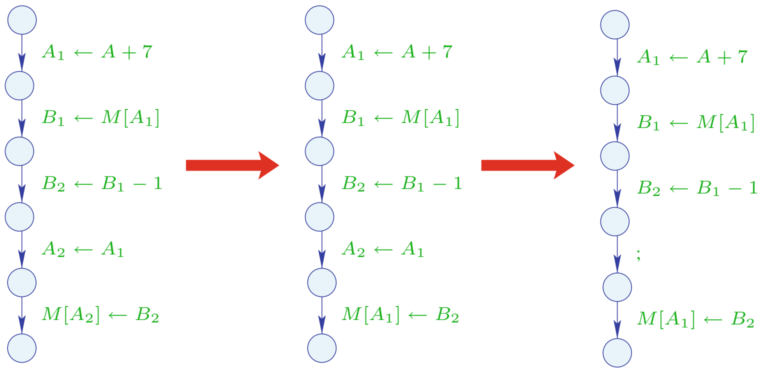


Fig. 1.18 The transformations CE and DE for the implementation of $a[7]--;$

```
x ← 7;
if (x > 0)
        M[A] ← B;
```

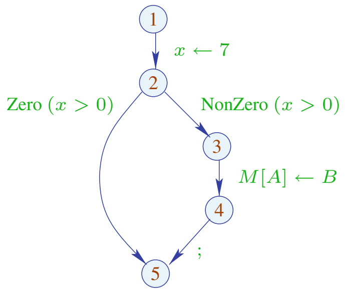


Fig. 1.19 An example for constant folding

renders variable A_2 dead. So, an application of transformation DE in the last step can eliminate the assignment to A_2. The inserted empty statement can later be removed in some clean-up step. □

1.9 Constant Folding

The goal of constant folding is to move parts of the computation from run time to compile time.

Example 1.9.1 Consider the program of Fig. 1.19. Variable x has always the value 7 at program point 2. Therefore, the condition $x > 0$ at the edges emanating from program point 2 will always evaluate to 1 such that the access to memory will be always executed. A compiler can therefore eliminate the condition following program point 2 (Fig. 1.20). The *else* part will become unreachable by eliminating the condition. □

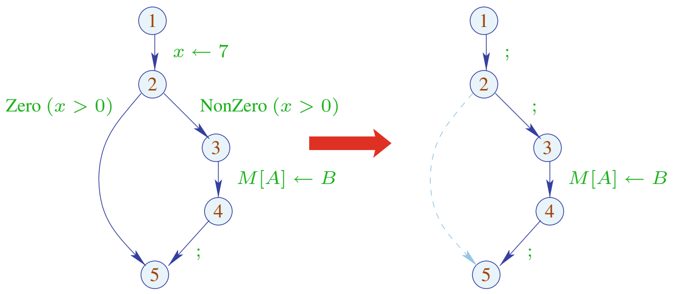


Fig. 1.20 An optimization of the example program of Fig. 1.19

The question is, do such inefficiencies occur in real programs? The answer is yes. There are several reasons why constant folding might find much to optimize. It is good programming style to use *named constants* to make programs easily modifyable. The compiler then is expected to propagate the constants through the program and fold expressions away where possible. Often, a program is written for many configurations of parameters. The automotive industry develops such "generic" programs, also called *program families*, which can be instantiated for many different types of cars, e.g. those with four or those with five gears, just by setting the named constants to different values.

Also, many programs are not written by programmers, but are generated from other programs. These generated programs tend to contain such inefficiencies. The compiler itself may produce constant subexpressions during the translation process, e.g., when it translates accesses to data structures such as arrays, or as the result of other program transformations.

Constant folding is a special case of *partial evaluation* of programs, which is the *essence of compilation* according to A. Ershov, one of the pioneers of compiler design. Partial evaluation performs computations on statically known parts of the program's state already at compile time. In this book, we are only concerned with constant folding. Our goal is to develop an analysis that computes for each program point v the following information: Which value does each program variable have when program control reaches v? It should be clear that a variable, in general, will have different values at a program point for different executions of the program or even during the same execution of a program, when control reaches this program point several times. The analysis can, at best, find the cases in which for every execution of the program and every time control reaches the program point a variable has the same value. We call this analysis *constant propagation*. As a side effect, this analysis also determines whether each program point is potentially reachable.

We construct the complete lattice for this analysis in two steps. In the first step, we design a partial order for the possible values of variables. To do this, we extend the set of integer numbers by an element $\top$, which represents an *unknown* value.

$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \qquad \text{and} \quad x \sqsubseteq y \quad \text{iff } y = \top \text{ or } x = y$$

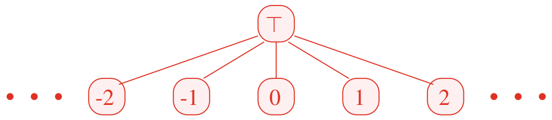


Fig. 1.21 The partial order $\mathbb{Z}^\top$ for values of variables

Figure 1.21 shows this partial order. The partial order $\mathbb{Z}^\top$ by itself is not yet a complete lattice, since it lacks a least element. In a second step, we construct the complete lattice of *abstract variable bindings* by defining

$$\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\bot = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\bot\},$$

i.e., $\mathbb{D}$ is the set of all functions mapping variables to abstract values, extended by an additional value, $\bot$ as the unique least element. We say an abstract binding $D \neq \bot$ *knows* the value of a variable x if $D\,x \in \mathbb{Z}$. If, however, $D\,x \notin \mathbb{Z}$, that is, if $D\,x = \top$, then the value of x is unknown. The value $\top$ for x means that the analysis could not determine one single value for x, perhaps, since x was found to have several distinct values in the course of execution.

The new element $\bot$ is associated with every program point that, according to the current fixed-point iteration is not yet known to be reachable. If the solution found by the fixed-point iteration still has program points with associated value $\bot$, these points cannot be reached by any program execution. We define an order on this set of *abstract states* by:

$$D_1 \sqsubseteq D_2 \quad \text{iff} \quad \bot = D_1 \quad \text{or} \quad D_1\,x \sqsubseteq D_2\,x \quad \text{for all } x \in \mathit{Vars}$$

The abstract variable binding $\bot$ denoting *not yet reachable* is considered as smaller than any other abstract state. The idea is that later, the corresponding program point still may turn out to be reachable and thus receive any abstract variable assignment $\neq \bot$. An abstract binding $D_1 \neq \bot$ is possibly better, i.e., less than or equal to another binding $D_2 \neq \bot$, if it agrees with D_2 on the values of all variables that D_2 knows, but possibly knows more values of variables than D_2. Intuitively, an abstract variable binding that knows values of more variables may lead to more optimizations and thus is better information. Going up in the partially ordered set $\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\bot$ thus means “forgetting” values of variables.

We want to show that $\mathbb{D}$ together with this order is a complete lattice. Consider a subset $X \subseteq \mathbb{D}$. Without loss of generality, we may assume that $\bot \notin X$. We have then $X \subseteq (\mathit{Vars} \to \mathbb{Z}^\top)$.

From $X = \emptyset$ follows $\bigsqcup X = \bot \in \mathbb{D}$. Therefore, $\mathbb{D}$ has a least upper bound for X. For $X \neq \emptyset$, the least upper bound $\bigsqcup X = D$ is given by:

$$D\,x = \bigsqcup\{f\,x \mid f \in X\} = \begin{cases} z & \text{if} \quad f\,x = z \quad \text{for all } f \in X \\ \top & \text{otherwise} \end{cases}$$

This shows that every subset X of $\mathbb{D}$ has a least upper bound and that $\mathbb{D}$ is a complete lattice. For each edge $k = (u, lab, v)$ we construct an abstract edge effect $[\![k]\!]^\sharp = [\![lab]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, which simulates the concrete computation. Since unreachability should be preserved by all abstract edge effects, we define all abstract effects as *strict*, i.e., $[\![lab]\!]^\sharp \perp = \perp$ holds for all edge labels lab.

Now let $D \neq \perp$ be an abstract variable binding. We need an *abstract* evaluation function for expressions to define the abstract edge effects. This function determines the value of an expression as far as possible for the given information in D. The abstract evaluation has to handle the situation that the precise value of a given expression cannot be determined in the given abstract variable binding D. This means that the expression should be evaluated to $\top$. The abstract evaluation of expressions works like the concrete evaluation of expressions as long as all operands of the operators in the expression are concrete values. To handle the case of an operand $\top$, the concrete arithmetic, Boolean, and comparison operators, $\Box$, are replaced by the corresponding *abstract* operators, $\Box^\sharp$, which are also able to deal with $\top$ operands. For binary operators, $\Box$, we define:

$$a \,\Box^\sharp\, b = \begin{cases} \top & \text{if } a = \top \text{ or } b = \top \\ a \,\Box\, b & \text{otherwise} \end{cases}$$

The result of the abstract evaluation of an expression shall be unknown, that is $\top$, whenever at least one of the operands is unknown.

This definition of the abstract operators is quite natural. Still, better information can be obtained for some combinations of operators and operand values by exploiting algebraic laws. For instance, knowing that one operand of a multiplication is 0 can be exploited to infer that the result is 0 no matter what the other operand is. More of these algebraic identities can be used to refine and improve the abstract evaluation.

Let us assume that we have defined an abstract operator $\Box^\sharp$ on abstract values for each concrete operator $\Box$. We then define the *abstract* evaluation

$$[\![e]\!]^\sharp \;:\; (Vars \to \mathbb{Z}^\top) \to \mathbb{Z}^\top$$

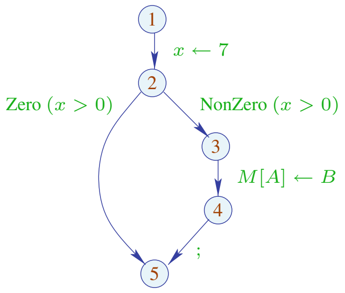
of an expression e by:

$$\begin{array}{rcll} [\![c]\!]^\sharp\, D & = & c & \\ [\![\Box\, e]\!]^\sharp\, D & = & \Box^\sharp\, [\![e]\!]^\sharp\, D & \text{for unary operators } \Box \\ [\![e_1 \,\Box\, e_2]\!]^\sharp\, D & = & [\![e_1]\!]^\sharp\, D \,\Box^\sharp\, [\![e_2]\!]^\sharp\, D & \text{for binary operators } \Box \end{array}$$

Example 1.9.2 Consider the abstract variable binding

$$D = \{x \mapsto 2, y \mapsto \top\}$$

We get:



1	$\{x \mapsto \top\}$
2	$\{x \mapsto 7\}$
3	$\{x \mapsto 7\}$
4	$\{x \mapsto 7\}$
5	$\bot \sqcup \{x \mapsto 7\} = \{x \mapsto 7\}$

Fig. 1.22 Solution of the system of inequalities of Fig. 1.19

$$\begin{aligned} [\![x + 7]\!]^\sharp\, D &= [\![x]\!]^\sharp\, D \;+^\sharp\; [\![7]\!]^\sharp\, D \\ &= 2 \;+^\sharp\; 7 \\ &= 9 \\ [\![x - y]\!]^\sharp\, D &= 2 \;-^\sharp\; \top \\ &= \top \end{aligned}$$

□

Next, we define the abstract edge effects $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$. We set $[\![lab]\!]^\sharp\, \bot = \bot$, and for $D \neq \bot$, we define:

$$\begin{aligned} [\![;]\!]^\sharp\, D &= D \\ [\![\mathsf{NonZero}\,(e)]\!]^\sharp\, D &= \begin{cases} \bot & \text{if } 0 = [\![e]\!]^\sharp\, D \\ D & \text{otherwise} \end{cases} \\ [\![\mathsf{Zero}\,(e)]\!]^\sharp\, D &= \begin{cases} \bot & \text{if } 0 \not\sqsubseteq [\![e]\!]^\sharp\, D \quad \text{cannot be zero} \\ D & \text{if } 0 \sqsubseteq [\![e]\!]^\sharp\, D \quad \text{could be zero} \end{cases} \\ [\![x \leftarrow e]\!]^\sharp\, D &= D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\ [\![x \leftarrow M[e]]\!]^\sharp\, D &= D \oplus \{x \mapsto \top\} \\ [\![M[e_1] \leftarrow e_2]\!]^\sharp\, D &= D \end{aligned}$$

The operator $\oplus$ changes a function at a given argument to a given value.

We assume that no values of variables are known at the start of program execution. Therefore, we select the abstract variable binding $D_\top = \{x \mapsto \top \mid x \in \mathit{Vars}\}$ for the program point *start*.

The abstract edge effects $[\![k]\!]^\sharp$ can, as usual, be composed to the abstract effects of paths $\pi = k_1 \dots k_r$ by:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \dots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Example 1.9.3 The least solution of the system of inequalities of our introductory example is shown in Fig. 1.22. □

How do we show the correctness of the computed information? This proof is based on the theory of *abstract interpretation* as developed by Patrick and Radhia Cousot in the late 1970s. We present this theory in a slightly simplified form. The main idea is to work with *abstract values*, which are *descriptions* of (sets of) *concrete values*. These descriptions are elements of a partial order $\mathbb{D}$. A *description relation*, Δ, relates concrete and abstract values. For $x\ \Delta\ a$ we say, "x is described by a".

This relation Δ should have the following properties:

$$x\ \Delta\ a_1 \quad \wedge \quad a_1 \sqsubseteq a_2 \quad \Longrightarrow \quad x\ \Delta\ a_2$$

If a_1 is a description of a concrete value x, and $a_1 \sqsubseteq a_2$ holds, then a_2 is also a description of x. We can define a *concretization*, γ, for such a description relation. It maps each abstract value $a \in \mathbb{D}$ to the set of all concrete values described by a:

$$\gamma\, a = \{x \mid x\ \Delta\ a\}$$

An abstract value a_2 that is greater than another abstract value a_1 describes a superset of the set of concrete values described by a_1. The greater abstract value, a_2, is therefore less precise information than the smaller value, a_1:

$$a_1 \sqsubseteq a_2 \quad \Longrightarrow \quad \gamma(a_1) \subseteq \gamma(a_2)$$

The description relation for constant propagation is built up in several steps. We start with a description relation $\Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top$ on the values of program variables and define:

$$z\ \Delta\ a \quad \text{iff} \quad z = a\ \vee\ a = \top$$

This description relation has the concretization:

$$\gamma\, a = \begin{cases} \{a\} & \text{if } a \sqsubset \top \\ \mathbb{Z} & \text{if } a = \top \end{cases}$$

We extend the description relation for values of program variables to one between concrete and abstract variable bindings. For simplicity, it gets the same name, Δ. This description relation $\Delta \subseteq (\mathit{Vars} \to \mathbb{Z}) \times (\mathit{Vars} \to \mathbb{Z}^\top)_\perp$ is defined by:

$$\rho\ \Delta\ D \quad \text{iff} \quad D \neq \perp\ \wedge\ \rho\, x \sqsubseteq D\, x \quad (\text{for all } x \in \mathit{Vars})$$

This definition of Δ implies that there exists no concrete variable binding ρ such that $\rho\ \Delta\ \perp$. Therefore, the concretization γ maps $\perp$ to the empty set. γ maps each abstract variable binding $D \neq \perp$ to the set of all concrete variable bindings that know for each variable x either $D\, x \in \mathbb{Z}$ or an arbitrary value, if $D\, x = \top$.

$$\gamma\, D = \{\rho \mid \forall x :\ (\rho\, x)\ \Delta\ (D\, x)\}$$

We have, for instance:

$$\{x \mapsto 1, y \mapsto -7\} \ \Delta \ \{x \mapsto \top, y \mapsto -7\}$$

The simple constant propagation, we consider here, ignores the values in memory. We can therefore describe program states (ρ, μ) just by abstract variable bindings, which only describe ρ. Overall, the description relation is defined by:

$$(\rho, \mu) \ \Delta \ D \quad \text{iff} \quad \rho \ \Delta \ D$$

The concretization γ returns:

$$\gamma\, D = \begin{cases} \emptyset & \text{if } D = \bot \\ \{(\rho, \mu) \mid \rho \in \gamma\, D\} & \text{otherwise} \end{cases}$$

We want to prove that each path π of the control-flow graph maintains the description relation Δ between concrete and abstract states. The claim is:

(K) If $s \ \Delta \ D$ holds, and if $[\![\pi]\!]\, s$ is defined then $([\![\pi]\!]\, s) \ \Delta \ ([\![\pi]\!]^\sharp\, D)$ holds.
The following diagram visualizes this claim:

$$\begin{tikzcd}
\boxed{s} \arrow[r, "{[\![\pi]\!]}"] \arrow[d, dash, "\Delta"'] & \boxed{s_1} \arrow[d, dash, dotted, "\Delta"] \\
\boxed{D} \arrow[r, "{[\![\pi]\!]^\sharp}"'] & \boxed{D_1}
\end{tikzcd}$$

Claim (K) implies in particular that

$$[\![\pi]\!]\, s \in \gamma\, ([\![\pi]\!]^\sharp\, D),$$

whenever $s \in \gamma(D)$ holds. Property (K) is formulated for arbitrary paths. It is sufficient for the proof to show (K) for a single edge k. The claim is then proved by induction on the length of paths since the claim trivially holds for paths of length 0. It therefore suffices to show that for each edge k and $s \ \Delta \ D$ that $([\![k]\!]\, s) \ \Delta \ ([\![k]\!]^\sharp\, D)$ holds whenever $[\![k]\!]\, s$ is defined.

The essential step in the proof of property (K) for an edge consists in showing for each expression e:

$$([\![e]\!]\, \rho) \Delta ([\![e]\!]^\sharp\, D), \qquad \text{if } \rho \Delta D. \qquad (**)$$

To prove claim $(**)$, we show for each operator $\Box$:

$$(x \,\square\, y) \;\;\Delta\;\; (x^\sharp \,\square^\sharp\, y^\sharp), \qquad \text{if} \;\; x \;\Delta\; x^\sharp \wedge y \;\Delta\; y^\sharp$$

Claim $(**)$ then follows by induction over the structure of expressions e. The claim about the relation between concrete and abstract operators has to be shown for each operator individually. For constant propagation with our simple definition of $\square^\sharp$, this is certainly fulfilled.

Overall, we wanted to prove that each edge $k = (u, lab, v)$ maintains the description relation Δ between concrete and abstract states. Stated differently, this means that the concrete and the abstract edge effects are compatible with the description relation. Let us return to this proof. The proof goes by case distinction according to the label *lab* of the edge.

We assume that $s = (\rho, \mu) \;\;\Delta\;\; D$ and that $D \neq \bot$.

Assignment, $x \leftarrow e$: We have:

$$\begin{array}{llll} [\![x \leftarrow e]\!]\, s & = (\rho_1, \mu) & \text{where} & \rho_1 \;= \rho \oplus \{x \mapsto [\![e]\!]\,\rho\} \\ [\![x \leftarrow e]\!]^\sharp\, D = D_1 & & \text{where} & D_1 = D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \end{array}$$

The claim $(\rho_1, \mu) \;\Delta\; D_1$ follows from the compatibility of the concrete and the abstract expression evaluation with the description relation Δ.

Read, $x \leftarrow M[e]$: We have:

$$\begin{array}{llll} [\![x \leftarrow M[e]]\!]\, s & = (\rho_1, \mu) & \text{where} & \rho_1 \;= \rho \oplus \{x \mapsto \mu\,([\![e]\!]\,\rho)\} \\ [\![x \leftarrow M[e]]\!]^\sharp\, D = D_1 & & \text{where} & D_1 = D \oplus \{x \mapsto \top\} \end{array}$$

The claim $(\rho_1, \mu) \;\Delta\; D_1$ follows since $\rho_1\, x \;\Delta\; \top$ holds.

Store, $M[e_1] \leftarrow e_2$:

The claim holds since neither the concrete nor the abstract edge effect modify the variable binding.

Condition, $\mathsf{Zero}(e)$:

Let $[\![\mathsf{Zero}(e)]\!]\, s$ be defined. We have $0 = ([\![e]\!]\,\rho) \;\Delta\; ([\![e]\!]^\sharp\, D)$.

Therefore, $[\![\mathsf{Zero}(e)]\!]^\sharp\, D = D \neq \bot$ holds, and the claim is shown.

Condition, $\mathsf{NonZero}(e)$:

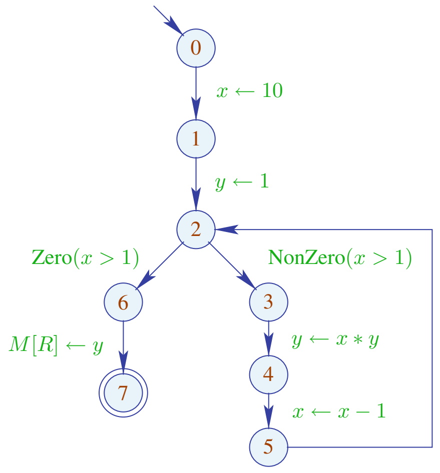
Let $[\![\mathsf{NonZero}(e)]\!]\, s$ be defined. We have $0 \neq ([\![e]\!]\,\rho) \;\Delta\; ([\![e]\!]^\sharp\, D)$.

It follows $[\![e]\!]^\sharp\, D \neq 0$, and we have: $[\![\mathsf{NonZero}(e)]\!]^\sharp\, D = D$, which implies the claim.

Altogether we conclude that the invariant (K) holds.

The MOP solution for constant propagation at a program point v is the least upper bound of all informations contributed by all paths from the entry point to v for the initial binding $D_\top$:

$$\mathcal{D}^*[v] \;=\; \bigsqcup \{[\![\pi]\!]^\sharp\; D_\top \mid \pi : \textit{start} \rightarrow^* v\}\,,$$



	1		2		3	
	x	y	x	y	x	y
0	$\top$	$\top$	$\top$	$\top$		
1	10	$\top$	10	$\top$		
2	10	1	$\top$	$\top$		
3	10	1	$\top$	$\top$		
4	10	10	$\top$	$\top$	ditto	
5	9	10	$\top$	$\top$		
6	$\bot$		$\top$	$\top$		
7	$\bot$		$\top$	$\top$		

Fig. 1.23 Constant propagation for the factorial program

where $D_\top\, x = \top$ for all $x \in Vars$ holds. Invariant (K) implies for all initial states s and all paths π reaching program point v:

$$([\![\pi]\!]\, s) \;\; \Delta \;\; (\mathcal{D}^*[v])$$

Solving the associated system of inequalities leads to an approximation of the MOP solution.

Example 1.9.4 Consider the factorial program, this time with a given initial value for variable x. Figure 1.23 shows the result of the analysis. We know that, with a given initial value, the whole computation of the factorial of that value could be executed at compile time. Our static analysis on the other hand, does *not* identify this possibility. The reason is that constant propagation determines values of variables at program points that are the same each time execution reaches that program point. The values of the variables x and y, however, change within the loop. □

In conclusion, we note that constant propagation computes with concrete values as far as they can be determined statically. Expressions consisting of only known values can be evaluated by the compiler. In general, though, constant propagation will only be able to determine a subset of the concrete variable bindings. The fixed-point iteration to determine the least solution of the system of inequalities always terminates. With n program points and m variables it takes at most $\mathcal{O}(m \cdot n)$ rounds. Example 1.9.4 shows that the iteration often terminates faster. There is one caveat, though: the edge effects for constant propoagation are not all distributive. As a counterexample consider the abstract edge effect for the assignment $x \leftarrow x + y$ together with the two variable bindings:

$$D_1 = \{x \mapsto 2, y \mapsto 3\} \quad \text{and} \quad D_2 = \{x \mapsto 3, y \mapsto 2\}$$

On the one hand, we have:

$$\begin{aligned}[\![x \leftarrow x + y]\!]^\sharp D_1 \sqcup [\![x \leftarrow x + y]\!]^\sharp D_2 &= \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\} \\ &= \{x \mapsto 5, y \mapsto \top\}\end{aligned}$$

On the other hand, it holds that:

$$\begin{aligned}[\![x \leftarrow x + y]\!]^\sharp (D_1 \sqcup D_2) &= [\![x \leftarrow x + y]\!]^\sharp \{x \mapsto \top, y \mapsto \top\} \\ &= \{x \mapsto \top, y \mapsto \top\}\end{aligned}$$

Therefore

$$[\![x \leftarrow x + y]\!]^\sharp D_1 \sqcup [\![x \leftarrow x + y]\!]^\sharp D_2 \;\neq\; [\![x \leftarrow x + y]\!]^\sharp (D_1 \sqcup D_2)$$

violating the distributivity property.

The least solution $\mathcal{D}$ of the system of inequalities thus in general delivers only an *upper approximation* of the MOP solution. This means:

$$\mathcal{D}^*[v] \;\sqsubseteq\; \mathcal{D}[v]$$

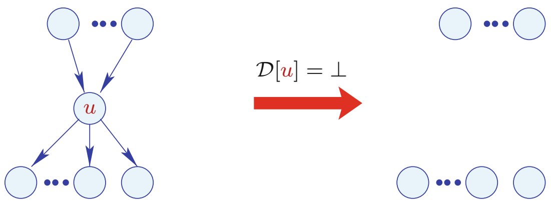
for each program point v. Being an upper approximation, $\mathcal{D}[v]$ still describes the result of each computation along a path π that ends in v:

$$([\![\pi]\!](\rho, \mu))\,\Delta\,\mathcal{D}[v],$$

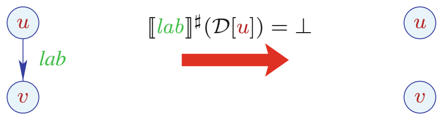
whenever $[\![\pi]\!]\,(\rho, \mu)$ is defined. Therefore, the least solution is *safe* information, which the compiler can use to check the applicability of program transformations.

Transformation CF:

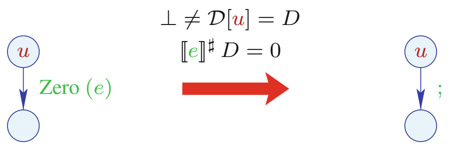
The first use of information $\mathcal{D}$ consists in removing program points that are identified as unreachable. The following transformation performs the removal of *dead code*:

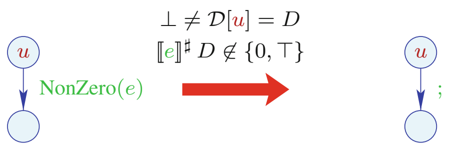


Furthermore, the compiler may remove all condition edges that might lead to a reachable node, but whose abstract edge effect delivers $\bot$:

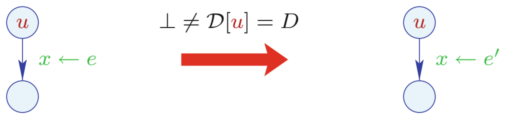


The next two rules simplify condition edges whose conditions deliver a definite, i.e., non-$\top$ value. Having a definite value means that this edge will be taken in all executions.




u
NonZero(e)
$\bot \neq \mathcal{D}[u] = D$
$[\![e]\!]^\sharp\ D \notin \{0, \top\}$
u
;

Finally, the compiler uses the information $\mathcal{D}$ to evaluate program expressions at compile time whenever this is shown to be possible. For assignments, we obtain:



where the expression e' results from evaluating the expression e in the abstract variable binding D .

$$e' = \begin{cases} c & \text{if } [\![e]\!]^\sharp\ D = c \neq \top \\ e & \text{if } [\![e]\!]^\sharp\ D = \top \end{cases}$$

The simplification of expressions at other edges works similarly.

Constant folding as explained so far is always applied to maximal expressions in statements. It can also be extended to subexpressions:

$$x + (3 \cdot y) \quad \xRightarrow{\{x \mapsto \top, y \mapsto 5\}} \quad x + 15$$

$$y \cdot (x + 3) \quad \xRightarrow{\{x \mapsto \top, y \mapsto 5\}} \quad 5 \cdot (x + 3)$$

Our analysis can be improved to better exploit the information contained in conditions.

Example 1.9.5 Consider the following example program:

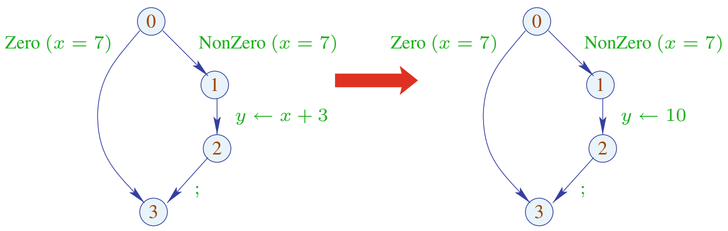


Fig. 1.24 Exploiting the information in conditions

$$\begin{array}{l} \textbf{if } (x = 7) \\ \qquad y \leftarrow x + 3; \end{array}$$

Without knowing the value of x before the *if* statement, the analysis can derive that x has the value 7 when control enters the *then* part. □

Conditions testing the equality of variables with values can be exploited particularly well.

$$[\![\mathsf{NonZero}\,(x = e)]\!]^\sharp\, D = \begin{cases} \bot & \text{if } [\![x = e]\!]^\sharp\, D = 0 \\ D_1 & \text{otherwise} \end{cases}$$

where we define:

$$D_1 = D \oplus \{x \mapsto (D\,x \;\sqcap\; [\![e]\!]^\sharp\, D)\}$$

We can choose an analogous abstract edge effect for $\mathsf{Zero}\,(x \neq e)$.

Figure 1.24 shows the improvement that the compiler achieves for Example 1.9.5.

1.10 Interval Analysis

Constant propagation attempts for each program point v to determine the values of variables that the variables have every time execution reaches v. Often, a variable has different values at a program point v when execution reaches v several times. Interval analysis makes the best of this situation by computing an interval *enclosing* all possible values that the variable may have when execution reaches v.

Example 1.10.1 Consider the following program:

$$\textbf{for } (i \leftarrow 0;\, i < 42;\, i{+}{+}) \quad a[i] = i;$$

Programming languages such as JAVA require that array indices always lie within the declared bounds of the array. Let the *int* array a begin at address A, and let it have the bounds 0 and 41. The code generated for the program above can look as follows:

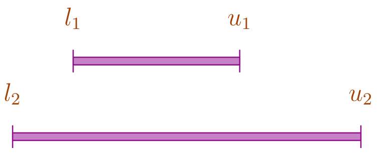


Fig. 1.25 The order on intervals $[l_1, u_1] \sqsubseteq [l_2, u_2]$

```
    $i \leftarrow 0;$
$B$ : **if** $(i < 42)$ {
        **if** $(0 \leq i \wedge i < 42)$ {
            $A_1 \leftarrow A + i;$
            $M[A_1] \leftarrow i;$
            $i \leftarrow i + 1;$
        } **else goto** error;
        **goto** $B$;
    }
```

The condition of the outer loop makes the inner bounds check superfluous. It will never trigger the jump to program point error. The inner bounds check can therefore be eliminated. □

Interval analysis generalizes constant propagation by replacing the domain $\mathbb{Z}^\top$ for the values of variables by a domain of intervals. The set of all intervals is given by:

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

l stands for *lower* and u for *upper*. According to this definition, each interval represents a *nonempty* set of integers. There exists a natural order on intervals: $\sqsubseteq$:

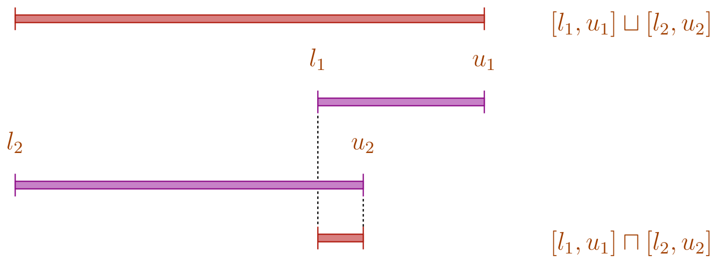
$$[l_1, u_1] \sqsubseteq [l_2, u_2] \quad \text{iff} \quad l_2 \leq l_1 \wedge u_1 \leq u_2$$

Figure 1.25 represents the geometric intuition behind this definition.

The least upper bound and the greatest lower bounds on intervals are defined as follows:

$$\begin{array}{l} [l_1, u_1] \sqcup [l_2, u_2] = [\mathsf{min}\{l_1, l_2\}, \mathsf{max}\{u_1, u_2\}] \\ [l_1, u_1] \sqcap [l_2, u_2] = [\mathsf{max}\{l_1, l_2\}, \mathsf{min}\{u_1, u_2\}], \text{ sofern } \mathsf{max}\{l_1, l_2\} \leq \mathsf{min}\{u_1, u_2\} \end{array}$$

The geometric intuition for these operations is illustrated in Fig. 1.26. The least upper bound is depicted on top of the two given intervals; the greatest lower bound below them. Like $\mathbb{Z}^\top$, the set $\mathbb{I}$ together with the partial order $\sqsubseteq$ is a partial order, but *not* a complete lattice. It has no least element since the empty set is explicitly excluded. Least upper bounds therefore only exist for *nonempty* sets of intervals. Also, the greatest lower bound operation is is only defined, if intervals overlap. There is one



$[l_1, u_1] \sqcup [l_2, u_2]$
l_1
u_1
l_2
u_2
$[l_1, u_1] \sqcap [l_2, u_2]$

Fig. 1.26 Two intervals, in the *middle*, and their least upper bound, on *top*, and their greatest lower bound, on the *bottom*

important difference, though, between the partial order $\mathbb{Z}^\top$ and the partial order $\mathbb{I}$. The partial order $\mathbb{Z}^\top$ has only *finite* strictly ascending chains while $\mathbb{I}$ has ascending chains that never stabilize, for instance, the following:

$$[0, 0] \sqsubseteq [0, 1] \sqsubseteq [-1, 1] \sqsubseteq [-1, 2] \sqsubseteq \dots$$

The natural description relation Δ between integer values and integer intervals is given by:

$$z \;\Delta\; [l, u] \qquad \text{iff} \qquad l \leq z \leq u$$

This description relation leads to the following concretization function:

$$\gamma\,[l, u] = \{z \in \mathbb{Z} \mid l \leq z \leq u\}$$

Example 1.10.2 We have:

$$\gamma\,[0, 7] = \{0, \dots, 7\}$$
$$\gamma\,[0, \infty] = \{0, 1, 2, \dots\}$$

□

Interval analysis needs to calculate with intervals. These calculations are expressed in terms of abstract versions of the arithmetic, Boolean and comparison operators. The sum of two intervals should contain all values that result when any two values from the argument intervals are added. We therefore define:

$$\begin{array}{rcl} [l_1, u_1] \;+^\sharp\; [l_2, u_2] & = & [l_1 + l_2, u_1 + u_2] \qquad \text{where} \\ -\infty + \_ & = & -\infty \\ +\infty + \_ & = & +\infty \end{array}$$

Note that the value of $-\infty + \infty$ never need to be computed.

Negation on intervals is defined as:

$$-^\sharp [l, u] = [-u, -l]$$

To define multiplication on intervals is more difficult. The smallest interval must be determined that contains all products of values taken from two argument intervals. A rather simple definition that saves many case distinctions is the following:

$$\begin{array}{ll} [l_1, u_1] \cdot^\sharp [l_2, u_2] = [a, b] & \text{where} \\ & a = \min\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\} \\ & b = \max\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\} \end{array}$$

Example 1.10.3 We check the plausibility of this definition of interval multiplication by inspecting a few examples.

$$\begin{aligned} [0, 2] \cdot^\sharp [3, 4] &= [0, 8] \\ [-1, 2] \cdot^\sharp [3, 4] &= [-4, 8] \\ [-1, 2] \cdot^\sharp [-3, 4] &= [-6, 8] \\ [-1, 2] \cdot^\sharp [-4, -3] &= [-8, 4] \end{aligned}$$

□

To define division on intervals is really problematic! Let $[l_1, u_1] /^\sharp [l_2, u_2] = [a, b]$.

- If 0 is not contained in the denominator interval we can define:

$$\begin{aligned} a &= \min\{l_1/l_2, l_1/u_2, u_1/l_2, u_1/u_2\} \\ b &= \max\{l_1/l_2, l_1/u_2, u_1/l_2, u_1/u_2\} \end{aligned}$$

- However, if 0 is contained in the denominator interval, that is: $l_2 \leq 0 \leq u_2$, a run-time error cannot be excluded. The semantics of our example language does not state what happens in the case of such a run-time error. We assume for simplicity that *any* value is a legal result. We therefore define for this case:

$$[a, b] = [-\infty, +\infty]$$

Besides abstract versions of the arithmetic operators, we need abstract versions of comparison operators. The abstract version of the comparison for equality is quite different from the "natural" equality of intervals. Abstract comparisons of intervals can have the values *true, false*, or *unknown* Boolean value which describes both *true* and *false*. According to the semantics of our example language, the value *false* is represented by 0, while the value *true* (when returned by a Boolean operator) should be represented by 1. The corresponding intervals are [0, 0] and [1, 1]. Accordingly, the unknown Boolean value *true* $\sqcup$ *false* is represented by the interval $[0, 0] \sqcup [1, 1] = [0, 1]$.

The value *true* results for two identical singleton intervals. The result must be *false* for two disjoint intervals because the comparison can never deliver *true* for any pair of values from the argument intervals. *true*$\sqcup$*false* must be the result in the case of nondisjoint, nonsingleton intervals because there are pairs of identical values in the argument intervals, but also pairs of nonidentical values.

$$[l_1, u_1] =^\sharp [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if } u_1 < l_2 \vee u_2 < l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Example 1.10.4 We use some examples to convince ourselves that this definition makes sense:

$$\begin{aligned} [42, 42] =^\sharp [42, 42] &= [1, 1] \\ [1, 2] =^\sharp [3, 4] &= [0, 0] \\ [0, 7] =^\sharp [0, 7] &= [0, 1] \end{aligned}$$

□

We now treat just one more comparison operator, namely the operation $<$. We have:

$$[l_1, u_1] \;<^\sharp\; [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{if } u_1 < l_2 \\ [0,0] & \text{if } u_2 \leq l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Example 1.10.5 Some example calculations illustrate the abstract comparison operator.

$$\begin{aligned} [1, 2] <^\sharp [9, 42] &= [1, 1] \\ [0, 7] <^\sharp [0, 7] &= [0, 1] \\ [3, 4] <^\sharp [1, 3] &= [0, 0] \end{aligned}$$

□

Starting with the partial order $(\mathbb{I}, \sqsubseteq)$, we construct a complete lattice for abstract variable bindings. This procedure is analogous to the construction of the complete lattice for constant propagation.

$$\mathbb{D}_{\mathbb{I}} = (\mathit{Vars} \to \mathbb{I})_\perp = (\mathit{Vars} \to \mathbb{I}) \cup \{\perp\}$$

for a new element $\perp$, which is the least element and again denotes unreachability. We define a description relation between concrete and abstract variable bindings in the natural way by:

$$\rho \;\Delta\; D \qquad \text{iff} \qquad D \neq \bot \quad \wedge \quad \forall x \in \mathit{Vars} : \; (\rho\, x) \;\Delta\; (D\, x).$$

This leads to a corresponding description relation Δ between concrete states (ρ, μ) and abstract variable bindings:

$$(\rho, \mu) \;\Delta\; D \qquad \text{iff} \qquad \rho \;\Delta\; D$$

The abstract evaluation of expressions is also defined in analogy to the abstract evaluation for constant propagation. It holds for all expressions:

$$([\![e]\!]\, \rho) \;\;\Delta\;\; ([\![e]\!]^\sharp\, D) \qquad \text{if} \qquad \rho \;\;\Delta\;\; D$$

Next to define are the abstract edge effects for interval analysis. They also look quite like the ones for constant propagation, apart from the fact that they now calculate over interval domains:

$$\begin{aligned}
[\![;]\!]^\sharp\, D &= D \\
[\![x \leftarrow e]\!]^\sharp\, D &= D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x \leftarrow M[e]]\!]^\sharp\, D &= D \oplus \{x \mapsto \top\} \\
[\![M[e_1] \leftarrow e_2]\!]^\sharp\, D &= D \\
[\![\mathsf{NonZero}\,(e)]\!]^\sharp\, D &= \begin{cases} \bot & \text{if } [0,0] = [\![e]\!]^\sharp\, D \\ D & \text{otherwise} \end{cases} \\
[\![\mathsf{Zero}\,(e)]\!]^\sharp\, D &= \begin{cases} \bot & \text{if } [0,0] \not\sqsubseteq [\![e]\!]^\sharp\, D \\ D & \text{if } [0,0] \sqsubseteq [\![e]\!]^\sharp\, D \end{cases}
\end{aligned}$$

if $D \neq \bot$. Here, $\top$ denotes the interval $[-\infty, \infty]$.

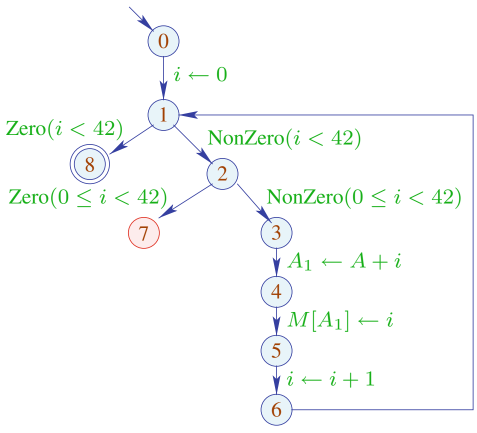
We assume, like in the case of constant propagation, that nothing is known about the values of variables at the entry to the program. This is expressed by associating the largest lattice element, $\top = \{x \mapsto [-\infty, \infty] \mid x \in \mathit{Vars}\}$, with program entry.

For the proof of correctness of interval analysis we formulate an invariant that is very similar to the invariant (K) of constant propagation, the only difference being that all computations are on intervals instead of on $\mathbb{Z}^\top$. The proof uses the same argumentation so that we omit it here.

Conditions are an essential source of information for interval analysis, even more so than they are for constant propagation. Comparisons of variables with constants can be very fruitfully exploited. Let us assume that e is of the form $x \,\Box\, e_1$ for comparison operators $\Box \in \{=, <, >\}$. We define:

$$[\![\mathsf{NonZero}\,(e)]\!]^\sharp\, D = \begin{cases} \bot & \text{if} \quad [0,0] = [\![e]\!]^\sharp\, D \\ D_1 & \text{otherwise} \end{cases}$$

where



	i	
	l	u
0	$-\infty$	$+\infty$
1	0	42
2	0	41
3	0	41
4	0	41
5	0	41
6	1	42
7	$\perp$	
8	42	42

Fig. 1.27 The least solution of the interval analysis of Example 1.10.1

$$D_1 = \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\, D)\} & \text{if } e \equiv (x = e_1) \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u-1]\} & \text{if } e \equiv (x < e_1),\ [\![e_1]\!]^\sharp\, D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l+1, \infty]\} & \text{if } e \equiv (x > e_1),\ [\![e_1]\!]^\sharp\, D = [l, \_] \end{cases}$$

A condition NonZero$(x < e_1)$ allows "cutting" the interval $[u, \infty]$ from the interval for x where u is the largest possible value in the interval for e_1. We define correspondingly:

$$[\![\mathsf{Zero}\,(e)]\!]^\sharp\, D = \begin{cases} \perp & \text{if} \quad [0,0] \not\sqsubseteq [\![e]\!]^\sharp\, D \\ D_1 & \text{otherwise} \end{cases}$$

where

$$D_1 = \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{if } e \equiv (x > e_1),\ [\![e_1]\!]^\sharp\, D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{if } e \equiv (x < e_1),\ [\![e_1]\!]^\sharp\, D = [l, \_] \\ D & \text{if } e \equiv (x = e_1) \end{cases}$$

Note that greatest lower bounds of intervals are used here. These greatest lower bounds are defined in this context, because otherwise the abstract evaluation of the condition would have returned an interval not subsuming $[0, 0]$.

Let us regard the program of Example 1.10.1. Its control-flow graph and the least solution of the system of inequalities for the interval analysis of variable i are shown in Fig. 1.27.

The partial order $\mathbb{I}$ has ascending chains that never stabilize. It is therefore not clear how to determine the least solution of the system of inequalities for interval analysis. In our example, fixed-point iteration would terminate, but only after 43 rounds. Other programs, though, can be constructed where round-robin iteration for interval analysis would not terminate.

Apparently, we need new techniques to deal with complete lattices that have infinite ascending chains. The inventors of abstract interpretation, Patrick and Radhia Cousot, also invented the necessary techniques, *widening* and *narrowing*. Their first publication presented interval example with widening as an example.

The idea of *widening* is to *speed* fixed-point iteration, albeit at the cost of a possibly reduced precision. The measure to speed up the iteration guarantees that each abstract value of an unknown can only undergo *finitely* many changes.

One idea to widening for interval analysis is not to allow *arbitrary* enlargements of intervals. No enlargements from *finite to finite* intervals are admitted. An admissible ascending chain of intervals could look like the following:

$$[3, 17] \sqsubseteq [3, +\infty] \sqsubseteq [-\infty, +\infty]$$

Let us formalize the general approach of widening. Let

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

be again a system of inequalities over a complete lattice $\mathbb{D}$. We consider the *accumulating* system of equations associated with this system of inequalities:

$$x_i \;=\; x_i \sqcup f_i\,(x_1, \ldots, x_n)\;, \quad i = 1, \ldots, n$$

A tuple $\underline{x} = (x_1, \ldots, x_n) \in \mathbb{D}^n$ is a solution of the system of inequalities if and only if it is a solution of the associated accumulating system of equalities. The reformulation of the system of inequalities as an accumulating system of equations alone does not solve our problem. Fixed-point iteration for the accumulating system as for the original system may not necessarily terminate. Therefore, we replace the operator $\sqcup$ of the accumulating system by a *widening* operator, $\sqcup\!\!\!\!\!-$ which can be used to enforce termination. As a result, we obtain the system of equations:

$$x_i = x_i \sqcup\!\!\!\!\!- f_i(x_1, \ldots, x_n)\;, \quad i = 1, \ldots, n$$

The new operator $\sqcup\!\!\!\!\!-$ must satisfy:

$$v_1 \sqcup v_2 \;\sqsubseteq v_1 \sqcup\!\!\!\!\!- v_2$$

The values accumulated for an unknown x_i during a fixed-point iteration for the system with widening therefore will grow at least as fast as the values for the fixed-point iteration for the system without widening. Round-robin iteration for the modified system, if it terminates, still computes a solution of the accumulating system of equations and therefore also for the original system of inequalities.

We apply the general method of widening to interval analysis and the complete lattice $\mathbb{D}_{\mathbb{I}} = (\mathit{Vars} \to \mathbb{I})_\perp$. A widening operator $\sqcup\!\!\!\!\!-$ for this complete lattice is defined by:

$$\perp \sqcup\!\!\!\!\!- D \;=\; D \sqcup\!\!\!\!\!- \perp \;=\; D$$

and for $D_1 \neq \bot \neq D_2$

$$\begin{array}{rcll} (D_1 \sqcup\!\!\!\!\sqcup D_2)\, x & = & (D_1\, x) \sqcup\!\!\!\!\sqcup (D_2\, x) & \text{where} \\ [l_1, u_1] \sqcup\!\!\!\!\sqcup [l_2, u_2] & = & [l, u] & \text{such that} \\ l & = & \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases} \\ u & = & \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases} \end{array}$$

The widening operator for variable bindings is based on a widening operator for intervals. During fixed-point iteration, the left operand operand is the old value while the right operand is the new value. Therefore, the operator treats its two arguments differently and is, thus, not commutative.

Example 1.10.6 Here are some iteration steps:

$$\begin{array}{l} [0, 2] \sqcup\!\!\!\!\sqcup [1, 2] = [0, 2] \\ [1, 2] \sqcup\!\!\!\!\sqcup [0, 2] = [-\infty, 2] \\ [1, 5] \sqcup\!\!\!\!\sqcup [3, 7] = [1, +\infty] \end{array}$$

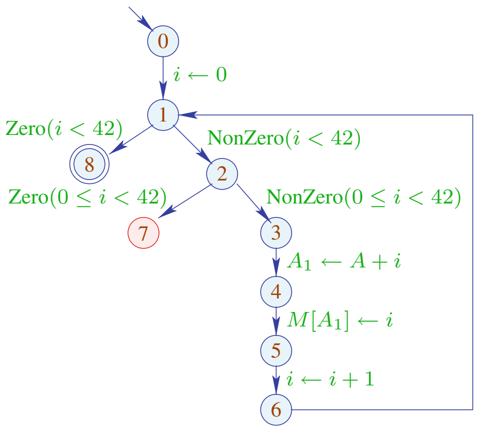
□

The widening operator, in general, does not deliver the least upper bound, but only *some* upper bound. The values of the unknowns therefore may grow faster. A practical widening operator should be chosen in a way that guarantees that the resulting ascending chains eventually stabilize so that fixed-point iteration terminates. The widening operator that we have presented, guarantees that each interval can grow at most two times. Therefore, the number of iteration steps for round-robin iteration for a program with n program points to $\mathcal{O}(n \cdot \#\mathit{Vars})$.

In general, the starting point is a complete lattice with infinite ascending chains together with a system of inequalities over this lattice. In order to determine some (hopefully nontrivial) solution for this system, we first rewrite it into an equivalent accumulating system of equations. Then we replace the least-upper bound operation of the accumulation with a widening operator. This operator speeds up iteration and enforces termination of the fixed-point iteration by admitting only a finite number of changes to the values of the unknowns.

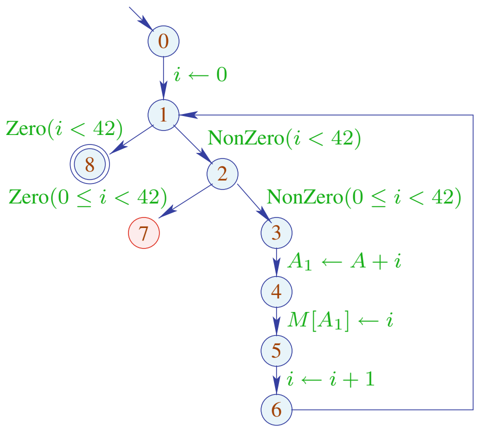
The design of such widening operators is black magic. On the one side, the widening operator needs to radically lose information in order to guarantee termination. On the other hand, it should keep enough relevant information such that the results of the analysis still have some value. Figure 1.28 shows round-robin iteration for the program of Example 1.10.1. The iteration terminates rather quickly as we have hoped, but with a disappointing result. The analysis loses all knowledge of upper bounds. An elimination of the index out of bounds check is not possible.

Apparently, information is thrown away too generously. We therefore need to improve on this naive procedure. Some thinking reveals that the widening operator should be applied more economically. It is not necessary to apply it for each unknown



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	$+\infty$		
2	0	0	0	$+\infty$		
3	0	0	0	$+\infty$		
4	0	0	0	$+\infty$	ditto	
5	0	0	0	$+\infty$		
6	1	1	1	$+\infty$		
7	$\perp$		42	$+\infty$		
8	$\perp$		42	$+\infty$		

Fig. 1.28 Accelerated round-robin iteration for Example 1.10.1



$I_1 = \{1\}$ or
$I_2 = \{2\}$

Fig. 1.29 Feedback vertex set for the control-flow graph of Example 1.10.1

at each program point and still guarantee termination. It suffices to apply widening at least once in each cycle of the control-flow graph.

A set I of nodes in a directed graph G is called *feedback vertex set* if it contains at least one node of each directed cycle in G. Round-robin iteration still terminates if widening is applied only at the nodes of a feedback vertex set of the control-flow graph.

Example 1.10.7 This idea is tried out at our program from Example 1.10.1. Figure 1.29 shows example sets I_1 and I_2 of nodes each of which contain one node in the (unique) directed cycle of the program. For widening placed at node 1, round-robin iteration yields:

	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	$+\infty$		
2	0	0	0	41		
3	0	0	0	41		
4	0	0	0	41	ditto	
5	0	0	0	41		
6	1	1	1	42		
7	$\perp$		$\perp$			
8	$\perp$		42	$+\infty$		

In fact, it is almost the least solution that is obtained. The only information lost is the upper bound for loop variable i at program points 1 and 8.

For widening placed at the node 2, we obtain:

	1		2		3		4	
	l	u	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	1	0	42		
2	0	0	0	$+\infty$	0	$+\infty$		
3	0	0	0	41	0	41		
4	0	0	0	41	0	41	ditto	
5	0	0	0	41	0	41		
6	1	1	1	42	1	42		
7	$\perp$		42	$+\infty$	42	$+\infty$		
8	$\perp$		$\perp$		42	42		

The analysis using this feedback vertex set obtains better information about variable i at program points 1 and 8, but loses so much information at program point 2 that it can no longer derive the nonreachability of program point 7. □

This example shows that the restriction of widening to some relevant program points may improve the precision of the analysis considerably. The example also shows that is not always clear where to apply widening to obtain the most precise information. A complementary technique is now presented, *narrowing*.

Narrowing is a technique to gradually improve a possibly too imprecise solution. As for widening, we first develop the general approach for arbitrary systems of inequalities and then turn to how narrowing can be applied to interval analysis.

Let $\underline{x}$ be some solution to the system of inequalities

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

Let us assume further that the right-hand sides f_i are all monotonic and that F is the associated function $\mathbb{D}^n \to \mathbb{D}^n$. The monotonicity of F implies:

$$\underline{x} \sqsupseteq F\,\underline{x} \sqsupseteq F^2\,\underline{x} \sqsupseteq \ldots \sqsupseteq F^k\,\underline{x} \sqsupseteq \ldots$$

This iteration is called *narrowing*. Narrowing has the property that all tuples $F^i\,\underline{x}$ that are obtained after some iteration steps are solutions to the system of inequalities. This also holds for narrowing by round-robin iteration. Termination is not a problem any more: iteration can be stopped whenever the obtained information is good enough.

Example 1.10.8 Consider again the program of Example 1.10.1 where narrowing is applied to the result produced by naive widening. We obtain:

	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\bot$		$\bot$	
8	42	$+\infty$	42	$+\infty$	42	42

In fact, the optimal solution is obtained! □

In our example, the narrowing, following the widening, completely compensates for the widening's loss of information. This can not always be expected. It is also possible that narrowing needs a long time. It even may not terminate, namely if the lattice has infinite *descending chains*. This is the case for the interval lattice. Termination, though, can be enforced by *accelerated narrowing*. Let us assume that we are given some solution of the system of inequalities

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

We consider the following system of equations:

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

We start with a possibly too large solution. To improve, that is to shrink, the values for the unknowns, the contributions of the right sides are used.

Let $H : \mathbb{D}^n \to \mathbb{D}^n$ be the function defined by $H\,(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ such that $y_i = x_i \sqcap f_i\,(x_1, \ldots, x_n)$. If all f_i are monotonic we have:

$$H^i\,\underline{x} = F^i\,\underline{x} \qquad \text{for all } i \geq 0\,.$$

Now the operator $\sqcap$ in the system of equations is replaced by a new operator $\mathbin{\bar{\sqcap}}$, which possesses the following property:

$$a_1 \sqcap a_2 \;\sqsubseteq\; a_1 \mathbin{\bar{\sqcap}} a_2 \;\sqsubseteq\; a_1$$

We call the new operator the *narrowing* operator. The new operator does not necessarily reduce the values as quickly as the greatest-lower-bound operator, but at least returns values which are less or equal to the old values.

In the case of the interval analysis, a narrowing operator is obtained by allowing interval bounds only to be improved by replacing infinite bounds with finite bounds. This way, each interval can at most be improved at most twice. For variable bindings D we define:

$$\bot \mathbin{\bar{\sqcap}} D \;=\; D \mathbin{\bar{\sqcap}} \bot \;=\; \bot$$

and for $D_1 \neq \bot \neq D_2$

$$\begin{array}{rcl}
(D_1 \mathbin{\bar{\sqcap}} D_2)\,x &=& (D_1\,x) \mathbin{\bar{\sqcap}} (D_2\,x) \qquad \text{where} \\
{[l_1, u_1]} \mathbin{\bar{\sqcap}} [l_2, u_2] &=& [l, u] \qquad \text{where} \\
l &=& \begin{cases} l_2 & \text{if} \;\; l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases} \\
u &=& \begin{cases} u_2 & \text{if} \;\; u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}
\end{array}$$

In the applications of the narrowing operator, the left operand is the value of the last iteration step, while the right operand is the newly computed value. Therefore, the narrowing operator does not treat both its operands in the same way and thus is not necessarily commutative.

Example 1.10.9 Let us apply the accelerated narrowing with round-robin iteration to the program of Example 1.10.1. We obtain:

	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\bot$		$\bot$	
8	42	$+\infty$	42	$+\infty$	42	42

We observe that no information is lost despite the application of accelerated narrowing. □

Widening, in principle, also works for nonmonotonic right sides in the system of inequalities. However, narrowing requires monotonicity. Accelerated narrowing is guaranteed to terminate if the narrowing operator admits only descending chains of bounded length. In the case of interval analysis, our operator $\sqcap\!\!\!\!-$ is defined in a way that each interval would be modified at most twice. This means that round-robin iteration using this narrowing operator takes at most $\mathcal{O}(n \cdot \#Vars)$ rounds, where n is the number of program points.

1.11 Alias Analysis

The analyses and transformations presented so far were concerned with variables. The memory component M of our programming language was considered as one large statically allocated array. This view is sufficient for analysis problems that deal with variables and expressions only. Many programming languages, however, offer dynamic allocation of anonymous objects and constructs to indirectly access anonymous data objects through pointers (references). This section treats analyses to deal with pointers and dynamically allocated memory. Therefore, we extend our programming language by *pointers*, which point to the beginning of dynamically allocated blocks of memory. We use small letters for *int* variables to distinguish them from pointer variables, for which we use capital letters. The generic name z can denote both *int* variables and pointer variables. There is one pointer constant, null. As new statements in our language we introduce:

- A statement $R \leftarrow \mathsf{new}(e)$ for an expression e and a pointer variable R. The operator new() allocates a new block in memory and returns in z a pointer to the beginning of this block. The size of this block is given by the value of the expression e.
- A statement $z \leftarrow R[e]$ for a pointer variable R, an expression e, and a variable z. The value of e is used as an index into the block to which R points and selects one cell in this block. This index is assumed to be within the range of 0 and the size of the block. The value in the indexed cell then is assigned to z.
- A statement $R[e_1] \leftarrow e_2$ with a pointer variable R and expressions e_1 and e_2. Expression e_2's value is stored in the cell whose index is the value of e_1 in the block pointed to by R. Again, the index is assumed to lie within the range of 0 and the size of the block pointed at by R.

We do not allow *pointer arithmetic*, that is, arithmetic operations on pointer values. We also do not allow pointers to variables. To keep the presentation simple, we do not introduce a type system that would distinguish *int* variables from pointer variables. We just assume that, during runtime, *int* variables will only hold *int* values and pointer variables only pointer values, and that for indexing and in arithmetic operations only *int* values are used.

Pointer variables R_1 and R_2 are *aliases* of each other in some state if they have the same value, that is, point to the same block in memory in the given state. We also say that R_1 is *an alias for* R_2 and vice versa. An important question about programs written in a language with dynamic memory allocation is whether two pointer variables *possibly* have the same value at some program point, that is, whether the program may be in a state in which the two pointers are aliases. This problem is called the *may-alias problem*. Another question is whether two pointer variables *always* have the same value at a program point. This problem correspondingly is called the *must-alias problem*.

Use of Alias Information

Here is an example of the use of alias information. The compiler may want to optimize the statement $x \leftarrow R[0] + 2$ in the following code fragment:

$$\begin{aligned}&R[0] \leftarrow 0;\\&S[0] \leftarrow 1;\\&x \leftarrow R[0] + 2;\end{aligned}$$

There are three different cases:

- Program analysis has found out that S and R cannot be aliases. In this case it may transform the assignment to x into $x \leftarrow 2$.
- It has found out that S and R are must aliases. The compiler can transform the assignment into $x \leftarrow 3$.
- It is unknown whether S and R are aliases. In this case, the compiler cannot do any optimization.

The most important use of may-alias information is in *dependence analysis*. This analysis determines information about the flow of values from definitions to uses also in presence of dynamically allocated memory. Several optimizations attempt to improve the efficiency of programs by reordering the statements of the program. One such optimization, performed on the machine program by the compiler back-end, is *instruction scheduling*, which tries to exploit the parallel processing capabilities of modern processors. Reordering the statements or instructions of a program must not change its semantics. A sufficient condition for semantics preservation in reordering transformations is that the flow of values from definitions to uses is not changed.

Dependence analysis determines several types of dependences:
True dependence: A use of a resource, e.g., a variable or a memory cell, follows a definition of the same resource without an intervening redefinition.
Output dependence: A definition of a resource follows another definition of the same resource without intervening use or definition of that resource.
Antidependence: A definition of a resource follows a use of the same resource without an intervening redefinition.

Any reordering of statements changing such dependences would be forbidden. In a language with pointers, writing accesses to resources (definitions) and reading accesses to resources (uses) can be performed indirectly through pointers. May-alias information can then be interpreted as, “could be the same resource” in the above definitions of dependences.

Must-alias information allows optimizations exploit the information that the access through a pointer R goes to a particular memory block. Must-alias information for pointer variables R and R' can then be used to infer that the access through R goes to the same memory block as through R'. If we additionally know that the corresponding index expressions are equal, we can infer that the accesses even go to the identical memory cell. An extension of redundancy elimination to memory operations is considered in Exercise 27.

Background: Over- and Underapproximations

There is an interesting observation we can make about may- and must-alias analysis. Both analyses attempt to determine whether there exist memory cells to which two pointers point. May-alias analysis computes an *overapproximation*, that is, a *superset* of the set of existing alias relationships. This means it detects all cases of aliases that happen in some execution, but it may also report some aliases that never occur. A *safe* use of this information is the use of its complement. If two pointers are not in the may-alias set they will never be aliased. Must-alias analysis, on the other hand, computes an underapproximation, that is, a *subset* of the set of actually occurring alias relationships. It will only report aliases at a program point that definitely exist every time execution reaches this program point. It may, however, miss some aliases occurring during program execution.

Now that we have met these two notions, we can also try to classify the analyses we have met so far as either over- or underapproximations. Available-assignments analysis computes an underapproximation of the set of assignments that are available along all program execution paths. Live-variable analysis computes an overapproximation of the set of variables whose values are used later on. Constant propagation, on the other hand, again determines an underapproximation of the set of invariant bindings of variables to constants. Interval analysis computes an overapproximation of the set of values a variable may have.

Formally, however, the abstract domains and their partial orders, denoted by $\sqsubseteq$, are arranged in such a way that our analyses always compute *overapproximations*. So, in the example of may-alias analysis, the partial order $\sqsubseteq$ of the lattice is the subset relation, $\subseteq$, and the lattice element $\top$, which represents *no information*, is the set of *all* alias relationships. In the case of the must-alias analysis, $\sqsubseteq$ is the superset relation, $\supseteq$, and $\top$ is the empty set of alias relationships.

Some Programs Using Pointers

Example 1.11.1 A first example of a program using pointers is shown in Fig. 1.30.

```
$X \leftarrow \mathsf{new}(2);$
$Y \leftarrow \mathsf{new}(2);$
$X[0] \leftarrow Y;$
$Y[1] \leftarrow 7;$
```

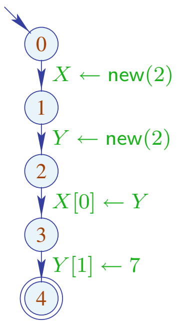


Fig. 1.30 A simple pointer-manipulating program and its control-flow graph

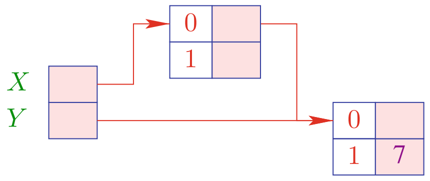


Fig. 1.31 Program state after the execution of the program of Fig. 1.30

```
        $R \leftarrow \mathsf{null};$
$A$ :   **if** $(T \neq \mathsf{null})$ {
            $H \leftarrow T;$
            $T \leftarrow T[0];$
            $H[0] \leftarrow R;$
            $R \leftarrow H;$
            **goto** $A;$
        }
```

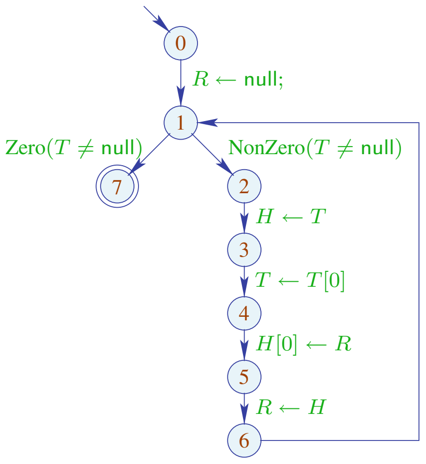


Fig. 1.32 A program that reverses a list

The program allocates two blocks. A pointer to the second block is stored at address 0 of the first block. Value 7 is stored at address 1 of the second block. Figure 1.31 shows the state of memory after the execution of this program. □

Example 1.11.2 A somewhat more complex example is the program in Fig. 1.32, which reverses a list pointed to by pointer variable T.

Although this program is short, it is by no means easy to convince oneself of its correctness. It demonstrates that even short programs doing nontrivial pointer manipulations are hard to understand and are prone to subtle errors. □

Extension of the Operational Semantics

We modify the operational semantics of our programming language to serve as the basis for the definition of a may-alias analysis. The memory component is no longer a single potentially infinite array of memory cells, but a potentially infinite array of blocks, each consisting of an array of memory cells.[1] Each execution of the statement $\mathsf{new}()$ makes available a new block. The size of these blocks is only known when the program is executed. Each block consists of as many memory cells as are indicated in the $\mathsf{new}()$ statement, where we assume for the semantics that during each program execution, only those cells are accessed which have been allocated.

$$\begin{array}{lll}
\mathit{Addr}_h & = \{\mathsf{null}\} \cup \{\mathsf{ref}\, a \mid a \in \{0, \ldots, h-1\}\} & \text{adresses} \\
\mathit{Val}_h & = \mathit{Addr}_h \cup \mathbb{Z} & \text{values} \\
\mathit{Store}_h & = (\mathit{Addr}_h \times \mathbb{N}_0\}) \to \mathit{Val}_h & \text{memory} \\
\mathit{State}_h & = (\mathit{Vars} \to \mathit{Val}_h) \times \{h\} \times \mathit{Store}_h & \\
\mathit{State} & = \bigcup_{h \geq 0} \mathit{State}_h & \text{states}
\end{array}$$

The program state has an integer component, h, which keeps track of how many blocks have already been allocated, that is, how often the statement $\mathsf{new}()$ has been executed. The set of values also contains, in addition to the integer numbers, addresses of memory blocks. In each state this is an address between $\mathsf{ref}\ 0$ and $\mathsf{ref}\ h-1$, i.e., the ith allocated block is associated with address $\mathsf{ref}\ i-1$. Addresses are values of pointer variables. Recall that pointers may only point to the beginning of memory blocks and not inside blocks. A program state consists of a variable binding and a memory. The memory associates a value with each cell in each allocated block.

Let $(\rho, h, \mu) \in \mathit{State}$ be a program state. The concrete edge effects for the new statements are:

$$\begin{array}{rl}
[\![R \leftarrow \mathsf{new}(e)]\!]\,(\rho, h, \mu) & = (\rho \oplus \{R \mapsto \mathsf{ref}\, h\}, h+1, \\
& \quad\ (\mu \oplus \{(\mathsf{ref}\, h, i) \mapsto \mathsf{null} \mid i \in [\![e]\!]\,\rho\}) \\
[\![z \leftarrow R[e]]\!]\,(\rho, h, \mu) & = (\rho \oplus \{z \mapsto \mu\,(\rho\, R, [\![e]\!]\,\rho)\}, h, \mu) \\
[\![R[e_1] \leftarrow e_2]\!]\,(\rho, h, \mu) & = (\rho, h, \mu \oplus \{(\rho\, R, [\![e_1]\!]\,\rho) \mapsto [\![e_2]\!]\,\rho\})
\end{array}$$

The most complex operation is the operation $\mathsf{new}()$. According to our semantics, it performs the following steps:

1. It computes the size of the new block;

[1] Note that this roughly corresponds to a change from considering the memory component as a contiguous array containing directly addressed objects to a managed memory component where new blocks are dynamically allocated.

2. It provides the new block—by incrementing h;
3. It initializes all memory cells in the new block with null (or any other value we could have selected);
4. It returns, into R, the address of the new block.

This semantics is very detailed since it works with *absolute* addresses. For some purposes, it may be even too detailed. Consider, for example, the two following program fragments:

$$\begin{array}{ll} X \leftarrow \mathsf{new}(4); & \qquad Y \leftarrow \mathsf{new}(4); \\ Y \leftarrow \mathsf{new}(4); & \qquad X \leftarrow \mathsf{new}(4); \end{array}$$

After executing the left program fragment, X and Y have received the values $\mathsf{ref}\,1$ and $\mathsf{ref}\,2$ while after executing the right program fragment, X and Y have received the values $\mathsf{ref}\,2$ and $\mathsf{ref}\,1$. The two fragments, therefore, cannot be considered as equivalent. In many cases, though, the semantics of a program is meant to be independent of the precise values of addresses. In these cases, program states should be considered as equal, if they are equal—up to some *permutation* of addresses appearing in the program states.

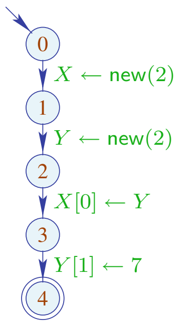
A Flow-Sensitive Points-to Analysis

A pointer variable may contain several different values at a program point when program execution reaches this program point several times. We design a *points-to analysis*, which determines for each pointer variable a superset of these values, that is, all the addresses that the pointer variable may contain. After these supersets are computed one can check whether two pointer variables have a nonempty intersection of their possible values. Those for that this is the case may be aliases of each other.

Starting with the concrete semantics we define an analysis for this problem. The analysis has to deal with potentially infinitely many concrete addresses created by executing new operators in loops. It needs a way to abstract this potentially infinite set to a set of bounded size. Our analysis uses *allocation sites*, that is, statements in which a new operator occurs to partition the potentially infinite set into finitely many sets, represented by *abstract* addresses. It also does not distinguish the contents of the different cells within a block, but manages for each block a set of addresses possibly contained in any one of its cells.

The analysis describes all addresses created by executing an edge $(u, R \leftarrow \mathsf{new}(e), v)$ by one abstract address, which is identified with the starting point u of the edge. We define:

$$\begin{array}{lll} \mathit{Addr}^\sharp & = \mathit{Nodes} & \text{abstract addresses} \equiv \text{creation points} \\ \mathit{Val}^\sharp & = 2^{\mathit{Addr}^\sharp} & \text{abstract values} \\ \mathit{Store}^\sharp & = \mathit{Addr}^\sharp \to \mathit{Val}^\sharp & \text{abstract memory} \\ \mathit{State}^\sharp & = (\mathit{Pointers} \to \mathit{Val}^\sharp) \times \mathit{Store}^\sharp & \text{abstract states} \end{array}$$



	X	Y	0	1
0	$\{0,1\}$	$\{0,1\}$	$\emptyset$	$\emptyset$
1	$\{0\}$	$\{0,1\}$	$\emptyset$	$\emptyset$
2	$\{0\}$	$\{1\}$	$\emptyset$	$\emptyset$
3	$\{0\}$	$\{1\}$	$\{1\}$	$\emptyset$
4	$\{0\}$	$\{1\}$	$\{1\}$	$\emptyset$

Fig. 1.33 The abstract states for the program of Example 1.11.1

Pointers $\subseteq$ *Vars* is the set of pointer variables. The abstract states ignore all *int* values and the special pointer constant null. We will use the generic name (D, M) for abstract states. Abstract states have a canonical partial order, derived from set inclusion:

$$\begin{array}{lll}(D_1, M_1) \sqsubseteq (D_2, M_2) & \text{if} & (\forall\, R \in \mathit{Pointers}.\ D_1(R) \subseteq D_2(R)) \ \wedge \\ & & (\forall\, u \in \mathit{Addr}^\sharp.\ M_1(u) \subseteq M_2(u))\end{array}$$

Example 1.11.3 Let us regard again the program of Example 1.11.1. Figure 1.33 shows the abstract states for the different program points. The analysis does not lose any information in this example since each edge allocating a new block is only visited once, and since each block is assigned an address only once. □

We have seen above that the points-to analysis we are about to design will, in general, have imprecise information about where a pointer variable or a pointer expression point to. Let us consider how this lack of information propagates: It propagates from the right side of an assignment to the left side, from a pointer component in memory to the left side of a read if the analysis has already collected several possible values for this pointer component. It may increase when the analysis accumulates information for all possible target addresses for a write to memory.
The abstract edge effects for the points-to analysis are:

$$\begin{array}{rcl}[\![(\_, R_1 \leftarrow R_2, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{R_1 \mapsto D\,R_2\}, M) \\ [\![(u, R \leftarrow \mathsf{new}(e), \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{R \mapsto \{u\}\}, M) \\ [\![(\_, R_1 \leftarrow R_2[e], \_)]\!]^\sharp\,(D, M) & = & 0(D \oplus \{R_1 \mapsto \bigcup\{M\,a \mid a \in D\,R_2\}\}, M) \\ [\![(\_, R_1[e_1] \leftarrow R_2, \_)]\!]^\sharp\,(D, M) & = & (D, M \oplus \{a \mapsto (M\,a) \cup (D\,R_2) \mid a \in D\,R_1\})\end{array}$$

All other statements do not change the abstract state.

The edge effects for those edge labels that allocate new blocks now depend on the whole edge. *Assignments* to a variable overwrite the corresponding entry in the variable binding D. This was what we had before considering pointers. Overwriting the entry for a variable in an abstract variable binding is called *destructive update*. Destructive update, although it may sound negative, leads to more precise information. In the presence of pointers, we resort to *nondestructive updates* since a pointer variable or pointer expression at some program point may point to different memory cells when the program point is reached several times. Non-destructive update accumulates all possibilities that cannot be excluded and may therefore lead to less precise information.

For a *read* from a block in memory, the address is not necessarily known. To be on the safe side, the new value of a pointer variable on the left side is defined as the union of the contributions of all blocks whose abstract addresses the analysis has collected for the right side.

For a *write* to memory, we need to take care of the case of multiple abstract target addresses a, which may each correspond to a set of concrete target addresses. Writing to memory can therefore not be recorded *destructively*, i.e., by overwriting. Instead, the set of addresses forming the potential new abstract address is *added* to the sets $M\,a$.

Without initializing new blocks the analysis would have to assume for each block that it may contain any possible value. Only since the operation $\mathsf{new}()$ returns initialized blocks can the analysis produce any meaningful information about memory contents. Alternatively, we could assume that a correct program execution would never use the contents of an uninitialized memory cell as address. Program behavior is exactly as in the case of a program where each cell of a newly allocated block is initialized to null before the block is used.

A System of Inequalities

A system of inequalities is derived from the control-flow graph of a program based on the abstract domain $\mathit{State}^\sharp$ and the abstract edge effects. Nothing is known about the values of pointer variables before program execution. No blocks are, yet, allocated. The initial abstract state is therefore $(D_\emptyset, M_\emptyset)$, where

$$D_\emptyset\, x = \emptyset, \quad D_\emptyset\, R = \mathit{Addr}^\sharp, \quad M_\emptyset\, a = \emptyset$$

for all *int*-variables x, all pointer variables R, and all abstract addresses a.

Let $\mathcal{P}[v]$ for all program points v be the least solution of the system of inequalities. This least solution associates with each program point v an abstract state $\mathcal{P}[v] = (D, M)$ that delivers for each pointer variable R a superset of the abstract addresses of memory blocks to which R may point when v is reached. In consequence, R is not an alias of any other pointer variable R' if $(D\ R) \cap (D\ R') = \emptyset$.

Note that we ignore the concrete value null in the abstract state. The error of dereferencing a null pointer can therefore not be detected nor the absence of such an error be proved.

We would like to prove the correctness of the analysis. Disappointingly, this proof is not possible with respect to the operational semantics we started with. This is due to the fact that different program executions may perform the hth allocation of a block at different edges. However, we have already complained about our operational semantics being too detailed as it uses absolute addresses. The number h of an allocation should have no semantical significance. One way out of this problem is to use an *auxiliary semantics*, which is *instrumented* with additional, helpful information. The auxiliary semantics does not just use the values $\mathsf{ref}\, h, h \in \mathbb{N}_0$, as concrete addresses. Instead, it uses:

$$Addr = \{\mathsf{ref}\,(u, h) \mid u \in Nodes, h \in \mathbb{N}_0\}$$

Thus, the instrumented concrete semantics keeps track of the source node of the edge at which a new block is allocated. The addresses grouped at the edges this way can be easily mapped to abstract addresses. First, a proof of correctness with respect to the instrumented semantics needs to performed for the analysis. Then the equivalence of the original and the instrumented semantics needs to be shown. Exercise 23 gives the reader the opportunity to produce these two proofs.

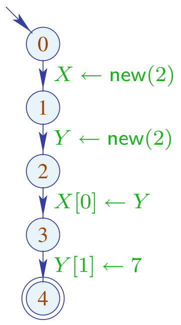
A Flow-Insensitive May-Alias Analysis

The points-to analysis described so far keeps one abstract memory for each program point. It may be quite expensive if there are many abstract addresses. On the other hand, the abstract edge effects do not employ destructive operators on the abstract memory. Therefore the abstract memories at all program points within a loop are the same! In order to reduce the complexity of the analysis, we therefore may prefer to compute just *one* abstract state (D, M) and hope not to lose too much information. The single abstract state then describes the concrete states at *all* program points. This is an example of *flow-insensitive* analysis.

Example 1.11.4 Let us consider again the program of Example 1.11.1. The expected result of the analysis is shown in Fig. 1.34. No loss of information is encountered since each program variable and each memory cell receives a value only once. □

An Efficient Implementation

The implementation of the flow-insensitive analysis merits some more consideration. We introduce one unknown $\mathcal{P}[R]$ per program variable R and one unknown $\mathcal{P}[a]$ per abstract address a instead of considering the one global abstract state as a whole.



X	Y	0	1
$\{0\}$	$\{1\}$	$\{1\}$	$\emptyset$

Fig. 1.34 The result of the flow-insensitive analysis of the program of Example 1.11.1

An edge (u, lab, v) of the control-flow graph leads to the following inequalities:

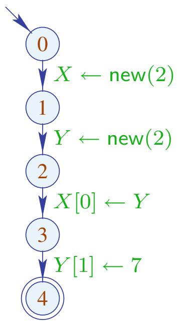
Lab	*Inequalities*
$R_1 \leftarrow R_2$	$\mathcal{P}[R_1] \supseteq \mathcal{P}[R_2]$
$R \leftarrow \text{new}(e)$	$\mathcal{P}[R] \supseteq \{u\}$
$R_1 \leftarrow R_2[e]$	$\mathcal{P}[R_1] \supseteq \bigcup\{\mathcal{P}[a] \mid a \in \mathcal{P}[R_2]\}$
$R_1[e] \leftarrow R_2$	$\mathcal{P}[a] \supseteq (a \in \mathcal{P}[R_1])\,?\,\mathcal{P}[R_2] \,:\, \emptyset$ for all $a \in Addr^\sharp$

All other edges do not have an effect. In this system of inequalities, the inequalities for assignments to pointer variables and read operations are no longer destructive. We assume that all pointer variables are initialized with *null* at program start to be able to compute nontrivial information for pointer variables. Alternatively, we could assume that the first access will only happen after an initialization. The system of inequalities has a least solution $\mathcal{P}_1[R]$, $R \in Pointers$, $\mathcal{P}_1[a]$, $a \in Addr^\sharp$ since the right sides of the inequalities are monotonic functions over the set of addresses. This least solution can again be determined by round-robin iteration.

In order to prove the correctness of a solution $s^\sharp \in State^\sharp$ of the system of inequalities, it suffices to show for each edge k of the control-flow graph that the following diagram commutes:

$$\begin{tikzcd}
s \arrow[rr, "[\![k]\!]"] \arrow[dr, dash, "\Delta"'] & & s_1 \arrow[dl, dash, dotted, "\Delta"] \\
& s^\sharp &
\end{tikzcd}$$

where Δ is a description relation between concrete and abstract values. The system of inequalities has the size $\mathcal{O}(k \cdot n)$, if k is the number of needed abstract addresses and n is the number of edges in the control-flow graph. The values which the fixed-point algorithm computes are sets of a cardinality less than or equal to k. Therefore,



$$\equiv \; = \; \{\{X\}, \{Y, X[]\}, \{Y\}, \{Y[]\}\}$$

Fig. 1.35 The equivalence classes of the relation $\equiv$ for the program of Example 1.11.1

the values of each of the unknowns $\mathcal{P}_1[R]$ and $\mathcal{P}_1[a]$ can change at most k times. Given the low precision of the flow-insensitive analysis this method is still rather expensive. Also, for may alias analysis, one is not interested in in the sets $\mathcal{P}_1[R]$ or $\mathcal{P}_1[a]$ themselves, but whether or nor their pairwise intersection is nonempty.

In order to do so, we consider two radical modifications of the flow-insensitive points-to analysis. First, we replace the set $Addr^\sharp$ of abstract addresses with the set of all expressions $R[]$, R a pointer variable. The abstract address $R[]$ then represents all memory blocks possibly pointed at by R. Second, we no longer consider *inclusions* of abstract values but *equivalences*. Let $Z = \{R, R[] \mid R \in Pointers\}$. Two elements from Z should be as equivalent, if they may represent variables or memory blocks which may contain the same address. Accordingly, the idea is to compute an *equivalence relation* $\equiv$ on the set Z of variables and abstract addresses.

Example 1.11.5 Consider again the trivial program of Example 1.11.1. Figure 1.35 shows an equivalence relation for this example. The equivalence relation directly indicates which pointer expressions possibly evaluate to the same addresses different from null. □

Let $\mathbb{E}$ be the set of equivalence relations over Z. We regard an equivalence relation $\equiv_1$ as less than or equal to another equivalence relation $\equiv_2$, if $\equiv_2$ contains more equivalences than $\equiv_1$, that is, if $\equiv_1 \subseteq \equiv_2$. $\mathbb{E}$ is a complete lattice with respect to this order.

Like the preceding points-to analysis, the new alias analysis is flow-insensitive, that is, one equivalence relation is computed for the whole program. As any equivalence relation, $\equiv$ can be represented as the *partition* $\pi = \{P_1, \ldots, P_m\}$ of pointer variables and abstract addresses that are considered as equivalent. Let $\equiv_1$ and $\equiv_2$ be equivalence relations and π_1 and pi_2 be the associated partitions. Then $\equiv_1 \subseteq \equiv_2$ holds if and only if the partition π_1 is a *refinement* of the partition π_2, that is, if each equivalence class $P_1 \in \pi_1$ is contained in an equivalence class $P_2 \in \pi_2$.

An individual equivalence class $P \subseteq Z$ of an equivalence relation π should be identified by a *representative* $p \in P$. For simplicity, we choose this representative in *Pointers* whenever $P \cap Pointers \neq \emptyset$. Let $\pi = \{P_1, \ldots, P_r\}$ be a partition and p_i

be the representative of the equivalence class P_i. The analysis we aim at needs the following two operations over π:

$\mathit{Pointers}\ \mathsf{find}\,(\pi, p)$	returns the representative of class P_i where $p \in P_i$
$\mathit{Partition}\ \mathsf{union}\,(\pi, p_{i_1}, p_{i_2})$	returns $\{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\}$ i.e., forms the union of the two represented classes.

If $R_1, R_2 \in \mathit{Pointers}$ are equivalent then we regard $R_1[]$ and $R_2[]$ as equivalent. Therefore, the operation union will be applied *recursively*:

```
Partition union* (π, q1, q2) {
        $p_{i_1} \leftarrow \mathsf{find}\,(\pi, q_1)$;
        $p_{i_2} \leftarrow \mathsf{find}\,(\pi, q_2)$;
        if ($p_{i_1} = p_{i_2}$) return $\pi$;
        else {
            $\pi \leftarrow \mathsf{union}\,(\pi, p_{i_1}, p_{i_2})$;
            if ($p_{i_1}, p_{i_2} \in \mathit{Pointers}$) return union* ($\pi, p_{i_1}[], p_{i_2}[]$);
            else return $\pi$;
        }
}
```

The operation union as well as the derived operation union* are monotonic on partitions. The alias analysis using these operations iterates exactly once over the edges of the control-flow graph and unifies the left and the right side when it encounters an edge at which pointers are changed:

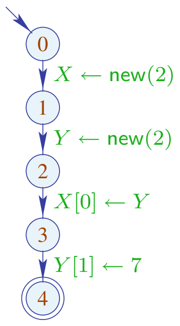
$$\begin{array}{l}\pi \leftarrow \{\{R\}, \{R[]\} \mid R \in \mathit{pointer}\};\\ \textbf{forall}\ ((\_, \mathit{lab}, \_)\ \text{edge})\ \ \pi \leftarrow [\![\mathit{lab}]\!]^\sharp\,\pi;\end{array}$$

Thereby, we have:

$$\begin{array}{rcl}[\![R_1 \leftarrow R_2]\!]^\sharp\,\pi &=& \mathsf{union}^*\,(\pi, R_1, R_2)\\ [\![R_1 \leftarrow R_2[e]]\!]^\sharp\,\pi &=& \mathsf{union}^*\,(\pi, R_1, R_2[])\\ [\![R_1[e] \leftarrow R_2]\!]^\sharp\,\pi &=& \mathsf{union}^*\,(\pi, R_1[], R_2)\\ [\![\mathit{lab}]\!]^\sharp\,\pi &=& \pi \qquad\qquad \text{otherwise}\end{array}$$

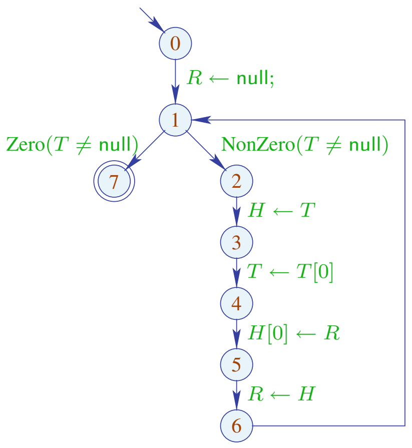
Example 1.11.6 Consider again the program of Example 1.11.1. Figure 1.36 shows the steps of the new analysis for this program. □

Example 1.11.7 Let us also look at the result of the flow-insensitive alias analysis for the program of Example 1.11.2 to reverse lists in Fig. 1.37. The result of the analysis is not very precise: All pointer variables and all blocks may be aliases of each other. □


0
$X \leftarrow \mathsf{new}(2)$
1
$Y \leftarrow \mathsf{new}(2)$
2
$X[0] \leftarrow Y$
3
$Y[1] \leftarrow 7$
4

	$\{\{X\},\{Y\},\{X[]\},\{Y[]\}\}$
$(0,1)$	$\{\{X\},\{Y\},\{X[]\},\{Y[]\}\}$
$(1,2)$	$\{\{X\},\{Y\},\{X[]\},\{Y[]\}\}$
$(2,3)$	$\{\{X\},\boxed{\{Y,X[]\}},\{Y[]\}\}$
$(3,4)$	$\{\{X\},\{Y,X[]\},\{Y[]\}\}$

Fig. 1.36 The flow-insensitive alias analysis for the program of Example 1.11.1


0
$R \leftarrow \mathsf{null};$
1
$\mathsf{Zero}(T \neq \mathsf{null})$
$\mathsf{NonZero}(T \neq \mathsf{null})$
7
2
$H \leftarrow T$
3
$T \leftarrow T[0]$
4
$H[0] \leftarrow R$
5
$R \leftarrow H$
6

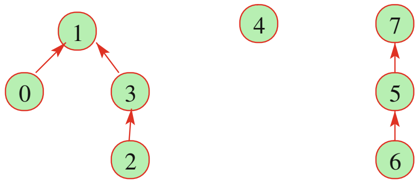
	$\{\{H\},\{R\},\{T\},\{H[]\},\{R[]\},\{T[]\}\}$
$(2,3)$	$\{\boxed{\{H,T\}},\{R\},\boxed{\{H[],T[]\}},\{R[]\}\}$
$(3,4)$	$\{\boxed{\{H,T,H[],T[]\}},\{R\},\{R[]\}\}$
$(4,5)$	$\{\boxed{\{H,T,R,H[],R[],T[]\}}\}$
$(5,6)$	$\{\{H,T,R,H[],R[],T[]\}\}$

Fig. 1.37 Result of the analysis for Example 1.11.2

The alias analysis iterates once over the edges. This is no accident. A second iteration would not change the partition, see Exercise 24. This method computes the least solution of the system of inequalities over partitions:

$$\mathcal{P}_2 \sqsupseteq [\![lab]\!]^\sharp\, \mathcal{P}_2\,, \quad (\_, lab, \_) \quad \text{edge of the control-flow graph}$$

The correctness proof again assumes that all accesses to cells only happen after these have been initialized. Let us now estimate the needed effort for the alias analysis. Let k be the number of pointer variables and n be the number of edges in the control-flow graph. Each edge is considered exactly once. For each edge, there is at most one call to the function union^*. Each call to union^* performs two calls of the function *find*. The operation union and possibly also recursively the function union^* are only called if these calls to *find* return representatives of two different equivalence classes. At the beginning, there are $2k$ equivalence classes. Each call to union decreases the



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

Fig. 1.38 The partition $\pi = \{\{0, 1, 2, 3\}, \{4\}, \{5, 6, 7\}\}$ of the set $\{0, \ldots, 7\}$ represented by parent links

number of equivalence classes. So, at most $2k - 1$ calls of the operation union are possible and therefore at most $\mathcal{O}(n + k)$ calls of the operation find.

We need an efficient data structure to support the operations union and find. Such *union-find* data structures are well-known in the literature. We present a particularly simple implementation, invented by Robert E. Tarjan. A partition of a finite base set U is represented as a *directed forest*:

- For each $u \in U$ there exists a parent link $F[u]$.
- An element u is a *root* in the directed forest if the parent link points from u to u, i.e. if $F[u] = u$.

All nodes that may indirectly reach the same root through their parent links form an equivalence class, whose representative is the root.

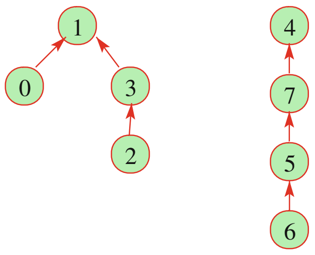
Figure 1.38 shows the partition $\{\{0, 1, 2, 3\}, \{4\}, \{5, 6, 7\}\}$ of the base set $U = \{0, \ldots, 7\}$. The lower part shows the representation by an array F with parent links, which are visualized above the array.

The operations find and union can be easily implemented in this representation.

find: To find the representative of the equivalence class of an element u it suffices to follow the parent links starting at u until an element u' is found whose parent link points to u'.

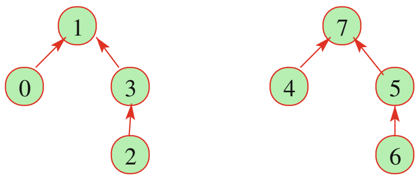
union: To form the union of the equivalence classes of two representatives u_1 and u_2 the only action needed is to make the parent link of one of the elements point to the other element. The result of applying the union operation to the example partition of Fig. 1.38 is shown in Fig. 1.39.

The operation union only requires $\mathcal{O}(1)$ steps. The costs of the operation find, however, are proportional to the length of the path from the element at the start of the search to the root of the associated tree. This path can be very long in the worst case. An idea to prevent long paths is to always hang the smaller tree below the bigger one. Using this strategy in the example of Fig. 1.38, the operation union would set



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	4

Fig. 1.39 The result of applying the operation union(π, 4, 7) to the partition π of Fig. 1.38

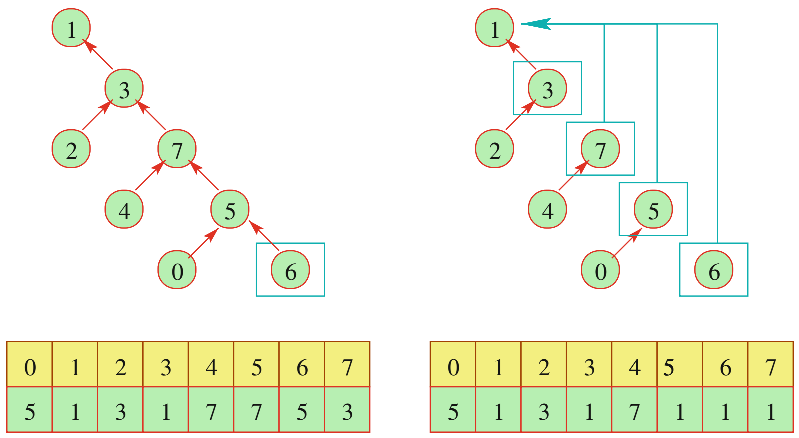


0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7

Fig. 1.40 The result of applying the operation union(π, 4, 7) to the partition π of Fig. 1.38 considering the size of the involved equivalence classes

the parent link of the element 4 to 7 and not the other way round (Fig. 1.40). The algorithm needs to account for the size of the equivalence classes in order to know which class is the smaller and which is the bigger. This makes the costs of the union operation slightly more expensive. Let n be the number of union operations that are applied to the initial partition $\pi_0 = \{\{u\} \mid u \in U\}$. The length of the paths to a root is then at most $\mathcal{O}(\log(n))$. Accordingly, each find operation has costs at most $\mathcal{O}(\log(n))$.

Amazingly enough, this data structure can be improved even further. To do this, the algorithm redirects the parent links of all visited elements directly to the root of the associated tree during a find operation. This increases the costs of each find operation by a small constant factor, but decreases the costs of later find operations. Figure 1.41 shows how the paths to the root are shortened when this idea is used.



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3

0	1	2	3	4	5	6	7
5	1	3	1	7	1	1	1

Fig. 1.41 Path compression by the find operation for 6

The left tree has paths of length up to 4. A find inquiry for node 6 turns nodes 3, 7, 5 and 6 into direct successors of root 1. This shortens the paths in the example to lengths of at most 2.

This implementation of a union-find data structure has the property that n union operations and m find operations together only have costs $\mathcal{O}((n + m) \cdot \log^*(n))$, where $\log^*$ is the inverse of the iterated exponentiation function: $\log^*(n)$ is the least number k such that $n \leq 2^{2^{\cdot^{\cdot^{2}}}}$ for a tower of exponentiations of height k. The function $\log^*$ therefore is an incredibly slowly growing function, which has a value ≤ 5 for all realistic inputs n. A proof for the upper bound can be found in textbook about data structures and algorithms, such as the book by Cormen, Leiserson, Rivest and Stein (2009).

Conclusion 1.11.1 This section has presented methods to analyze programs using pointers and dynamically allocated blocks of storage. We started with a flow-sensitive points-to analysis, which computes individual information for each program point. It uses destructive updating of analysis information for assignments to pointer variables, but accumulates the possible values at accesses for dynamically allocated storage. At the cost of losing the destructive update for program variables, we developed a possibly more efficient flow-insensitive points-to analysis, which produces only one analysis information describing all program states occurring during program execution. In case we are only interested in alias information, flow-insensitive analysis can be used which partitions pointer variables and abstract addresses of blocks into equivalences classes of possible aliases. This latter analysis is based on a union-find data structure and is very fast, but may be very imprecise on programs with complex pointer manipulations. □

1.12 Fixed-Point Algorithms

The last section detailed our search for an analysis of aliases, which is as efficient as possible. This leads to the question of how one, in general, computes (if possible, least) solutions of systems of inequalities over complete lattices. The only practical procedure we have met so far to determine solutions of systems of inequalities

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

is round-robin iteration. It is easily implementable and can nicely be manually simulated. However, this procedure has its disadvantages. First, it needs a whole round to detect termination of the iteration. Second, it reevalutes all right sides f_i for the unknowns x_i anew, although only the value of one variable might have changed since the last round. Last, the runtime depends heavily on the used order of the variables.

A more efficient algorithm is the *worklist algorithm*. This procedure administers the set of variables x_i whose values might no longer satisfy their inequality in a data structure W, the *worklist*. For a variable x_i taken out of the worklist, the value of its right side is computed using the actual values of the unknowns. The old value of x_i is replaced by a new value that subsumes the previous and the newly computed value of x_i if the newly computed value is not subsumed by the previous value. The worklist has been shortened by taking out one element. In the case that the value of x_i has grown the inequalities whose right sides depend directly on the value of x_i might be no more satisfied. The left sides of these possibly violated inequalities are remembered for a recomputation, i.e., inserted into the worklist W.

The implementation of this procedure uses for each variable x_i the set $I[x_i]$ of all variables whose right side possibly depends directly on x_i. These direct dependences between variables are easily identified in the examples of program analyses presented so far: In a forward analysis, the value for a program point u influences the value at a program point v directly if there is an edge from u to v in the program control-flow graph. Analogously, the value at v influences the value at u in a backwards analysis if there is an edge from u to v. The precise determination of dependences may not always be that easy. It may be more difficult if the right sides of constraints are only given as *semantic functions* f_i whose implementation is unknown.

In the description of the algorithm, we again distinguish between the *unknowns* x_i and their *values*. The values of variables are stored in the array D, which is indexed with variables. The worklist W, on the other hand, administers unknowns and not values. In our formulations of generic systems of inequalities, we have always assumed that the right sides f_i are functions of type $\mathbb{D}^n \to \mathbb{D}$, i.e. may possibly depend on *all* variables. We now want to take into account that evaluating the right side of a variable may access the values only of *some* other variables. Therefore, we now consider right sides f_i of the functionality

$$f_i \ : \ (X \to \mathbb{D}) \to \mathbb{D}$$

where $X = \{x_1, \dots, x_n\}$ is the set of unknowns of the system of inequalities. Such a function f_i expects a binding of the unknowns to values and returns a value. When the function accesses the value of a variable x_j, this value is obtained by applying the variable binding to x_j. Since the actual values of variables are stored in the array D, the actual binding of the unknowns is delivered by the function eval:

$$\mathbb{D}\ \mathsf{eval}(x_j)\ \{\ \mathbf{return}\ D[x_j];\ \}$$

The implementation of the worklist iteration looks as follows:

```
$W \leftarrow \emptyset$;
forall ($x_i \in X$) {
        $D[x_i] \leftarrow \bot$;  $W \leftarrow W \cup \{x_i\}$;
}
while (exists $x_i \in W$) {
        $W \leftarrow W \backslash \{x_i\}$;
        $t \leftarrow f_i$ eval;
        $t \leftarrow D[x_i] \sqcup t$;
        if ($t \neq D[x_i]$) {
                $D[x_i] \leftarrow t$;
                $W \leftarrow W \cup I[x_i]$;
        }
}
```

The set W of variables whose right sides need to be reevaluated can be administered in a simple list structure where insertions and extractions are performed *last in first out*, i.e., which behaves like a stack. Note that the last line of the body of the *while* loop indicates that elements from $I[x_i]$ need only be inserted into W if they are not yet in there. Another array of Boolean flags can be used to maintain this membership information and thus to avoid double insertions into the worklist.

Example 1.12.1 To illustrate the worklist algorithm, we have again a look at the system of inequalities of Example 1.5.2:

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

The right sides in this system are given by expressions which explicitly expose the variable dependences.

	I
x_1	x_3
x_2	
x_3	x_1, x_2

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	a, c	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	a, c	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	a, c	$\boxed{x_2}$
$\{a, c\}$	a	a, c	

Fig. 1.42 The worklist-based fixed-point iteration for Example 1.5.2

The steps of the worklist algorithm applied to this system of inequalities is shown in Fig. 1.42. The next variable x_i to be taken out of the worklist is emphasized in the actual worklist. Altogether six evaluations of right sides suffice. This would not be beaten by a round-robin iteration. □

The next theorem collects our observations about the worklist algorithm. To specify its precise runtime we recall that the *height* h of a complete lattice $\mathbb{D}$ is defined as the maximal length of any strictly ascending chain of elements in $\mathbb{D}$. The *size* $|f_i|$ of a right side f_i is defined as the number of variables that are possibly accessed during the evaluation of f_i. The *sum* of the sides of all right sides therefore is given by:

$$\sum_{x_i \in X} |f_i| = \sum_{x_j \in X} \#I[x_j]$$

This equality results from the fact that each variable dependence $x_j \to x_i$ is counted exactly once in the sum on the left and also exactly once in the sum on the right side. Accordingly, the *size* of the system of inequalities in a set of unknowns X is defined as the sum $\sum_{x_i \in X}(1 + \#I[x_i])$. Using this definition we find:

Theorem 1.12.1 *Let S be a system of inequalities of size N over the complete lattice $\mathbb{D}$ of height $h > 0$. We have:*

1. *The worklist algorithm terminates after at most $h \cdot N$ evaluations of right sides.*
2. *The worklist algorithm produces a solution. It delivers the least solution if all f_i are monotonic.*

Proof To prove the first claim, we observe that each variable x_i can only change its value at most h times. This means that the list $I[x_i]$ of variables depending on x_i is added to the worklist at most h times. Therefore, the number of evaluations is bounded from above by:

$$\begin{aligned} & n + \textstyle\sum_{i=1}^{n} h \cdot \# I[x_i] \\ = {} & n + h \cdot \textstyle\sum_{i=1}^{n} \# I[x_i] \\ \leq {} & h \cdot \textstyle\sum_{i=1}^{n} (1 + \# I[x_i]) \\ = {} & h \cdot N \end{aligned}$$

Of the second claim we only consider the statement about monotonic right sides. Let D_0 be an array which represents the least solution of the system of inequalities. We first prove that we have at any time:

$$D_0[x_i] \sqsupseteq D[x_i] \qquad \text{for all unknowns } x_i.$$

Finally, we convince ourselves that after executing the body of the *while* loop, all variables x_i for which the corresponding inequality is actually violated, are contained in the worklist. This worklist is empty when the algorithm terminates. Hence on termination, all inequalities must be satisfied, and the array D therefore represents a solution. The least solution of the system of inequalities is an upper bound of this solution. Consequently, the found solution must be equal to the least solution. □

According to Theorem 1.12.1, the worklist algorithm finds a solution also in case of nonmonotonic right sides. This solution is not necessarily a least solution. It is just some solution. A similar behavior has been observed for round-robin iteration.

The worklist algorithm can be simplified if all right sides are monotonic. The accumulation at the recomputation of the values can then be replaced with overwriting.

$$\boxed{t \leftarrow D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{;}$$

For iterations using widening accumulation works again differently: In this case, the widening operator $\sqcup\!\!\!\!-$ is applied to the old and the new value instead of the least upper bound.

$$\boxed{t \leftarrow D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t \leftarrow D[x_i] \sqcup\!\!\!\!- \; t;}$$

In case of narrowing we have:

$$\boxed{t \leftarrow D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t \leftarrow D[x_i] \sqcap\!\!\!\!- \; t;}$$

where the iteration of the *while* loop does not start with the $\bot$ value for each variable but with a previously computed solution of the system of inequalities.

In practice, the worklist algorithm has proved very efficient. It still has two disadvantages:

- The algorithm needs the direct dependences between the unknowns, that is, the sets $I[x_i]$. These dependences were quite obvious in the examples so far. This, however, is not the case in all applications.
- The actual value of a required variable x_i is accessed when the right side of an unknown is evaluated, no matter if this is still a *trivial* value or an already computed nontrivial value.

A better strategy would be to first try to compute a reasonably *good* value for a variable x_j before the value is accessed. To improve further, we extend the function eval by *monitoring*: before function eval delivers the values of a variable x_j it keeps book of the variable x_i for whose right side the value of x_j is needed, that is, the function eval adds x_i to the set $I[x_j]$. Function eval therefore receives variable x_i as a first argument. Function eval should not return the actual values of x_j, but the best possible value for x_j. Therefore, the computation of an as-good-as-possible value for x_j is triggered—even before the variable dependence between x_j and x_i is recorded. Altogther function eval turns into:

```
$\mathbb{D}$ eval $(x_i)$ $(x_j)$ { solve$(x_j)$;
                  $I[x_j] \leftarrow \{x_i\} \cup I[x_j]$;
                  **return** $D[x_j]$;
                }
```

Function eval together with procedure solve recursively compute a solution. To prevent an infinite recursion procedure solve manages a set *stable*. This set *stable* contains all variables for which an evaluation of the corresponding right sides has already been triggered and and not yet finished together with those variables for which (relative to the actual values of variables in the set *stable*) the fixed-point has already been reached. For the variables in the set *stable* procedure solve will not do anything.

Set *stable* is initialized with the empty set at the start of the fixed-point iteration. The program performing the fixed-point iteration looks as follows:

```
$stable \leftarrow \emptyset$;
**forall** $(x_i \in X)$ $D[x_i] \leftarrow \bot$;
**forall** $(x_i \in X)$ solve$(x_i)$;
```

where the procedure solve is defined as follows:

```
**void** solve $(x_i)${
    **if** $(x_i \notin stable)$ {
        $stable \leftarrow stable \cup \{x_i\}$;
        $t \leftarrow f_i$ (eval $(x_i)$);
        $t \leftarrow D[x_i] \sqcup t$;
        **if** $(t \neq D[x_i])$ {
            $D[x_i] \leftarrow t$;
            $W \leftarrow I[x_i]$;   $I[x_i] \leftarrow \emptyset$;
            $stable \leftarrow stable \backslash W$;
            **forall** $(x_i \in W)$ solve$(x_i)$;
        }
    }
}
```

The call of procedure solve(x_i) directly terminates if the variable x_i is already stable. Otherwise, the variable x_i is added to the set *stable*. After that, the right side f_i of x_i is evaluated. Instead fo the actual variable binding, procedure solve uses the function

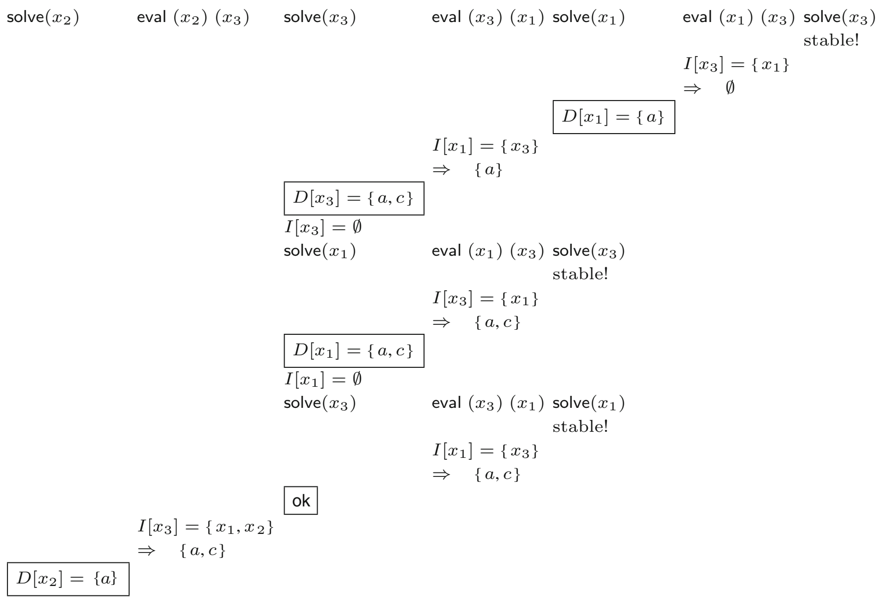


Fig. 1.43 An execution of the recursive fixed-point algorithm

eval partially applied to x_i, which computes when applied to another unknown x_j, the best possible value for x_j, adds x_i to the I set of variable x_j, and only then delivers the value of x_j.

Let t be the least upper bound of the value of x_i and the value delivered by the evaluation of the the right side of x_i. If this value t is subsumed by the previous value of x_i, the call of solve immediately returns. Otherwise the value of x_i is set to t. The change of the value of variable x_i then is propagated to all variables whose last evaluation accessed a smaller value of x_i. This means that their right sides must be scheduled for reevaluation. The set W of these variables is given by the set $I[x_i]$.

The old value of the set $I[x_i]$ is no longer needed and is therefore reset to the empty set: the reevaluation of the right sides of the variables from the set W will reconstruct the variable dependences, should these be needed. The variables of the set W can no longer be regarded as stable. They are therefore removed from the set *stable*. After that, procedure solve is called for all variables in the set W.

Example 1.12.2 We consider again the system of inequalities of Example 1.5.2 and Example 1.12.1:

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

An execution of the recursive fixed-point algorithm is shown in Fig. 1.43. A recursive descent is always shown to the right. The column of a call to procedure solve contains the computed new entries $D[x_i]$ and $I[x_i]$ as well as the calls of procedure solve to treat variables in the set W. An $\boxed{\text{ok}}$ in this column signals that a reevaluation of the right side does not require a change of the actual value of the variable. A stable! indicates that the variable for which the last call to solve was performed is stable, such that the call should directly terminate. The column of function eval indicates changes of the set $I[x_j]$ and the returned values. The algorithm evaluates fewer right sides than the worklist algorithm, although this example is very small. □

The recursive fixed-point algorithm can be elegantly implemented in a programming language such as OCAML, which has assignments on one side, and partial applications of higher-order functions on the other side.
The recursive fixed-point algorithm is more complicated than the worklist algorithm. It executes, in general, less evaluations of right sides. It does not need a precomputation of the variable dependences, and, even better, it also works when variable dependences *change* during the fixed-point iteration. In addition, it has a property that we will later exploit for interprocedural analysis in Sect. 2.5: The algorithm can be modified such that not the values of all unknowns are computed. Rather, the evaluation of an unknown *of interest*, x_i, can be started. Then only those unknowns whose values are needed to evaluate the unknown x_i are themselves evaluated. Fixed-point iterators having this property are called *local*.

1.13 Elimination of Partial Redundancies

We return to our question of how the compiler can speed up program execution by preventing the execution of redundant computations. In Sect. 1.2, we described how the repeated evaluation of an expression e along an edge $u \to v$ can be avoided if the value of e is definitely available in a variable x. This is the case if the assignment $x \leftarrow e$ has been executed on all paths from the entry point of the program to program point u and no variable occurring in the assignment has been changed in between. This optimization replaces a redundant occurrence of e by the variable x. So far, this substitution of e by x at an edge from u to v is only possible if there are occurrences of $x \leftarrow e$ on *all* paths from program entry to u. The optimization now described attempts to replace e by x if $x \leftarrow e$ is only available on *some* paths to u. This availability on some, but not all paths is called *partial redundancy*. We will use *redundancy* and *availability* interchangeably in this section.

Example 1.13.1 Regard the program on the left side of Fig. 1.44.

The assignment $T \leftarrow x + 1$ is evaluated on every path, on the path to the right even twice with identical result. The compiler cannot simply replace the occurrence of $x + 1$ at the edge from 5 to 6 by an access to T, although the value of $x + 1$ that is computed along the edge from 3 to 4 is stored in the variable T. However, the compiler may *move* the occurrence of the assignment $T \leftarrow x]1$ at the edge from 5 to

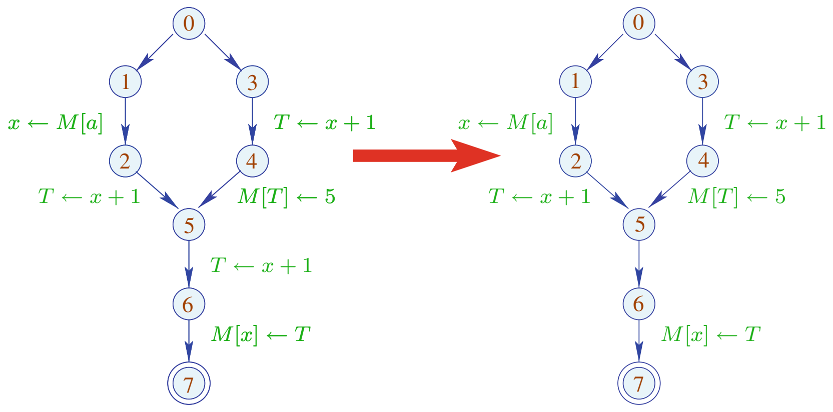


Fig. 1.44 A removal of a partial redundancy

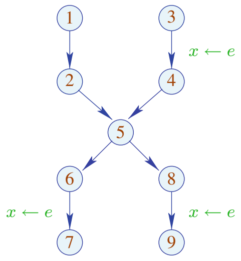


Fig. 1.45 $x \leftarrow e$ is partially redundant at 5, as well as very busy

6 to the edge from 2 to 5 and thus avoid a redundant computation on the right path. This program transformation results in the program on the right of Fig. 1.44. □

We look for a transformation that places assignments $x \leftarrow e$ at points in the program such that

- variable x is guaranteed to contain the value of e whenever the program executes the assignment $x \leftarrow e$ the next time, and
- the insertion of new redundant computations are avoided.

Consider Fig. 1.13. In Fig. 1.45, $x \leftarrow e$ is available along some, but not all paths, synonymously called partially redundant at program point 5. The compiler, therefore, cannot eliminate the two occurrences of $x \leftarrow e$ on the paths starting at 5. We observe, however, that $x \leftarrow e$ is going to be executed on *every* computation starting at program point 5, before the variable x is used or any variable occurring in this statement is overwritten. We say, the assignment $x \leftarrow e$ is *very busy* at program point 5. This also is true for program points 1, 2, and 4. At program point 4, however, $x \leftarrow e$ is already available and therefore need not be computed once more. Instead, the compiler may insert the assignment $x \leftarrow e$ before program point 1. After this insertion, $x \leftarrow e$ is redundant at program points 2 and 5 as well as at the program points 6 an 8

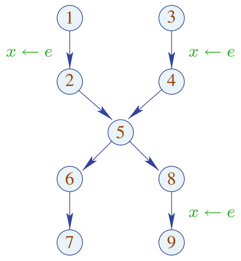


Fig. 1.46 $x \leftarrow e$ is totally redundant at 5, but not very busy

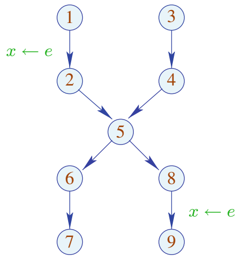


Fig. 1.47 $x \leftarrow e$ is neither partially redundant at 5, nor very busy

and thus allows to remove the assignments there. The situation looks different in Fig. 1.46. There, the assignment $x \leftarrow e$ is already redundant at program point 5 as well as at program point 8. Therefore, already redundancy elimination succeeds in removing the assignment $x \leftarrow e$ at the edge from 8 to 9. Finally in Fig. 1.47, the assignment $x \leftarrow e$ is neither redundant nor very busy at program point 5. Therefore, no optimization is possible.

The transformation therefore inserts an $x \leftarrow e$ at the end of an edge e with endpoint v, if two conditions are satisfied: First, the assignment should not already be available along this edge e. Second, it should be *very busy* at v. This means that the assignment $x \leftarrow e$ is executed on all outgoing paths from v before the left side of the assignment, x, is used, and before any of the variables of the statement is modified.

An Analysis for Very Busy Assignments

A new program analysis is required for determining very busy assignments. An assignment $x \leftarrow e$ is called *busy* along a path π to program exit if π has the form $\pi = \pi_1 k \pi_2$, where k is an assignment $x \leftarrow e$, and π_1 contains no use of the left side, x, and no definition of any variable of $x \leftarrow e$, that is, of any $\{x\} \cup \mathsf{Vars}(e)$.

The assignment $x \leftarrow e$ is *very busy* at program point v if it is busy along every path from v to program exit. The analysis for very busy assignments thus is a *backwards analysis*. Abstract values in this analysis are sets of assignments $x \leftarrow e$ where $x \notin$

$Vars(e)$, like in the available-assignments analysis. The complete lattice therefore is

$$\mathbb{B} = 2^{Ass}$$

where the ordering again is given by the superset relation $\supseteq$. No assignment is very busy at program exit. The abstract edge effect $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ for an edge $k = (u, lab, v)$ depends only on the edge label and is given by:

$$\begin{array}{rcl} [\![;]\!]^\sharp\, B &=& B \\ [\![\mathsf{NonZero}(e)]\!]^\sharp\, B &=& [\![\mathsf{Zero}(e)]\!]^\sharp\, B \;=\; B\backslash\mathsf{Ass}(e) \\ [\![x \leftarrow e]\!]^\sharp\, B &=& \begin{cases} B\backslash(\mathsf{Occ}(x) \cup \mathsf{Ass}(e)) \cup \{x \leftarrow e\} & \text{if } x \notin \mathsf{Vars}(e) \\ B\backslash(\mathsf{Occ}(x) \cup \mathsf{Ass}(e)) & \text{if } x \in \mathsf{Vars}(e) \end{cases} \\ [\![x \leftarrow M[e]]\!]^\sharp\, B &=& B\backslash(\mathsf{Occ}(x) \cup \mathsf{Ass}(e)) \\ [\![M[e_1] \leftarrow e_2]\!]^\sharp\, B &=& B\backslash(\mathsf{Ass}(e_1) \cup \mathsf{Ass}(e_2)) \end{array}$$

The set $\mathsf{Occ}(x)$ denotes the set of all assignments containing an occurrence of x. The abbreviation $\mathsf{Ass}(e)$ for an expression e denotes the set of all assignments whose left side occurs in e. The analysis is supposed to determine *very* busy assignments, i.e., assignments busy along all outgoing paths. It must, therefore, form the intersection of the contributions of all paths from v to program exit. Since the partial order on the set 2^{Ass} is the *superset* relation, the MOP solution for a program point v is

$$\mathcal{B}^*[u] \;=\; \bigcap\{[\![\pi]\!]^\sharp\, \emptyset \mid \pi : u \rightarrow^* stop\}$$

where $[\![\pi]\!]^\sharp$ is the effect of path π like in the other backwards analyses. Thus,

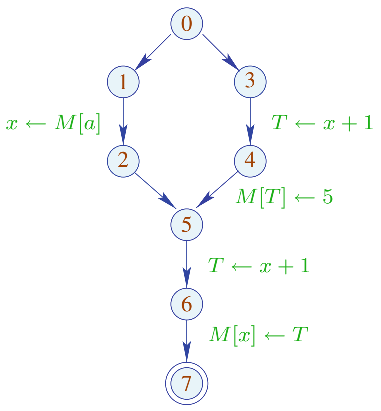
$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

for $\pi = k_1 \ldots k_m$. The abstract edge effects $[\![k_i]\!]^\sharp$ are all distributive. The least solution, $\mathcal{B}$, with respect to the chosen partial order is, therefore, equal to the MOP solution—provided that the program exit is reachable from any program point.

Example 1.13.2 Figure 1.48 shows the sets of very busy assignments for the program of Example 1.13.1. The control-flow graph is acyclic in this case. This means that round-robin iteration can compute these sets in one round. □

Note that the reachability of program exit is of great importance for the result of the backwards analysis for very busy assignments. Let us assume that program point v does not reach program exit. The analysis would start with the set of all assignments at all nodes without outgoing edges. Going backwards, it would remove only those with a new definition of a variable in $\mathsf{Vars}(e) \cup \{x\}$ until it reaches v. The remaining assignments would all be considered very busy at v.

Example 1.13.3 Consider the program of Fig. 1.49.



7	$\emptyset$
6	$\emptyset$
5	$\{T \leftarrow x+1\}$
4	$\emptyset$
3	$\{T \leftarrow x+1\}$
2	$\{T \leftarrow x+1\}$
1	$\emptyset$
0	$\emptyset$

Fig. 1.48 The very busy assignments of Example 1.13.1

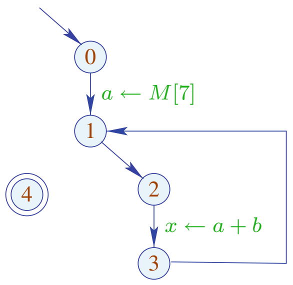


Fig. 1.49 A program whose exit is not reachable

The program admits only one, infinite computation. Program exit, node 4, is not reachable from program point 1. Therefore, any assignment not containing variable x is very busy at 1, even assignments that do not occur in the program. □

The Placement Phase

The compiler has now at its disposal an analysis for partially available (partial redundant) and an analysis for very busy assignments. Assignments recognized as very busy at some program point v are called *movable* at v, and the set of all occurrences of the assignment that make it very busy at v are called its *business sites*. Note that an assignment $x \leftarrow e$ with $x \in Vars(e)$ at some edge from u to v is never movable. Let us now turn to the optimization phase.

There are two different versions in which the transformation can be described: Assume that the assignment $x \leftarrow e$ is movable at v. In the *insertion* version, the compiler inserts copies of the movable assignment $y \leftarrow e$ onto all paths reaching

v on which it was previously not available. This insertion makes the business sites redundant such that they can be eliminated. In the *motion* view of the optimization, the compiler takes the business sites and moves copies backwards into the paths on which the movable assignment was not previously available.

The remaining open question for both views is at what edges to place the copies of the movable assignment $z \leftarrow e$. The answer is that a placement at the new positions should establish that $y \leftarrow e$ is definitely available at all positions at which it has been very busy before. In the control-flow graphs that we consider here, $y \leftarrow e$ is always very busy at the source program-point of an edge labeled with $y \leftarrow e$.

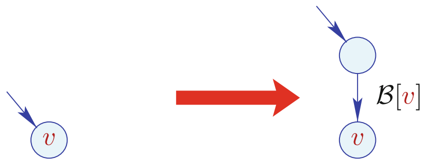
The optimization uses the strategy to place assignments as early as possible, maybe even before the program entry point. This placement is constrained by correctness and by efficiency considerations: Correctness reasons inhibit the movement of the assignment $y \leftarrow e$ over an edge leading from u to v at which y or a variable occurring in e receive a new value. After such a move, y might have a different value at v, or the evaluation of e in the moved assignment might result in a wrong value. For efficiency reasons, the optimization may not move $y \leftarrow e$ onto a path on which it originally would not have been executed. Otherwise, an execution of this path would lead to a potential runtime increase.

There is another argument why we are so cautious not to move move an assignment onto a path that did not contain that assignment before. Depending on the semantics of the programming language, the assignment may have side effects, e.g., throwing an exception at a division by zero. Throwing an exception on a path of the optimizd program where the corresponding path of the original program did not throw an expection, violates the requirement of semantics preservation.

We now go through the different cases for the placement of movable assignments. Let us first consider potential insertions at the entry point, *start*, of the program.

We introduce a new entry point and insert all assignments from $\mathcal{B}[start]$ before *start* to make all assignments from $\mathcal{B}[start]$ definitely available at *start*. This is realized by the first transformation rule.

Transformation PRE for the Start Node:



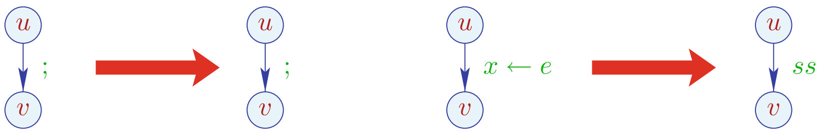
We have taken the liberty to annotate an edge with a set of mutually *independent* assignments, which can be executed in any order. To assure independence of two assignments $y_1 \leftarrow e_1$ and $y_2 \leftarrow e_2 \in \mathcal{B}[u]$ one checks that $y_1 \not\equiv y_2$ and that neither y_1 occurs in e_2 nor y_2 in e_1.

Next, we consider a program point u which is nonbranching. This means that u has exactly one outgoing edge whose label s either is a skip operation, with an assignment or a memory access. This edge is assumed to lead to a program point v. All assignments from $\mathcal{B}[v]$ should be placed at this edge besides the ones that are still very busy at u, i.e., could be moved further, and the ones that were already available at v, i.e., need not be placed here. This set is given by:

$$ss = \mathcal{B}[v] \backslash ([\![s]\!]^\sharp_\mathcal{B}(\mathcal{B}[v]) \cup [\![s]\!]^\sharp_\mathcal{A}(\mathcal{A}[u]))$$

As in Sect. 1.2, $\mathcal{A}[u]$ denotes the set of assignments definitely available at program point u. We have used the indices $\mathcal{A}$ and $\mathcal{B}$ to differentiate the abstract edge effects of available assignments from those for very busy assignments. Let us make this case concrete.

Transformation PRE for Empty Statements and for Movable Assignments:



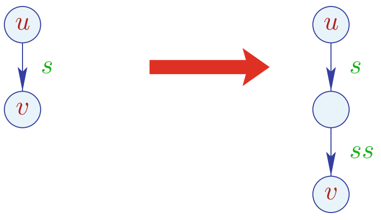
An edge labeled with the empty statement can receive the set ss of assignments where ss is defined as: $\mathcal{B}[v]\backslash(\mathcal{B}[v] \cup \mathcal{A}[u]) = \emptyset$. No additional assignment needs to be placed at this edge.

A *movable* assignment, $x \leftarrow e$ with $x \not\in \mathsf{Vars}(e)$, is moved to another place, and the set ss of assignments is placed at its position where ss is obtained from the definition of ss above by substituting the definitions of the abstract edge effects for their names:

$$\begin{aligned} ss &= \mathcal{B}[v]\backslash(\mathcal{B}[v]\backslash(\mathsf{Occ}(x) \cup \mathsf{Ass}(e)) \cup \mathcal{A}[u]\backslash\mathsf{Occ}(x) \cup \{x \leftarrow e\}) \\ &= (\mathcal{B}[v] \cap \mathsf{Occ}(x)\backslash\{x \leftarrow e\}) \cup (\mathcal{B}[v] \cap \mathsf{Ass}(e)\backslash\mathcal{A}[u]) \end{aligned}$$

Transformation PRE for Nonmovable Statements:

An edge labeled with a *nonmovable* statement s is replaced by a sequence of two edges labeled with s and ss:



The new label ss for an edge from u to v labeled with the nonmovable assignment $x \leftarrow e$ with $x \in \mathsf{Vars}(e)$ is:

$$\begin{aligned} ss &= \mathcal{B}[v] \cap (\mathsf{Occ}(x) \cup \mathsf{Ass}(e))\backslash(\mathcal{A}[u]\backslash\mathsf{Occ}(x)) \\ &= (\mathcal{B}[v] \cap (\mathsf{Occ}(x)) \cup (\mathcal{B}[v] \cap \mathsf{Ass}(e)\backslash\mathcal{A}[u]) \end{aligned}$$

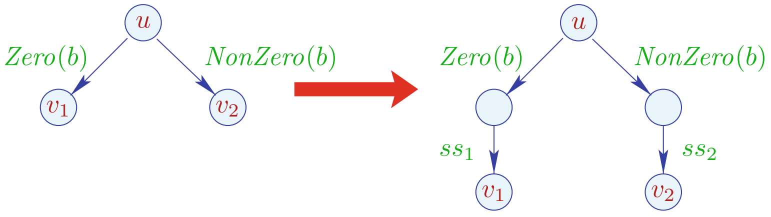
The set of assignments ss to place at a read operation $x \leftarrow M[e]$ are defined analogously. For a write operation $M[e_1] \leftarrow e_2$ we obtain:

$$ss = \mathcal{B}[v] \cap (\mathsf{Ass}(e_1) \cup \mathsf{Ass}(e_2)) \backslash \mathcal{A}[u]$$

It remains to consider a program point u with more than one outgoing edge, i.e., a branching on some condition b.

Transformation PRE for Conditional Branches:

Let v_1 and v_2 be its successor nodes for the cases of 0 and not 0, respectively. Assignments in $\mathcal{A}[u]$ need not be placed at any outgoing edge since they are available at their target nodes already before the transformation. Of the other assignments in $\mathcal{B}[v_1]$ those assignments need to be placed that cannot be moved over u. These are the ones that modify variables of the condition b or that are not contained in $\mathcal{B}[v_2]$. The edge to v_2 is handled analogously. Therefore, we have:



where

$$\begin{aligned} ss_1 &= (\mathcal{B}[v_1] \cap \mathsf{Ass}(b) \backslash \mathcal{A}[u]) \cup (\mathcal{B}[v_1] \backslash (\mathcal{B}[v_2] \cup \mathcal{A}[u]) \\ ss_2 &= (\mathcal{B}[v_2] \cap \mathsf{Ass}(b) \backslash \mathcal{A}[u]) \cup (\mathcal{B}[v_2] \backslash (\mathcal{B}[v_1] \cup \mathcal{A}[u]) \end{aligned}$$

The given transformation rules for PRE make each assignment $x \leftarrow e$ at all program points available at which $x \leftarrow e$ was very busy before the transformation. Therefore, an assignment $x \leftarrow e$ is in particular available at all program points where it would have been computed in the original program.

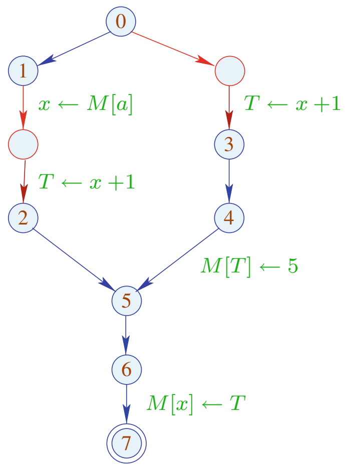
Example 1.13.4 Figure 1.50 shows the analysis information for the program of Example 1.13.1 together with the result of the optimization. In fact, one partially redundant computation could be removed. □

Let ss be the set of assignments that are very busy at v. To prove correctness one shows that for all execution paths π of the program from the original entry point of the program to a program point v and all program states σ before program execution it holds that:

$$[\![ss]\!] \; ([\![\pi]\!] \; \sigma) = [\![k_0\pi]\!]' \; \sigma$$

where k_0 is the new edge leading to the original entry point of the program, and $[\![\pi]\!]$ and $[\![\pi]\!]'$ are the semantics of the program paths π before and after the application of the transformation. The validity of the claim can be proved by induction. For the empty program path $\pi = \epsilon$ it is trivially true. For a nonempty program path $\pi = \pi' k$, it follows from the induction hypothesis with a case distinction according to the label of the last edge k of the path.

As the example indicates, the number of executions of the assignment $x \leftarrow e$ has increased on no path, but may have been decreased on some paths. It would be nice



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{T \leftarrow x + 1\}$
3	$\emptyset$	$\{T \leftarrow x + 1\}$
4	$\{T \leftarrow x + 1\}$	$\emptyset$
5	$\{\emptyset\}$	$\{T \leftarrow x + 1\}$
6	$\{T \leftarrow x + 1\}$	$\emptyset$
7	$\{T \leftarrow x + 1\}$	$\emptyset$

Fig. 1.50 The result of the transformation PRE for the Example 1.13.1 together with the necessary analyses

to prove this property of *nonpessimization* for all programs. However, we will not do this here. The intuition on which the proof is based is that the assignment $x \leftarrow e$ becomes available at all program points in the transformed program at which it was previously very busy and is now up to removal. Each copy of a movable assignment can so be associated with at least one subsequent occurrence that is eliminated.

The elimination of partially redundant assignments also removes some totally redundant assignments as a side effect. More assignments may become partially available by the application of transformation PRE. It may, therefore, pay to apply the transformation PRE repeatedly. Similar methods can be use to save on memory accesses. An alias analysis then must be used to refine the dependences between the set of memory accesses. Such analyses are described in Sect. 1.11.

An important aspect of the transformation PRE is that it supports the removal of loop-invariant code from loops. This will be considered in the next section.

1.14 Application: Moving Loop-Invariant Code

One important instance of a superfluous, repeated computation is an assignment that occurs in a loop and computes the same value on each iteration of the loop.

Example 1.14.1 Let us consider the following program:

```
for ($i \leftarrow 0; i < n; i{+}{+}$) {
        $T \leftarrow b + 3;$
        $a[i] \leftarrow T;$
}
```

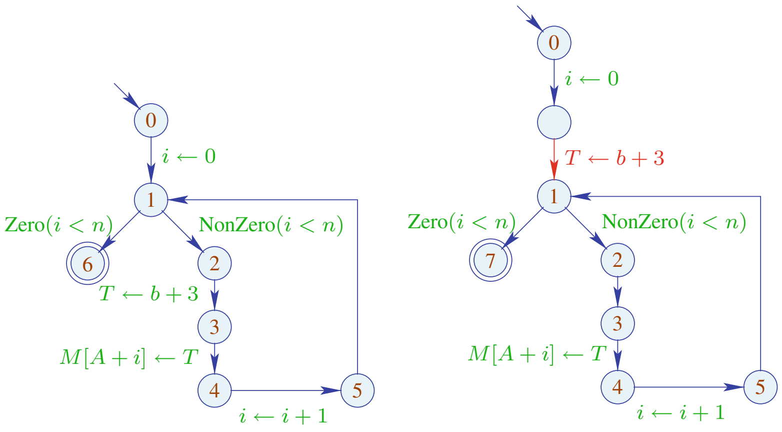


Fig. 1.51 A loop containing invariant code and an inadequate transformation

Figure 1.51 shows the control-flow graph of this program. The loop contains the assignment $T \leftarrow b + 3$, which computes the same value on each iteration of the loop and stores it in the variable T. Note that the assignment $T \leftarrow b + 3$ cannot be moved *before* the loop, as indicated in Fig. 1.51 because it would be executed in executions that would not enter the loop. □

This problem does not occur with *do-while* loops. In a *do-while* loop, loop-invariant code can be placed just infront of the loop. One way to avoid the placement problem with invariant code in *while* loops therefore is to transform them into *do-while* loops beforehand. The corresponding transformation is called *loop inversion*.

Example 1.14.2 Let us regard again the *while* loop of Example 1.14.1. The following Fig. 1.52 shows the inverted loop on the left side. The inverted loop has an extra occurrence of the loop condition guarding the first entry into the loop. Another occurrence of the loop condition is at the end of the loop. Figure 1.53 shows the analysis information for partial redundancy elimination. Figure 1.52 on the right shows the application of transformation PRE. The loop-invariant code has been moved to before the *do-while* loop. □

We conclude that transformation PRE is able to move loop-invariant code out of *do-while* loops. To treat *while* loops, these need to be transformed into *do-while* loops. This is straightforward in most imperative and object-oriented programming languages if the source code of the program is available. In C or JAVA, for example, the *while* loop:

while (b) *stmt*

can be replaced by:

if (b) **do** *stmt* **while** (b);

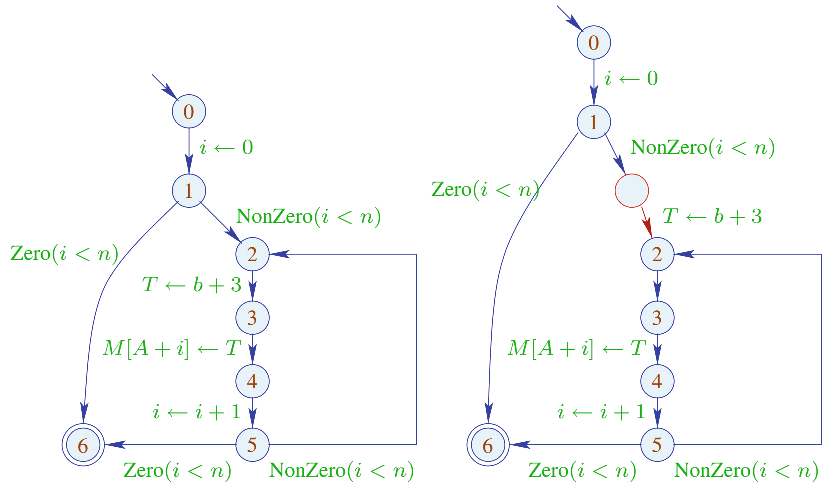


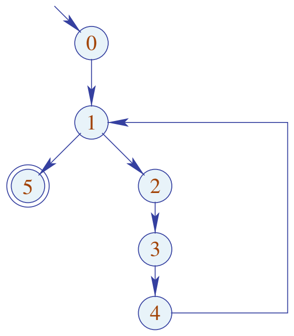
Fig. 1.52 The inverted loop of Example 1.14.1 and the result of moving invariant code out of the loop

	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{T \leftarrow b + 3\}$
3	$\{T \leftarrow b + 3\}$	$\emptyset$
4	$\{T \leftarrow b + 3\}$	$\emptyset$
5	$\{T \leftarrow b + 3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$

Fig. 1.53 The partial-redundancy analysis-information for the inverted loop of Fig. 1.52

However, often programs and programming-language constructs are intermediately represented in the more flexible form of control-flow graphs, in particular, if optimizations are applied which transform the control-flow graph in such a way that it cannot easily be mapped back to control constructs of the language. We should therefore identify loops by graph properties. We only consider the case of loops with a unique loop head and use the *predominator* relation between program points.

We say that a program point u *predominates* a program point v if each path π starting at the entry point of the program and reaching v passes through u. We write $u \Rightarrow v$. The relation $\Rightarrow$ is reflexive, transitive, and antisymmetric and therefore defines a partial order over the set of program points. This relation allows the compiler to discover *back edges* in the control-flow graph. An edge $k = (u, \_, v)$ is called a *back edge* if the target node v predominates the start node u of the edge.



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Fig. 1.54 The predominator sets for a simple control-flow graph

Example 1.14.3 Regard the example of the *while* loop on the left side of Fig. 1.51. Each program point of the loop body is predominated by program point 1. Edge (6, ; , 1) therefore is a back edge. □

We design a simple analysis to determine the set of predominators at the program points of programs. It collects the set of program points traversed along each path. The set of predominators for a program point v is obtained as the *intersection* of these sets. As complete lattice we therefore choose:

$$\mathbb{P} = 2^{Nodes}, \text{ with the partial order } \supseteq$$

We define as abstract edge effects:

$$[\![(u, lab, v)]\!]^\sharp \; P \quad = \quad P \cup \{v\}$$

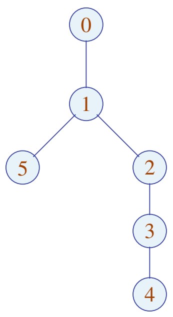
for all edges (u, lab, v). Note that the labels of the edges play no role. Instead, the analysis collects the *endpoints* of the edges. These abstract edge effects lead to the following set of predominators: $\mathcal{P}^*[v]$ at program point v:

$$\mathcal{P}[v] = \bigcap \{[\![\pi]\!]^\sharp \; \{start\} \mid \pi : start \rightarrow^* v\}$$

All abstract edges effects are distributive such that these sets can be determined as least solution of the associated system of inequalities.

Example 1.14.4 Regard the control-flow graph for the example program of Example 1.14.1. Figure 1.54 shows the associated predominator sets. Figure 1.55 shows the associated partial order $\Rightarrow$.

As usual in the representation of partial orders, only the *direct* relations are represented. Transitivity stays implicit. The result apparently is a tree! This is by no means an accident, as the following theorem shows. □



	$\mathcal{P}$
0	$\{0\leftarrow\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

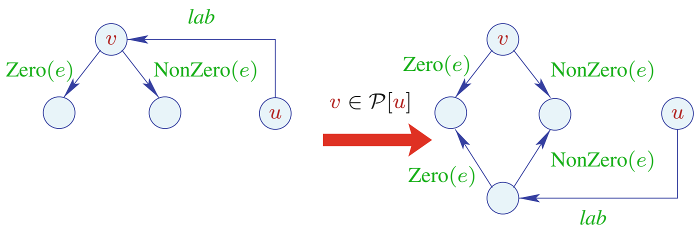
Fig. 1.55 The predominator relation for the control-flow graph of Fig. 1.54. The direction goes from *top* to *bottom*

Theorem 1.14.1 *Each program point v has at most one immediate predominator.*

Proof Assume that a program point v had two different direct predominators u_1, u_2. Neither $u_1 \Rightarrow u_2$ nor $u_2 \Rightarrow u_1$ can be true. Therefore, not every path from entry point of the program to u_1 nor every path from u_1 to v can contain program point u_2. Thus, there exists a path from the entry point to v that does not contain u_2, and u_2 cannot be a predominator of v. This is a contradiction to the assumption above. So, v has at most one direct predominator. This proves the theorem. □

The entry condition of the *while* loop is represented by a program point v with two outgoing condition edges. v predominates all nodes in the loop body, in particular the source node u of the back edge. Loop inversion consists in creating a new node with copies of the condition edges emanating from it to the targets of the original targets, and then redirecting the back edge from u towards the new node. This is the next transformation:

Transformation LR:



Loop inversion works for all *while* loops. There are, however, loops that cannot be inverted in this way. One such, somewhat unusual, loop is shown in Fig. 1.56. Unfortunately, there exist quite normal loops that cannot be inverted by transformation LR. Figure 1.57 shows one such loop. One would have to copy the whole path from the back edge to the condition together with the condition edges to invert such a loop.

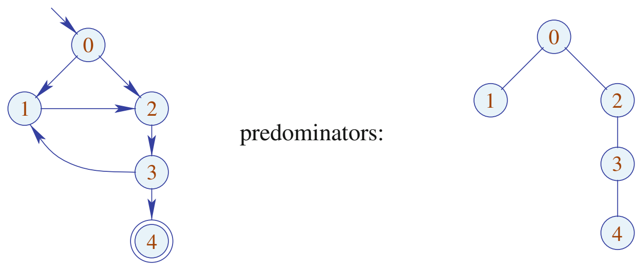


Fig. 1.56 A non-reversible loop

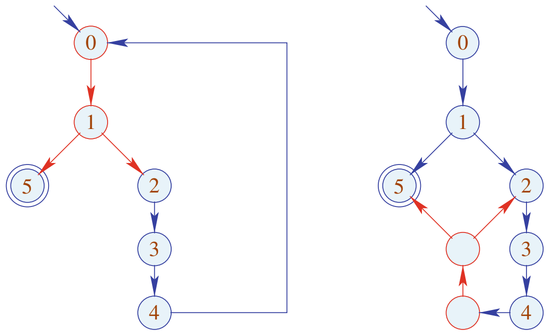


Fig. 1.57 A normal loop that cannot easily be reversed

This kind of loop can originate when a complex condition is evaluated in several steps.

1.15 Removal of Partially Dead Assignments

The removal of partial redundancy can be understood as a generalization of the optimization to avoid redundant computations. We now show how the removal of assignments to dead variables can also be generalized to a removal of assignments to *partially* dead variables.

Example 1.15.1 Consider the program of Fig. 1.58. The assignment $T \leftarrow x + 1$ needs only be computed on one of the two paths since variable x is dead along the other path. Such a variable is called *partially dead*.

Goal of the transformation is to delay the assignment $T \leftarrow x + 1$ as long as possible, that is, to move it forward along the control-flow path until the assignment is completely dead or certainly necessary. It is completely dead at a program point if the variable on the left side is dead at this program point. The desired result of this transformation is shown in Fig. 1.59. □

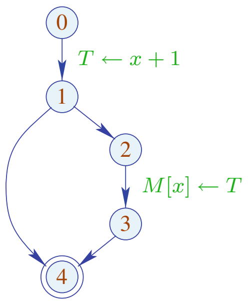


Fig. 1.58 An assignment to a partially dead variable

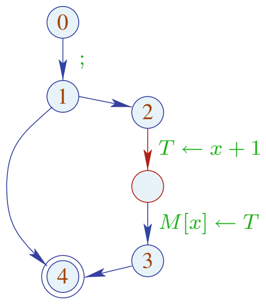


Fig. 1.59 The desired optimization for the program of Fig. 1.58

Delaying an assignment $x \leftarrow e$ must not change the semantics of the program. A sufficient condition for semantics preservation constrains the edges over which the assignment is shifted forward. None of those edges must change any variables occurring in the assignment; neither x nor any variable in $\mathsf{Vars}(e)$, and none of their labels may depend on the variable x. To guarantee profitability of the transformation we additionally require that when a merge point of the control flow is reached, i.e., when two edges meet, the assignment must be shifted over both edges to this merge point. To formalize these two requirements we define an analysis of *delayable assignments*. Like in the analysis of very busy assignments we use the complete lattice 2^{Ass} of all assignments $x \leftarrow e$, where x does not occur in e. The abstract edge effects remove those assignments that cannot be delayed over the edge and add the one that is computed at the edge if any. This latter assignment is a newly delayable assignment. The abstract edge effects are defined by:

$$[\![x \leftarrow e]\!]^\sharp\, D = \begin{cases} D\backslash(\mathsf{Ass}(e) \cup \mathsf{Occ}(x)) \cup \{x \leftarrow e\} & \text{if } x \notin \mathsf{Vars}(e) \\ D\backslash(\mathsf{Ass}(e) \cup \mathsf{Occ}(x)) & \text{if } x \in \mathsf{Vars}(e) \end{cases}$$

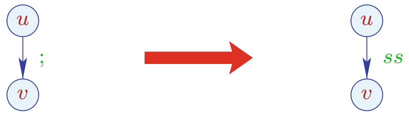
where $\mathsf{Ass}(e)$ is the set of assignments to variables, occurring in e and $\mathsf{Occ}(x)$ is the set of assignments in which x occurs. Using these conventions we define the rest of the abstract edge effects:

$$\begin{aligned}
[\![x \leftarrow M[e]]\!]^\sharp\, D &= D\backslash(\mathsf{Ass}(e) \cup \mathsf{Occ}(x))\\
[\![M[e_1] \leftarrow e_2]\!]^\sharp\, D &= D\backslash(\mathsf{Ass}(e_1) \cup \mathsf{Ass}(e_2))\\
[\![\mathsf{Zero}(e)]\!]^\sharp\, D = [\![\mathsf{NonZero}(e)]\!]^\sharp\, D &= D\backslash\mathsf{Ass}(e)
\end{aligned}$$

There are no delayable assignments at the entry point of the program. The initial value for the analysis therefore is $D_0 = \emptyset$. As partial order we choose the superset relation $\supseteq$ since an assignment can only be delayed up to a program point if it can be shifted there over all paths reaching that program point.

For the transformation rules to come we assume that $\mathcal{D}[\,.\,]$ and $\mathcal{L}[\,.\,]$ are the least solutions of the systems of inequalities for delayable assignments and the liveness of variables. Since we need the abstract edge effects of both analyses for the formulation of the applicability condition of the transformations, they are identified by the subscripts $\mathcal{D}$ and $\mathcal{L}$, respectively.

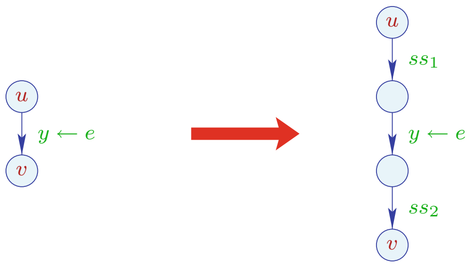
Transformation PDE for the Empty Statement:



This edge receives all assignments that cannot be moved beyond its target node v, but whose left side is live at v. The sequence of assignments, ss, to move there consists of all $x \leftarrow e' \in \mathcal{D}[u]\backslash\mathcal{D}[v]$ with $x \in \mathcal{L}[v]$.

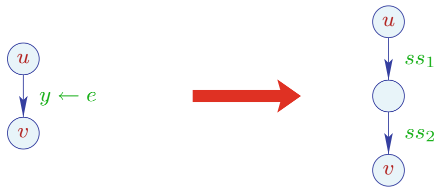
Transformation PDE for Assignments:

The assignment $y \leftarrow e$ cannot be moved if $y \in \mathsf{Vars}(e)$. In this case, the transformation is:



The sequence ss_1 collects those useful assignments that cannot be delayed beyond $y \leftarrow e$. The sequence ss_2 collects those useful assignments that can be delayed along this edge, but not beyond its target node. Therefore, ss_1 is a sequence consisting of the assignments $x \leftarrow e' \in \mathcal{D}[u] \cap (\mathsf{Ass}(e) \cup \mathsf{Occ}(y))$ with x in $\mathcal{L}[v]\backslash\{y\} \cup \mathsf{Vars}(e)$. Furthermore, ss_2 is a sequence consisting of the assignments $x \leftarrow e' \in \mathcal{D}[u]\backslash(\mathsf{Ass}(e) \cup \mathsf{Occ}(y) \cup \mathcal{D}[v])$ with $x \in \mathcal{L}[v]$.

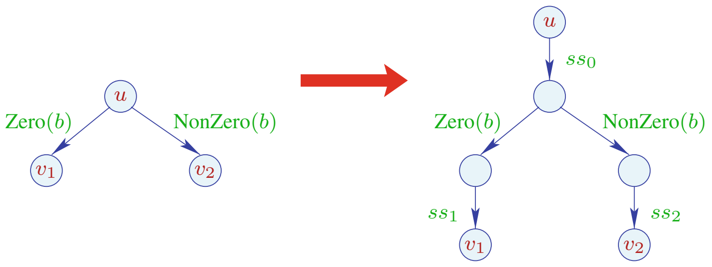
An assignment $y \leftarrow e$ satisfying $y \notin \mathsf{Vars}(e)$ can be delayed by the transformation:



The sequence ss_1 is defined in the same way as in the case of delayable assignments. The sequence ss_2 is defined analogously: It collects all useful assignments that can be delayed along the edge, but not beyond its target edge. Possibly, the sequence ss_2 could contain an occurrence of $y \leftarrow e$.

This means, ss_1 is a sequence formed out of assignments $x \leftarrow e' \in \mathcal{D}[u] \cap (\mathsf{Ass}(e) \cup \mathsf{Occ}(y))$ with $x \in \mathcal{L}[v] \backslash \{y\} \cup \mathsf{Vars}(e)$. Furthermore, ss_2 is a sequence of assignments $x \leftarrow e' \in (\mathcal{D}[u] \backslash (\mathsf{Ass}(e) \cup \mathsf{Occ}(y)) \cup \{y \leftarrow e\}) \backslash \mathcal{D}[v]$ with $x \in \mathcal{L}[v]$.

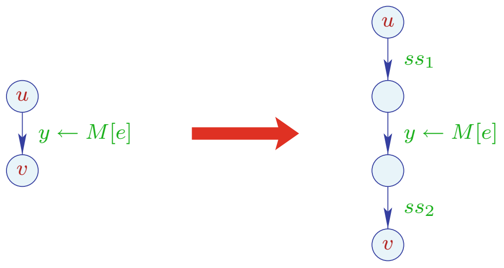
Transformation PDE for Conditional Branches:



The sequence ss_0 consists of all useful assignments that are delayable at u, but that cannot be delayed beyond the condition edges. The sequences ss_i, $i = 1, 2$, on the other hand, consist of all useful assignments that can be delayed beyond the condition edges, but not beyond the target node v_i.

This means that the sequence ss_0 consists of all assignments $x \leftarrow e \in \mathcal{D}[u]$ with $x \in \mathsf{Vars}(b)$, and the sequences ss_i for $i = 1, 2$, consist of all assignments with $x \leftarrow e \in \mathcal{D}[u] \backslash (\mathsf{Ass}(b) \cup \mathcal{D}[v_i])$ and $x \in \mathcal{L}[v_i]$.

Transformation PDE for Load Operations:

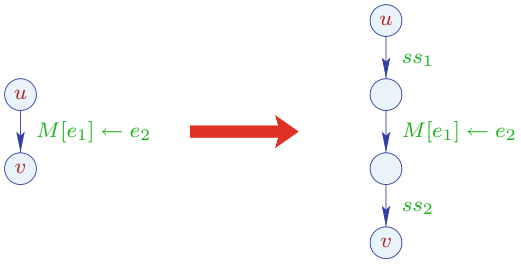


We do not present transformations that would delay load operations. Instead we treat them like nondelayable assignments. This means that the sequence ss_1 consists of assignments $x \leftarrow e' \in \mathcal{D}[u] \cap (\mathsf{Ass}(e) \cup \mathsf{Occ}(y))$ with $x \in \mathcal{L}[v] \backslash \{y\} \cup \mathsf{Vars}(e)$.

Furthermore, the sequence ss_2 consists of all assignments $x \leftarrow e' \in \mathcal{D}[u] \backslash (\mathsf{Occ}(y) \cup \mathsf{Ass}(e) \cup \mathcal{D}[v])$ with $x \in \mathcal{L}[v]$.

Transformation PDE for Write Operations:

The next transformation rule treats edges labeled with memory-write operations. These operations are not delayed.

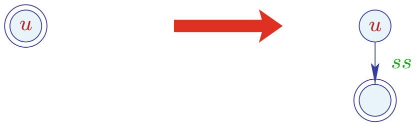


Again sequences ss_1 and ss_2 of assignments are required that are placed before and after the original statement, resp. The sequence ss_1 consists of the assignments $x \leftarrow e' \in \mathcal{D}[u] \cap (\mathsf{Ass}(e_1) \cup \mathsf{Ass}(e_2))$, and the sequence ss_2 of the assignments $x \leftarrow e' \in \mathcal{D}[u] \backslash (\mathsf{Ass}(e_1) \cup \mathsf{Ass}(e_2) \cup \mathcal{D}[v])$ with $x \in \mathcal{L}[v]$.

According to our assumption, a set X of variables is live at program exit. The last rule treats the case of a nonempty set X of live variables. Assignments to variables in X that are delayable at program exit need to be placed just before program exit. This is done by the following transformation rule:

Transformation PDE for Program Exit:

Let u be the exit point of the original program. Then a new program exit point is introduced:



Here ss is the set of assignments $x \leftarrow e$ in $\mathcal{D}[u]$ with $x \in X$.

Example 1.15.2 Let us return to our introductory example, and let us assume that no variable is live at program exit. No new program exit needs to be introduced in this case. The analyses for live variables and for delayable assignments result in:

	$\mathcal{L}$	$\mathcal{D}$
0	$\{x\}$	$\emptyset$
1	$\{x, T\}$	$\{T \leftarrow x + 1\}$
2	$\{x, T\}$	$\{T \leftarrow x + 1\}$
3	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$

The application of transformation PDE transforms the control-flow graph of Fig. 1.58 into the control-flow graph of Fig. 1.59. □

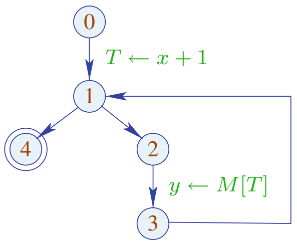


Fig. 1.60 A loop without delayable assignments

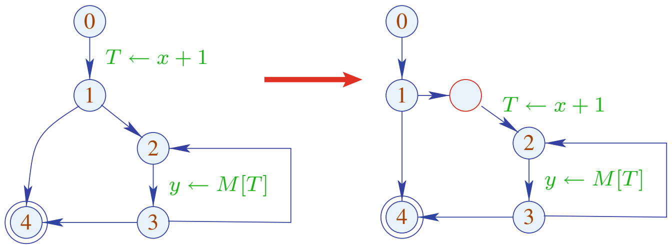


Fig. 1.61 The inverted loop and the removal of partially dead code

When we want to argue the correctness of transformation PDE we must take into account that some assignments are removed from the control-flow graph. The removed assignments are, however, only lost if their left sides are dead at the subsequent program point. Otherwise, they are remembered in the corresponding analysis information and are moved along the control-flow edges and reinserted into the control-flow graph when they can no longer be delayed.

An application of transformation PDE may remove some assignments and thus open new chances to remove newly partially dead code. Like with transformation PRE it can pay to apply this transformation repeatedly. One question is whether the transformation may not sometimes decrease the efficiency of the program, for example, by moving an assignment into a loop.

Example 1.15.3 Consider the loop of Fig. 1.60. The assignment $T \leftarrow x + 1$ is not delayable at any program point. This is different after the loop has been reversed as shown in Fig. 1.61. The assignment can now be moved past the loop-entry edge. This removes partially dead code. □

Transformation PDE did, in fact, *not* deteriorate the efficiency of the example program. In fact, it can be proved that it never decreases the efficiency of programs.

Conclusion 1.15.2 We have by now seen a number of optimizing program transformations. Several of these transformations may trigger another transformation applicable. For instance, transformation RE (removal of redundancies) may introduce inefficiencies, which may be removed by a subsequent application of transformation CE (copy elimination), followed by an application of DE (removal of assignments to dead variables). It is an interesting question in which order to apply optimizing transformations. Here is a meaningful order of the optimizing transformations we have described:

LR	Loop inversion
CF	Alias analysis
	Constant folding
	Interval analysis
RE	Removal of redundant computations
CE	Copy propagation
DE	Elimination of dead assignments
PRE	Removal of partially redundant assignments
PDE	Removal of partially dead assignments

1.16 Exercises

1. **Available assignments**
Regard the control-flow graph of the function swap of the introduction.

(a) Determine for each program point u the set $A[u]$ of assignments available at u.
(b) Apply transformation RE to remove redundant assignments.

2. **Complete lattices**
Consider the complete lattice M of monotonic boolean functions with two variables:

$$\begin{tikzcd}
& 1 \arrow[d, dash] & \\
& x \vee y \arrow[dl, dash] \arrow[dr, dash] & \\
x \arrow[dr, dash] & & y \arrow[dl, dash] \\
& x \wedge y \arrow[d, dash] & \\
& 0 &
\end{tikzcd}$$

(a) Determine the set of all monotonic functions that map M to the complete lattice $\mathbf{2} = \{0, 1\}$ with $0 < 1$.
(b) Determine the order on these functions.

3. **Complete lattices** Show the following claims:

(a) If $\mathbb{D}_1, \mathbb{D}_2$ are complete lattices then so is

$$\mathbb{D}_1 \times \mathbb{D}_2 = \{(x, y) \mid x \in \mathbb{D}_1, y \in \mathbb{D}_2\}$$

where $(x_1, y_1) \sqsubseteq (x_2, y_2)$ if and only if $x_1 \sqsubseteq x_2$ and $y_1 \sqsubseteq y_2$.

(b) A function $f : \mathbb{D}_1 \times \mathbb{D}_2 \to \mathbb{D}$ is monotonic if and only if the functions:

$$\begin{array}{lll} f_x : \mathbb{D}_2 \to \mathbb{D} & f_x(y) = f(x, y) & (x \in \mathbb{D}_1) \\ f^y : \mathbb{D}_1 \to \mathbb{D} & f^y(x) = f(x, y) & (y \in \mathbb{D}_2) \end{array}$$

are monotonic.

4. **Complete lattices**
For a complete lattice $\mathbb{D}$ let $h(\mathbb{D}) = n$ be the maximal length of a proper ascending chain $\bot \sqsubset d_1 \sqsubset \cdots \sqsubset d_n$. Show that for complete lattices $\mathbb{D}_1, \mathbb{D}_2$ it holds that:
   (a) $h(\mathbb{D}_1 \times \mathbb{D}_2) = h(\mathbb{D}_1) + h(\mathbb{D}_2)$
   (b) $h(\mathbb{D}^k) = k \cdot h(\mathbb{D})$
   (c) $h([\mathbb{D}_1 \to \mathbb{D}_2]) = \#\mathbb{D}_1 \cdot h(\mathbb{D}_2)$, where $[\mathbb{D}_1 \to \mathbb{D}_2]$ is the set of functions $f : \mathbb{D}_1 \to \mathbb{D}_2$ and $\#\mathbb{D}_1$ is the cardinality of $\mathbb{D}_1$.

5. **Introduction of temporary variables for expressions**
Introduce temporary variables T_e for given expressions e, such that the value of e is stored in T_e after each evaluation of e.
   (a) Define a program transformation that introduces these temporary variables. Argue that this transformation does not change the semantics of programs.
   (b) What influence does this transformation have on the removal of redundant computations RE?
   (c) For which expressions is this introduction of temporary variables profitable? How can the number of variable-to-variable copies be decreased by a subsequent transformation?
   (d) How can the transformation PRE profit from this introduction of temporary variables?

6. **Available memory reads**
Extend the analysis and transformation of available assignments such that the new analysis also determines the availability of memory reads $x \leftarrow M[e]$.

7. **Complete lattices**
Let $\mathcal{U}$ be a finite set and $\mathbb{D} = 2^{\mathcal{U}}$ be the powerset over $\mathcal{U}$, ordered by $\sqsubseteq = \subseteq$. Let $\mathcal{F}$ the set of all functions $f : \mathbb{D} \to \mathbb{D}$ of the form $f x = (x \cap a) \cup b$ with $a, b \subseteq \mathbb{D}$. Show:
   (a) $\mathcal{F}$ contains the identity function, and it has a least and a greatest element;
   (b) $\mathcal{F}$ is closed under composition, $\sqcup$, and $\sqcap$;
   (c) A (postfix-) operation * can be defined on $\mathcal{F}$ by:

$$f^* x = \bigsqcup_{j \geq 0} f^j x$$

8. **Fixed-point iteration**
   Consider a system of inequalities of the form:
   $$x_i \sqsupseteq f_i(x_{i+1}), \text{ where } f_i \text{is monotonic, for } i = 1, \ldots, n$$
   Show:
   (a) The fixed-point iteration terminates after at most n iterations.
   (b) One round of a round-robin iteration suffices given a suitable order of variables.
9. **Dead variables**
   Define a program analysis that determines for each program point the set of dead variables *directly*, that is, not as the complement of the set of live variables.
   (a) Define the associated lattice.
   (b) Define the associated abstract edge effects.
   (c) Extend the analysis to an analysis of *definite deadness*.

   How could one prove the correctness of the analysis?
10. **Distributivity I**
    Let $f_1, f_2 : \mathbb{D} \to \mathbb{D}$ be two distributive functions over a complete lattice $\mathbb{D}$. Show:
    (a) $f_1 \circ f_2$ is also distributive;
    (b) $f_1 \sqcup f_2$ is also distributive.
11. **Distributivity II**
    Prove Theorem 1.7.1.
12. **Distributivity III**
    Consider the function
    $$f(X) = (a \in X)?A : B$$
    with $A, B \subseteq U$ for some universe U.
    (a) Show that f is distributive both w.r.t. the ordering $\subseteq$ and the ordering $\supseteq$ on the set 2^U, whenever $B \subseteq A$.
    (b) Show for the ordering $\subseteq$ that f is completely distributive, if $B = \emptyset$.
    (c) Show for the ordering $\supseteq$ that f is completely distributive, if $A = U$.
13. **Optimization of function** swap
    Apply first the optimization RE, then the optimizations CE and DE to the example program swap!
14. **Constant propagation: signs**
    Simplify constant propagation such that it only considers the signs of values.
    (a) Define an appropriate partial order for this analysis.
    (b) Define the description relation Δ?
    (c) Define appropriate abstract operators on the abstract values.

(d) Prove that your operators respect the description relation Δ.
(e) Define abstract edge effects for the condition edges. Argue that these are correct.

15. **Constant propagation: excluded values**
Extend constant propagation in such a way that the new analysis determines not only definite values for variables, but also definitely excluded values.
Consider, e.g., a conditional:

$$\begin{array}{l}\textbf{if } (x = 3)\ y \leftarrow x;\\ \textbf{else } z \leftarrow x;\end{array}$$

Variable x can definitely not have the value 3 in the *else* part.

(a) Define an adequate partial order on values.
(b) Define the associated description relation Δ.
(c) Define meaningful abstract operators on the values.
(d) Prove that the abstract operators respect the description relation Δ.
(e) Strengthen the abstract edge effects for particular conditions and prove their correctness.

16. **Constant propagation: memory cells**
Extend constant propagation such that the contents of some memory cells are also tracked.

(a) Define the new abstract states.
(b) Define the new abstract edge effects for edges with load and store operations.
(c) Argue for the correctness of these new abstract edge effects.

17. **Stripes**
A generalization of constant propagation is obtained when sets of integers with more than one element are not abstracted just to the unknown value $\top$, but to a *common* linear progression. Such a progression is called a *stripe*. The single value 3 then could be described by the linear progression $3 + \lambda 0$, while the elements from the set $\{1, 3, 7\}$ all could be described by the linear progression $1 + 2\lambda$.
In general, the elements of the stripes domain are given by:

$$\{(a, b) \mid 0 \leq a < b\} \cup \{(a, 0) \mid a \in \mathbb{Z}\}$$

where the description relation Δ between integers and stripes is given by $z\ \Delta\ (a, b)$ iff $z = a + b\lambda$ for some $\lambda \in \mathbb{Z}$.

(a) Define a natural ordering $\sqsubseteq$ on stripes such that $z\ \Delta\ (a, b)$ implies $z\ \Delta\ (a', b')$ whenever $(a, b) \sqsubseteq (a', b')$.
(b) Show that the partial order of stripes has no infinite strictly ascending chains.
(c) Define a least upper bound operation on stripes and show that every *nonempty* set of stripes has a least upper bound.

(d) Define abstract versions of the arithmetic operators $+$ and $*$ for stripes.
(e) Use that to create an analysis in the style of constant propagation of the stripes.

18. **Interval operators**
Define the abstract operations ! (negation) and $\neq$ (inequality).
19. **Description relation for intervals**
Prove that the abstract multiplication for intervals respects the description relation Δ, i.e., show that $z_1 \; \Delta \; I_1$ and $z_2 \; \Delta \; I_2$ together imply that

$$(z_1 \cdot z_2) \; \Delta \; (I_1 \cdot^\sharp I_2)$$

20. **Interval analysis: refined widening**
Define a partial order on intervals that makes it possible to modify the lower as well as the upper bounds at most r times.
Define the description relation Δ for this abstract domain. Define a new widening.
21. **Interval analysis: termination**
Give an example program for which the interval analysis does not terminate without widening.
22. **Alias analysis**
Consider the following program:

```
for ($i \leftarrow 0; i < 3; i$++) {
    $R \leftarrow$ new();
    $R[1] \leftarrow i$;
    $R[2] \leftarrow l$;
    $l \leftarrow x$;
}
```

Apply the point-to analysis and the alias analysis of Sect. 1.11 to this program.
23. **Alias analysis: semantics**
In Sect. 1.11, we introduced an instrumented operational semantics to prove the correctness of the points-to analyses. Prove that this instrumented semantics is equivalent to the "natural" operational semantics for programs with dynamic allocation of memory blocks. To do this, formalize an equivalence notion that relates the different sets of addresses employed by the two semantics, though not globally, but for each concrete program execution.
24. **Alias analysis: number of iterations**
Show that the least fixed-point is already reached in exactly one iteration over all edges in the equality-based alias analysis of Sect. 1.11.
25. **Points-to analysis with Stripes**
The precision of point-to analyses can be improved by an accompanying analysis of stripes (see Exercise 17). considering blocks identified through abstract addresses not monolithically, but distiguishing an access $A[e_1]$ from an access $A[e_2]$ if the stripes corresponding to the index expressions e_1 and e_2 do not inter-

sect. Assume, e.g., that the stripe corresponding to e_1 is $(1, 2)$, while the stripe corresponding to e_2 is $(0, 4)$. Since for all $\lambda_1, \lambda_2 \in \mathbb{Z}$, $1 + 2\lambda_1 \neq 0 + 4\lambda_2$, these accesses definitely go to different memory locations.
Assume for the following, that for every program point v, an assignment $\mathcal{S}[v]$ from the *int* variables of the program to stripes is given. The goal is to design a refined points-to analysis which for each abstract address l, maintains a modulus b together with a mapping:

$$\mu_l^\sharp : \{0, \ldots, b - 1\} \to 2^{Val^\sharp}$$

(a) Define a partial ordering on such descriptions of heaps. How can a description relation be defined? What is the least upper bound operation between two such abstract heaps?
(b) Define abstract effects of the various statements. How do you interpret the *new* statement? What about reads from and writes to memory? Design accesses in such a way that the modulus b for an abstract location l is the gcd of all possible accesses to l.
(c) Use your analysis to infer more precise may alias information for expressions $A[e]$ and $A'[e']$ occurring at different program points.

26. **Worklist iteration**
Perform worklist iteration for the computation of available assignments for the factorial program. Determine the number of executed evaluations of right sides.
27. **Available memory look-ups**
Assume that we are given an equivalence relation $\equiv$ on pointer variables which equates variables R_1, R_2, if they may point to the same block in memory.

(a) Provide a refinement of availability analysis of expressions in variables to memory reads and memory writes, which takes the alias information into account.
(b) Use this analysis to extend redundancy elimination to memory operations.
(c) Apply your transformation to the body of the swap function from the introduction!
(d) Can your idea be extended also to partial redundancy elimination? How?
(e) Would your transformation benefit from stripes information (see Exercise 25)? Why?

28. **Loop-invariant code**
Perform code motion of loop-invariant code in the following program:

```
for ($i \leftarrow 0; i < n; i{+}{+}$) {
    $b \leftarrow a + 2;$
    $T \leftarrow b + i;$
    $M[T] \leftarrow i;$
    if ($j > i$) break;
}
```

Could the loop-invariant code also be moved if the condition **if** $(j > i) \ldots$ is located at the *beginning* of the loop body? Justify your answer.

29. **Loop-dominated programs**
    A program is called *loop dominated* if each loop has exactly one entry point, i.e., one program point that dominates all program points in the loop.
    (a) Prove that in loop-dominated programs the set of entry points of loops is a feedback vertex set of the control-flow graph.
    (b) Transform the loop in the example program for interval analysis into a *do-while* loop.
    (c) Perform interval analysis without narrowing on the transformed program. Compare the result with the result of Sect. 1.10.

1.17 Literature

The foundations of abstract interpretation were laid by Cousot and Cousot in 1977. Interval analysis is described for the first time in Cousot and Cousot (1976). This article also describes the widening and narrowing techniques. A precise interval analysis without widening and narrowing is described in Gawlitza and Seidl (2007). Monotone analysis frameworks are introduced by Kam and Ullman in 1976, 1977. Program analyses with distributive edge effects are considered by Kildall in 1973. Giegerich et al. develop the strengthening of liveness analysis to true liveness (1981).

The recursive fixed-point algorithm were formally proven correct in Coq (Hofmann et al. 2010a, b). The presentation of partial-redundancy elimination and partial dead-code elimination follows work by Knoop, Rüthing, and Steffen (Knoop et al. 1994a, b; Knoop 1998).

Karr (1976) and Granger (1991) present generalizations of constant propagation to the analysis of linear equalities. Their approaches are extended to interprocedural versions in Müller-Olm and Seidl (2004, 2005, 2007). Cousot and Halbwachs (1978) introduce an analysis of linear inequalities between variables. Practical applications in the analysis of C programs are extensively discussed by Simon in 2008.

Our rather brief overview only discusses very simple analyses of dynamic data structures. The alias-analysis problem for the programming language C is a real challenge, and is even more difficult if pointer arithmetic is considered. The simple methods described here follow the works of Steensgaard (1996) and Anderson et al. (2002). Fähndrich et al. present interprocedural extensions (2000) as do Liang et al. (2001). Ramalingam (2002) deals extensively with loop structures in control-flow graphs. He gives axiomatic and constructive definitions of loop structures. Sagiv et al. develop elaborated techniques for the analysis of program with dynamic memory allocation and linked structures in the heap (Sagiv et al. 1999, 2002). These analysis are of high complexity, but very powerful. They may automatically derive statements about the *shape* of linked data structures.

Chapter 2
Interprocedural Optimization

2.1 Programs with Procedures

In this chapter, we extend our programming language by procedures. Procedures have declarations and calls. Procedure calls have a complex semantics: The actual computation is *interrupted* at the call and *resumed* when the called procedure has terminated. The body of the procedure often forms a new scope for *local* names, i.e., names that are only visible within the body. Global names are possibly hidden by local names. Upon a procedure call, actual parameters are passed to formal parameters and, upon termination, results are passed back to the caller. Two different names for a variable coexist if the variable is passed to a reference parameter.

Our programming language has global and local variables. In this chapter, we use the naming convention that names of local variables begin with uppercase letters, while names of global variables begin with lowercase letters. Our interprocedural optimizations can only be applied if the called procedures can be statically determined. Therefore, we assume that at each call site the called procedure is explicitly given. In more expressive programming languages such as C, it is not always possible to determine statically which procedure is called, because the call goes indirectly through a function pointer. The same holds for object-oriented languages such as JAVA or C#, where the dynamic type of an object decides which method is actually called. For these languages an extension of the proposed techniques are required.

For simplicity, our programming language does not provide parameters for procedures or mechanisms for returning results. This is not as restrictive as it might seem, since call-by-value parameters as well as the passing of result values can be simulated through local and global variables, see Exercise 1. Since only procedures without parameters are considered, the only extension of the programming language as to statements, thus, is a procedure call, which has the form $\mathsf{f}()$.

A procedure f has a declaration:

$$\mathsf{f}()\ \{\ \mathit{stmt}^*\ \}$$

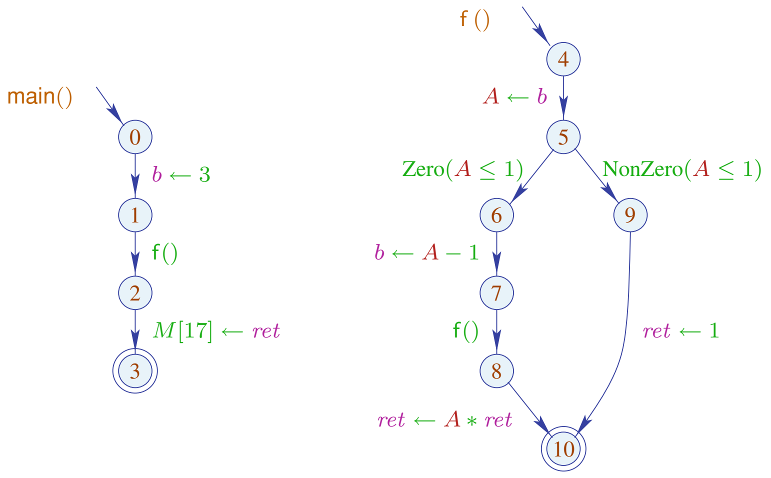


Fig. 2.1 The factorial program with its procedures

Each procedure f has one entry point, $start_{\mathsf{f}}$, to which control is passed from the caller, and one exit point, $stop_{\mathsf{f}}$, from which control is passed back to the caller. We forbid outgoing edges from exit points of procedures to ensure deterministic program execution. Program execution starts with the call to a dedicated procedure main ().

Example 2.1.1 We consider the factorial function, implemented by a procedure f and a procedure main calling f after setting the global variable b to 3.

```
main() {                     f() {
        b ← 3;                     A ← b;
        f();                       if (A ≤ 1) ret ← 1;
        M[17] ← ret;               else {
}                                          b ← A − 1;
                                           f();
                                           ret ← A · ret;
                                   }
                             }
```

The global variables b and ret hold the actual parameter and the return value of f, respectively. The formal parameter of f is simulated by the local variable A, which as first action in the body of procedure f receives the value of the actual parameter b. □

Programs in our extended programming language can be represented by sets of control-flow graphs, one control-flow graph per procedure. Figure 2.1 shows the two control-flow graphs for the program of Example 2.1.1.

2.2 Extended Operational Semantics

In order to argue about the correctness of programs and of program transformations, we need to extend our semantics of control-flow graphs to a semantics of control-flow graphs containing procedure calls. In the presence of procedure calls, executions of programs can no longer be described by paths, but need *nested paths* for their description. The actual nesting corresponds to the sequence of procedure calls. Each procedure call corresponds to an opening parenthesis, the return to the call site corresponds to the corresponding closing parenthesis. The sequence of already opened, but not yet closed parentheses can be represented by a stack, the *call stack*. Call stacks, therefore, form the foundation of the operational semantics of our program language with procedures.The operational semantics is defined by a one-step computation relation, $\vdash$, between configurations. A *configuration* is a triple consisting of a call stack, *stack*, a binding of the global variables, *globals*, and a contents of memory, *store*. A call stack consists of a sequence of *stack frames*, one for each entered but not yet terminated procedure. Each stack frame consists of a program point; this is the one in the associated procedure to which the computation has progressed, and a local state, that is, a binding of the local variables of the procedure:

$$\begin{array}{lcl}
\mathit{configuration} & == & \mathit{stack} \times \mathit{globals} \times \mathit{store} \\
\mathit{globals} & == & \mathit{Glob} \to \mathbb{Z} \\
\mathit{store} & == & \mathbb{N} \to \mathbb{Z} \\
\mathit{stack} & == & \mathit{frame} \cdot \mathit{frame}^* \\
\mathit{frame} & == & \mathit{point} \times \mathit{locals} \\
\mathit{locals} & == & \mathit{Loc} \to \mathbb{Z}
\end{array}$$

Glob and *Loc* denote the set of global and local variables of the program, and *point* denotes the set of program points.

Stacks grow from the bottom upwards in our graphical representation. The stack frame of the actual procedure is always on top. Sequences of stacks representing progress of a computation are listed from left to right. The stack of a caller is immediately to the left of the stack of the callee. The execution of the factorial program of Example 2.1.1 yields a sequence of call stacks (see Fig. 2.2).

Computations

The steps of a computation always refer to the actual procedure. In addition to the steps already known from programs without procedures, we need the following new cases:

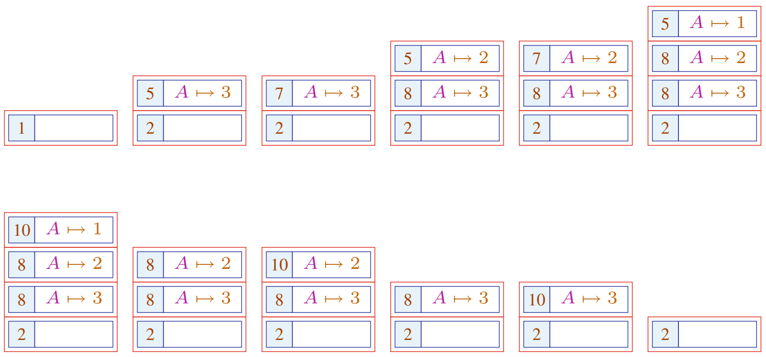


Fig. 2.2 A sequence of call stacks for the program of Example 2.1.1

$$\begin{array}{ll}
\text{call} \quad k = (u, \mathsf{f}(), v) \quad : & \\
(\sigma \cdot \boxed{(u, \rho_{Loc})}, \rho_{Glob}, \mu) & \vdash (\sigma \cdot \boxed{(v, \rho_{Loc}) \cdot (u_{\mathsf{f}}, \rho_{\mathsf{f}})}, \rho_{Glob}, \mu) \\
& \quad \mathit{start}_{\mathsf{f}} \text{ entry point of } \mathsf{f} \\
\text{return from a call :} & \\
(\sigma \cdot \boxed{(v, \rho_{Loc}) \cdot (r_{\mathsf{f}}, \_)}, \rho_{Glob}, \mu) & \vdash (\sigma \cdot \boxed{(v, \rho_{Loc})}, \rho_{Glob}, \mu) \\
& \quad \mathit{stop}_{\mathsf{f}} \text{ exit point of } \mathsf{f}
\end{array}$$

Mapping ρ_{f} binds the local variables to their values at the entry to procedure f. Program execution would be in principle nondeterministic if these values are left undefined and therefore could be arbitrary. To avoid this complication, we assume that local variables are always initialized to 0, i.e., $\rho_{\mathsf{f}} = \{x \mapsto 0 \mid x \in \mathit{Loc}\}$. This variable binding is given the name $\underline{0}$.

These two new transitions can be expressed by the two functions enter and combine, whose abstractions are the basis for interprocedural analysis. Both are applied to execution states of the program consisting of

1. a function ρ_{Loc} binding local variables to their values,
2. a function ρ_{Glob} binding global variables to their values, and
3. a function μ describing the contents of memory.

The effect of calling a procedure is described by the function enter. It computes the state after entry into the procedure from the state before the call. It is defined by:

$$\mathsf{enter}(\rho_{Loc}, \rho_{Glob}, \mu) = (\underline{0}, \rho_{Glob}, \mu)$$

Its arguments in an application are the execution state before the call.

The function combine determines the new state after the call by combining the relevant parts of the state before the call, namely the binding of the caller's local variables, with the effect the called procedure had on the global variables and the memory.

$$\mathsf{combine}((\rho_{Loc}, \rho_{Glob}, \mu), (\rho'_{Loc}, \rho'_{Glob}, \mu')) = (\rho_{Loc}, \rho'_{Glob}, \mu')$$

A sequence of computation steps, i.e., a *computation*, can equivalently represented by the sequence of edges as in the intraprocedural case, into which the subcomputations of calls are *embedded*. In order to indicate the nesting structure, we add two kinds of new labels, $\langle \mathsf{f} \rangle$ for the call to procedure f, and $\langle /\mathsf{f} \rangle$ for the exit from this procedure. In the example, we have the following sequence:

$$\begin{array}{l}
\langle \mathsf{main} \rangle \\
0, 1\ \langle \mathsf{f} \rangle\ 5, 6, 7 \\
\qquad \langle \mathsf{f} \rangle\ 5, 6, 7 \\
\qquad \langle \mathsf{f} \rangle\ 5, 6, 7 \\
\qquad \langle \mathsf{f} \rangle\ 5, 9, 10\ \langle /\mathsf{f} \rangle \\
\qquad\qquad 8, 10\ \langle /\mathsf{f} \rangle \\
\qquad\qquad 8, 10\ \langle /\mathsf{f} \rangle \\
\qquad\qquad 8, 10\ \langle /\mathsf{f} \rangle \\
2, 3\ \langle /\mathsf{main} \rangle
\end{array}$$

where we have listed only the traversed program points instead of the full edges (u, lab, v) to improve readability.

Same-Level Computations

A computation π that leads from a configuration $((u, \rho_{Loc}), \rho_{Glob}, \mu)$ to a configuration $((v, \rho'_{Loc}), \rho'_{Glob}, \mu')$ is called *same-level* since in such a computation each entered procedure is also exited, which means that the height of the stack at the end of the computation is exactly the height it had at the beginning. This means that the nested computation path contains for every opening parenthesis $\langle \mathsf{f} \rangle$ also the corresponding closing parenthesis $\langle /\mathsf{f} \rangle$. Since the same-level computation does not consult any stack frame in the stack below the initial one, it gives rise to computations leading from a configuration $(\sigma \cdot (u, \rho_{Loc}), \rho_{Glob}, \mu)$ to the configuration $(\sigma \cdot (v, \rho'_{Loc}), \rho'_{Glob}, \mu')$ for every call stack σ.

Let us assume that a same-level computation sequence leads from a configuration $((u, \rho_{Loc}), \rho_{Glob}, \mu)$ to a configuration $\big((v, \rho'_{Loc}), \rho'_{Glob}, \mu'\big)$. This means that this computation starts at program point u and reaches program point v, possibly with intervening procedure calls, if the variables have the values as given by $\rho = \rho_{Loc} \uplus \rho_{Glob}$ and the memory's content is described by μ. Here, $\uplus$ denotes the disjoint union of two functions. The value of the variables and the content of memory after the computation is described by $\rho' = \rho'_{Loc} \uplus \rho'_{Glob}$ and μ'. The same-level computation π therefore defines a partial function $[\![\pi]\!]$, which transforms $(\rho_{Loc}, \rho_{Glob}, \mu)$ into $(\rho'_{Loc}, \rho'_{Glob}, \mu')$. This transformation can be determined by induction over the structure of the computation:

$$[\![\pi\, k]\!] = [\![k]\!] \circ [\![\pi]\!] \qquad \text{for a normal edge } k$$
$$[\![\pi_1\, \langle \mathsf{f} \rangle\, \pi_2\, \langle /\mathsf{f} \rangle]\!] = H([\![\pi_2]\!]) \circ [\![\pi_1]\!] \qquad \text{for a procedure } \mathsf{f}$$

where the state transformation caused by the computation effected by a procedure's body is translated into one for the caller of that procedure by the operator $H(\cdots)$:

$$H(g)\ (\rho_{Loc}, \rho_{Glob}, \mu) = \mathsf{combine}\,((\rho_{Loc}, \rho_{Glob}, \mu), g\,(\mathsf{enter}\,(\rho_{Loc}, \rho_{Glob}, \mu)))$$

Besides the concept of same-level computation we also need the notion of u-*r*eaching computations. These start in a call of procedure main and lead to a call stack whose topmost stack frame contains program point u. Such computations can again be described as sequences of edges that contain labels $\langle \mathsf{f} \rangle$ and $\langle /\mathsf{f} \rangle$ for procedure entry and exit. Each such sequence is of the form:

$$\pi = \langle \mathsf{main} \rangle\, \pi_0\, \langle \mathsf{f}_1 \rangle\, \pi_1 \cdots \langle \mathsf{f}_k \rangle\, \pi_k$$

for procedures $\mathsf{f}_1, \ldots, \mathsf{f}_k$ and same-level computations $\pi_0, \pi_1, \ldots, \pi_k$. Each such sequence causes the transformation $[\![\pi]\!]$ of a triple $(\rho_{Loc}, \rho_{Glob}, \mu)$ consisting of variable bindings and a memory contents before the computation into a triple after the execution of π. We have

$$[\![\pi\, \langle \mathsf{f} \rangle\, \pi']\!] = [\![\pi']\!] \circ \mathsf{enter} \circ [\![\pi]\!]$$

for a procedure f and a same-level computation sequence π'.

Implementation of Procedures

The given operational semantics is closely related to an actual implementation of procedures. Therefore, the semantics is useful to estimate the *effort* needed by a procedure call.
Steps to be done before entering a procedure body:

- allocation of a stack frame;
- saving the local variables;
- saving the continuation address;
- jump to the code for the procedure body.

Steps to be taken upon termination of a procedure call:

- release of the stack frame;
- restoration of the local variables;
- jump to the continuation address.

Saving and restoring local variables is easy for a stack-based implementation. Realistic implementations on real machines, however, make use of the registers of the machine to have fast access to the values of local variables. The values of local variables of the caller may have to be stored to memory when control switches from the caller to the callee, and then reloaded into registers when control returns from the callee to the caller. While access to the values of local variables during the execution of a procedure body is fast, saving and restoring the values may require considerable effort.

2.3 Inlining

The first idea to reduce the overhead of procedure calls is to place a copy of the procedure's body at the call site. This optimization is called *inlining*.

Example 2.3.1 Consider the program:

```
abs () {                      max () {
        b_1 ← b;                      if (b_1 < b_2) ret ← b_2;
        b_2 ← −b;                     else ret ← b_1;
        max ();               }
}
```

Inlining the body of the procedure max() results in the program:

```
abs () {
  b_1 ← b;
  b_2 ← −b;
  ┌──────────────────────────┐
  │if (b_1 < b_2) ret ← b_2; │
  │else ret ← b_1;           │
  └──────────────────────────┘
}
```

□

The transformation Inlining for procedures is made simple by the simulation of parameter passing through the use of local and global variables. However, even the treatment of parameterless procedures offers some problems.

Inlining can only be done at call sites where the called procedure is statically known. This is not always the case as exemplified by programming languages such as C allowing indirect calls through function pointers. Object-oriented languages offer the problem that the method to call depends on the run-time type of the corresponding object. Additional analyses may be required to restrict the set of potentially called functions or methods. Inlining is only possible if the set of functions that can be called at this call site can be constrained to exactly one function.

Furthermore, it must be secured that the body of the inlined procedure does not modify the local variables of the calling procedure. This can be achieved by renam-

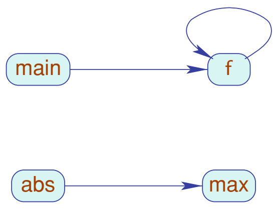


Fig. 2.3 The call graphs of Examples 2.1.1 and 2.3.1

ing the local variables of the called procedure. Still, there exists the threat of code explosion if a procedure is called multiple times. Even worse, complete inlining for recursive procedures does not terminate. Recursion must therefore be identified in procedure calls before inlining can be applied. This can be done using the *call graph* of the program. The nodes in this graph are the procedures of the program. An edge leads from a procedure p to a procedure q if the body of p contains a call to procedure q.

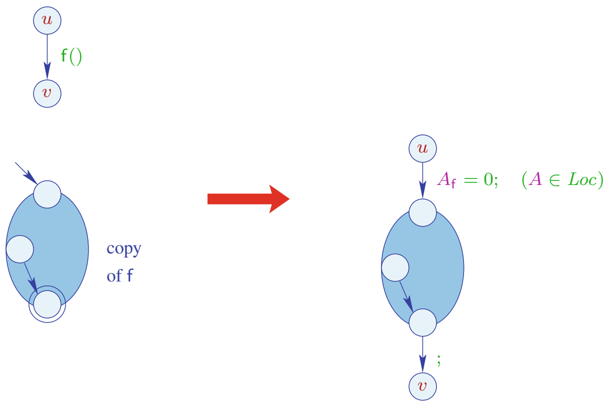
Example 2.3.2 The call graphs for the programs of Examples 2.1.1 and 2.3.1 are quite simple (see Fig. 2.3). In the first example, the call graph consists of the nodes main and f, and procedure main calls procedure f, which calls itself. In the second example, the call graph consists of the two nodes abs and max and the edge leading from abs to max. □

There are various strategies to decide for a procedure call whether to inline the procedure or not:

- Inline only *leaf* procedures; these are procedures without any calls.
- Inline at all *nonrecursive* call sites; these are call sites that are not part of strongly connected components of the call graph.
- Inline up to a certain fixed depth of nested calls.

Other criteria are possible as well. Once a procedure call is selected for inlining the following transformation is performed:

Transformation PI:



The edge labeled with the empty statement can be avoided if the *stop* node of f has no outgoing edges. Initializations of the local variables are inserted into the inlined procedure because our semantics requires these to receive initial values.

2.4 Tail-Call Optimization

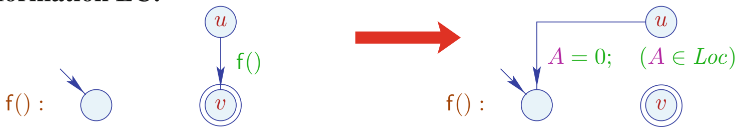
Consider the following example:

```
f () {
        if  ($b_2 \leq 1$)  ret ← $b_1$;
        else {
                $b_1 \leftarrow b_1 \cdot b_2$;
                $b_2 \leftarrow b_2 - 1$;
                f ();
        }
}
```

The last action to be performed in the body of the procedure is a call. Such a call is known as a *last* call. Last calls do not need a separate frame on the call stack. Instead, they can be evaluated in the same frame as the caller. For that, the local variables of the caller must be replaced by the local variables of the called procedure f. Technically, this means that the call of procedure f is replaced by an unconditional jump to the beginning of the body of f. If the last call is a recursive call then this *tail recursion* is tranformed into a loop, i.e., into *iteration*.In the example, this looks like:

```
f () {
    _f :  if  ($b_2 \leq 1$)  ret ← $b_1$;
          else {
                $b_1 \leftarrow b_1 \cdot b_2$;
                $b_2 \leftarrow b_2 - 1$;
                goto  _f;
          }
}
```

Transformation LC:



According to our semantics, the local variables must be initialized with 0 before the procedure's body can be entered.

Tail-call optimization is particularly important for declarative programming languages, which typically do not offer loops. Iterative computations have to be expressed by recursion, which is more expensive than iteration because of the allocation and deallocation of stack frames. An advantage of tail-call optimization over inlining is that it avoids code duplication. Last-call optimization can also be applied to nonrecursive tail calls. A somewhat disturbing feature of tail-call optimization is that it introduces jumps out of one procedure into another one, a concept that is shunned in modern high-level programming languages.

The reuse of the actual stack frame for the procedure called in a tail call is only possible if the local variables of the caller are no longer accessible. This is the case in our programming language. The programming language C, however, permits access to local variables at arbitrary positions in the call stack through pointers. Similar effects are possible if the programming language provides parameter passing *by-reference*. To apply tail-call optimization, the compiler then has to ensure that the local variables of the caller are indeed no longer accessible. An analysis determining sets of possibly accessible variables is the subject of Exercise 2.

2.5 Interprocedural Analysis

The analyses presented so far can only analyze single procedures. Applying such analyses to each procedure in a program has the advantage that complexity only grows linearly with the number of procedures. These techniques also work on separately compiled pieces of programs. The price to be paid is the limited precision that can be achieved. The analyses have little information at procedure boundaries and, therefore, must assume the worst.This means that typically no information about variables and data structures that the procedure might possibly access is available. For constant propagation, it means that only the values of local variables can be propagated. These are definitely not accessible from the outside.

However, with separate compilation, several related procedures may be compiled together since, for instance, they are defined in the same file or are parts of the same module or class. For this reason techniques are required that allow to analyze programs consisting of several procedures. We will, as an example, generalize an intraprocedural analysis to an interprocedural analysis. The considered example is copy propagation (described in Sect. 1.8). This analysis determines for a given variable x at each program point the set of variables that definitely contain the value last assigned to x. Such an analysis makes sense also across procedure boundaries.

Example 2.5.1 Consider the following example program:

```
main () {                        work () {
        A ← M[0];                        A ← b;
        if (A) print();                  if (A) work ();
        b ← A;                           ret ← A;
        work ();                 }
        ret ← 1 − ret;
}
```

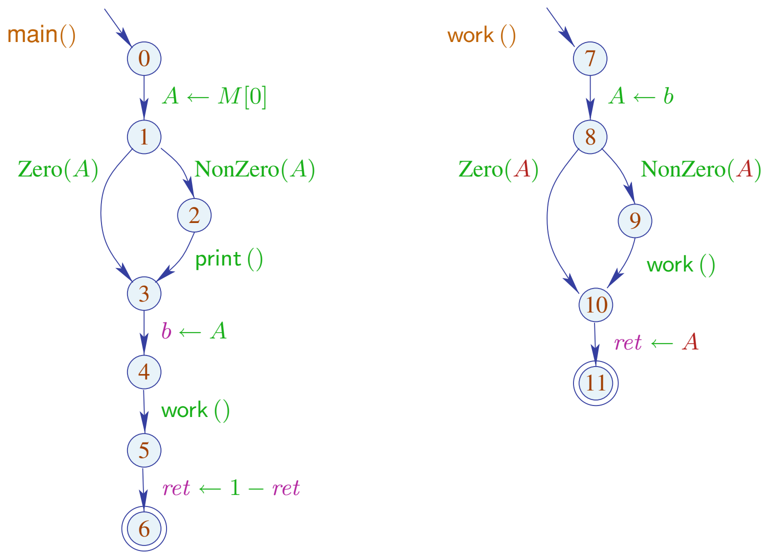


Fig. 2.4 The control-flow graphs for Example 2.5.1

Figure 2.4 shows the control-flow graphs associated with this program. Copying the value of global variable b into local variable A inside procedure work can be avoided. □

The problem with the generalization of the intraprocedural approach to copy propagation is that the interprocedural analysis no longer works on *one* control-flow graph, but needs to cope with the effects of possibly recursively called procedures.

2.6 The Functional Approach

Let us assume that we are given a complete lattice $\mathbb{D}$ of abstract states as potential analysis information at program points. The analysis performs a step along a normal edge k by applying to this information the abstract edge effect $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ corresponding to the edge label in the analysis. Now we have to deal with edges labeled with procedure calls.

The essential idea of the *functional* approach is to view the abstract edge effect of such a call also as a transformation of analysis information. The abstract edge effect of a call edge k is, thus, described by a function $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$. The difference to the abstract edge effects of normal edges is that this function is not known before the analysis is performed. It is only determined during the analysis of the body of the procedure.

To realize the analysis we need abstractions of the operations enter and combine of the operational semantics that are specific for the analysis to be designed. The abstract operation $\mathsf{enter}^\sharp$ initializes the abstract value for the entry point of the procedure using the analysis information available at the program point before the procedure call. The operation $\mathsf{combine}^\sharp$ combines the information obtained at the exit of the procedure body, its second argument, with the information available upon procedure entry, its first argument. These operations therefore have the functionality:

$$\begin{array}{ll} \mathsf{enter}^\sharp & : \ \mathbb{D} \to \mathbb{D} \\ \mathsf{combine}^\sharp & : \ \mathbb{D}^2 \to \mathbb{D} \end{array}$$

The overall abstract effect of a call edge k then is given by:

$$[\![k]\!]^\sharp\ D \ = \ \mathsf{combine}^\sharp\ (D, [\![\mathsf{f}]\!]^\sharp\ (\mathsf{enter}^\sharp\ \mathsf{D}))$$

if $[\![\mathsf{f}]\!]^\sharp$ is the transformation associated with the body of procedure f.

Interprocedural Copy Propagation

Let us consider how the functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ look in the analysis for copy propagation. The case of all variables being global is particularly simple: The function $\mathsf{enter}^\sharp$ is the identity function, that is, it returns its argument. The function $\mathsf{combine}^\sharp$ returns its second argument.

$$\mathsf{enter}^\sharp\ V = V \qquad \mathsf{combine}^\sharp\ (V_1, V_2) = V_2$$

Let us now consider an analysis of programs with local variables. The analysis should determine for each program point the set of variables containing the value last assigned to the global variable x.

During program execution, a copy of this value may be stored in a local variable of the caller of a procedure. This local variable is not visible inside the callee. If the callee assigns a new value to x the analysis cannot assume for any local variable of the caller that it (still) contains x's value. To determine whether such a recomputation may take place we add a new local variable $\bullet$; x's value before the call is recorded in it. $\bullet$ is not modified during the call. The analysis checks whether after return from the callee, $\bullet$ is guaranteed to still hold the value that x had before the call. Technically it means that function $\mathsf{enter}^\sharp$ adds the local variable $\bullet$ to the set of global variables that had the value of x before the call. After return from the callee, the local variables of the caller still contain the last computed value of x if $\bullet$ is contained in the set returned by the analysis of the callee. We define:

$$\begin{array}{ll} \mathsf{enter}^\sharp\ V & = V \cap \mathit{Glob} \cup \{\bullet\} \\ \mathsf{combine}^\sharp\ (V_1, V_2) & = (V_2 \cap \mathit{Glob}) \ \cup \ ((\bullet \in V_2)\ ?\ V_1 \cap \mathit{Loc}_\bullet : \emptyset) \end{array}$$

where $Loc_\bullet = Loc \cup \{\bullet\}$. The complete lattice used for the analysis of interprocedural copy propagation for a global variable x is:

$$\mathbb{V} = \{V \subseteq Vars_\bullet \mid x \in V\}$$

ordered by the superset relation, $\supseteq$, where $Vars_\bullet = Vars \cup \{\bullet\}$.

Abstract Edge Effects of Call Edges

In analogy to the concrete semantics we define, as a first step, for each complete lattice $\mathbb{D}$, for monotonic edge effects $[\![k]\!]^\sharp$, and for monotonic functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ the abstract transformation effected by a same-level computation:

$$\begin{array}{lll} [\![\pi\, k]\!]^\sharp & = [\![k]\!]^\sharp \circ [\![\pi]\!]^\sharp & \text{for a normal edge } k \\ [\![\pi_1 \langle \mathsf{f} \rangle\, \pi_2 \langle /\mathsf{f} \rangle]\!]^\sharp & = H^\sharp([\![\pi_2]\!]^\sharp) \circ [\![\pi_1]\!]^\sharp & \text{for a procedure } \mathsf{f} \end{array}$$

where the transformation

$$H^\sharp : (\mathbb{D} \to \mathbb{D}) \to \mathbb{D} \to \mathbb{D}$$

is defined by

$$H^\sharp\, g\, d = \mathsf{combine}^\sharp(d, g(\mathsf{enter}^\sharp(d)))$$

The abstract effect $[\![\mathsf{f}]\!]^\sharp$ for a procedure f should be an upper bound for the abstract effects $[\![\pi]\!]^\sharp$ of each same-level computation π for f, that is, each same-level computation starting at the entry point of f to its exit point. We use a system of inequalities over the lattice of all monotonic functions in $\mathbb{D} \to \mathbb{D}$ to determine or to approximate the effects $[\![\mathsf{f}]\!]^\sharp$.

$$\begin{array}{lll} [\![start_\mathsf{f}]\!]^\sharp & \sqsupseteq \mathsf{Id} & start_\mathsf{f} \text{ entry point of procedure } \mathsf{f} \\ [\![v]\!]^\sharp & \sqsupseteq H^\sharp([\![\mathsf{f}]\!]^\sharp) \circ [\![\mathsf{u}]\!]^\sharp & k = (u, \mathsf{f}(), v) \text{ call edge} \\ [\![v]\!]^\sharp & \sqsupseteq [\![k]\!]^\sharp \circ [\![u]\!]^\sharp & k = (u, lab, v) \text{ normal edge} \\ [\![\mathsf{f}]\!]^\sharp & \sqsupseteq [\![stop_\mathsf{f}]\!]^\sharp & stop_\mathsf{f} \text{ exit point of } \mathsf{f} \end{array}$$

The function $[\![v]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ for a program point v of a procedure f describes the effects of all same-level computations π that lead from the entry point of f to v. The expressions on the right side of the inequalities describe monotonic functions. Thus, the system of inequalities has a least solution. To prove the correctness of the approach one proves the following generalization of Theorem 1.6.1:

Theorem 2.6.1 *Let $[\![\,.\,]\!]^\sharp$ be the least solution of the interprocedural system of inequalities. We then have:*

1. $[\![v]\!]^\sharp \sqsupseteq [\![\pi]\!]^\sharp$ *for each same-level computation π leading from $start_\mathsf{f}$ to v, if v is a program point of procedure* f*;*

2. $[\![\mathsf{f}]\!]^\sharp \sqsupseteq [\![\pi]\!]^\sharp$ *for each same-level computation* π *of procedure* f. □

Theorem 2.6.1 can be proved by induction over the structure of same-level computations π. The theorem guarantees that each solution of the system of inequalities can be used to approximate the abstract effects of procedure calls.

Problems of the Functional Approach

There are two fundamental problems connected to this approach. First, the monotonic functions in $\mathbb{D} \to \mathbb{D}$ need to be effectively represented. However, functions occurring here do not always have simple representations. In the case of finite complete lattices $\mathbb{D}$ all occurring functions can, at least in principle, be represented by their value tables. This is no longer possible if the complete lattice $\mathbb{D}$ is infinite as is the case for constant propagation. In this case a further complication may arise, namely that of infinite ascending chains, which never stabilize.

Let us return to our example of interprocedural copy propagation. In this case, the following observation helps: The complete lattice $\mathbb{V}$ is *atomic*. The set of atomic elements in $\mathbb{V}$ is given by:

$$\{\mathit{Vars}_\bullet \backslash \{z\} \mid z \in \mathit{Vars}_\bullet \backslash \{x\}\}$$

Furthermore, all occurring abstract edge effects are not only monotonic, but are even distributive with respect to $\supseteq$. Instead of considering all monotonic functions in $\mathbb{V} \to \mathbb{V}$ it suffices to calculate in the sublattice of distributive functions.

Distributive functions over an atomic lattice $\mathbb{D}$ have compact representations. Let $A \subseteq \mathbb{D}$ be the set of atomic elements in $\mathbb{D}$. According to Theorem 1.7.1 in Sect. 1.7 each distributive function $g : \mathbb{D} \to \mathbb{D}$ can be represented as

$$g(V) = b \sqcup \bigsqcup \{h(a) \mid a \in A \wedge a \sqsubseteq V\}$$

for a function $h : A \to \mathbb{D}$ and an element $b \in \mathbb{D}$.

Each distributive function g for the propagation of copies can therefore be represented by at most k sets where k is the number of variables of the program. Thereby, the height of the complete lattice of distributive functions is bounded by k^2.

Example 2.6.1 Let us consider the program of Example 2.5.1. The set $\mathit{Vars}_\bullet$ is given as $\{A, b, \mathit{ret}, \bullet\}$. Assume that we want to analyze copy propagation for the global variable b. The functions

$$\begin{array}{rcll} [\![A \leftarrow b]\!]^\sharp C &=& C \cup \{A\} & =: g_1(C) \\ [\![\mathit{ret} \leftarrow A]\!]^\sharp C &=& (A \in C)\,?\,(C \cup \{\mathit{ret}\}) : (C \backslash \{\mathit{ret}\}) & =: g_2(C) \end{array}$$

correspond to the assignments $A \leftarrow b$ and $\mathit{ret} \leftarrow A$. The two abstract edge effects g_1, g_2 can be represented by the two pairs $(h_1, \mathit{Vars}_\bullet)$ and $(h_2, \mathit{Vars}_\bullet)$ with

	h_1	h_2
$\{b, ret, \bullet\}$	$Vars_\bullet$	$\{b, \bullet\}$
$\{b, A, \bullet\}$	$\{b, A, \bullet\}$	$Vars_\bullet$
$\{b, A, ret\}$	$\{b, A, ret\}$	$\{b, A, ret\}$

In a first round of a round-robin iteration, program point 7 is associated with the identity function; program points 8, 9, and 10 with the function g_1; and program point 11 with the composition $g_2 \circ g_1$. This last function is given by:

$$g_2(g_1(C)) = C \cup \{A, ret\} =: g_3(C)$$

It delivers the first approximation for the body of function work. Thereby, we obtain for a call to function work:

$$\mathsf{combine}^\sharp(C, g_3(\mathsf{enter}^\sharp(C))) = C \cup \{ret\} =: g_4(C)$$

In the second round of a round-robin iteration, the new value at program point 10 is the function $g_1 \cap g_4 \circ g_1$ where

$$\begin{array}{rll} g_4(g_1(C)) = C \cup \{A, ret\} & & \text{and consequently:} \\ g_1(C) \cap g_4(g_1(C)) = C \cup \{A\} & & = g_1(C) \end{array}$$

as in the last iteration. In this example, the fixed point is reached in the first iteration. □

An Interprocedural Coincidence Theorem

A coincidence theorem similar to 1.6.3 can be proved for analyses with distributive edge effects. An additional condition needed for this generalization is that the edge effect $[\![k]\!]^\sharp$ of a call edge $k = (u, \mathsf{f}(), v)$ is distributive. This leads to the following theorem.

Theorem 2.6.2 *Let us assume that for each procedure* f *and each program point v of* f *there exists at least one same-level computation from the entry point $start_\mathsf{f}$ of the procedure* f *to v. Let us further assume that all edge effects $[\![k]\!]^\sharp$ of normal edges as well as the transformations $H^\sharp$ are distributive. That is, in particular,*

$$H^\sharp(\bigsqcup \mathcal{F}) = \bigsqcup \{H^\sharp(g) \mid g \in \mathcal{F}\}$$

for each nonempty set $\mathcal{F}$ of distributive functions. We have then for each procedure f *and each program point v of* f*,*

$$[\![v]\!]^\sharp = \bigsqcup \{[\![\pi]\!]^\sharp \mid \pi \in \mathcal{T}_v\}$$

Here $\mathcal{T}_v$ is the set of all same-level computations from the entry point $start_{\mathsf{f}}$ of procedure f *to* v.

The proof of this theorem is a generalization of the proof of the corresponding theorem for intrarocedural analyses. To easily apply Theorem 2.6.2, we need a class of functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ as large as possible with the property that transformation $H^\sharp$ becomes distributive. We observe:

Theorem 2.6.3 *Let* $\mathsf{enter}^\sharp : \mathbb{D} \to \mathbb{D}$ *be distributive and* $\mathsf{combine}^\sharp : \mathbb{D}^2 \to \mathbb{D}$ *have the form:*

$$\mathsf{combine}^\sharp(x_1, x_2) = h_1(x_1) \sqcup h_2(x_2)$$

for two distributive functions $h_1, h_2 : \mathbb{D} \to \mathbb{D}$. *Then* $H^\sharp$ *is distributive, that is, it holds that*

$$H^\sharp(\bigsqcup \mathcal{F}) = \bigsqcup \{H^\sharp(g) \mid g \in \mathcal{F}\}$$

for each nonempty set $\mathcal{F}$ *of distributive functions.*

Proof Let $\mathcal{F}$ be a nonempty set of distributive functions. We then have:

$$\begin{aligned} H^\sharp(\bigsqcup \mathcal{F}) &= h_1 \sqcup h_2 \circ (\bigsqcup \mathcal{F}) \circ \mathsf{enter}^\sharp \\ &= h_1 \sqcup h_2 \circ (\bigsqcup \{g \circ \mathsf{enter}^\sharp \mid g \in \mathcal{F}\}) \\ &= h_1 \sqcup (\bigsqcup \{h_2 \circ g \circ \mathsf{enter}^\sharp \mid g \in \mathcal{F}\}) \\ &= \bigsqcup \{h_1 \sqcup h_2 \circ g \circ \mathsf{enter}^\sharp \mid g \in \mathcal{F}\} \\ &= \bigsqcup \{H^\sharp(g) \mid g \in \mathcal{F}\} \end{aligned}$$

The second equation holds because composition $\circ$ is distributive in its first argument, and the third equation holds since composition is distributive in its second argument, provided the first argument is a distributive function. □

Let us recall the definitions of the two functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ for the propagation of copies. We had:

$$\begin{aligned} \mathsf{enter}^\sharp\, V &= V \cap Glob \cup \{\bullet\} \\ \mathsf{combine}^\sharp\, (V_1, V_2) &= (V_2 \cap Glob) \cup (\bullet \in V_2)\, ?\, V_1 \cap Loc : \emptyset \\ &= ((V_1 \cap Loc_\bullet) \cup Glob)\ \cap \\ &\quad\ ((V_2 \cap Glob) \cup Loc_\bullet) \cap (Glob \cup (\bullet \in V_2)\, ?\, Vars_\bullet : \emptyset) \end{aligned}$$

The function $\mathsf{enter}^\sharp$ therefore is distributive. Likewise, the function $\mathsf{combine}^\sharp$ can be represented as the intersection of a distributive function of the first argument with a distributive function of the second argument. Theorem 2.6.3 can therefore be applied for the order $\supseteq$. We conclude that the transformation $H^\sharp$ for copy propagation is distributive for distributive functions. Therefore, the interprocedural coincidence theorem (Theorem 2.6.2) holds.

2.7 Interprocedural Reachability

Let us assume that we have determined, in a first step, the abstract effects $[\![f]\!]^\sharp$ of the bodies of procedures f, or at least safely approximated them. In a second step, we would like to compute for each program point u a property $\mathcal{D}[u] \in \mathbb{D}$ that is guaranteed to hold whenever program execution reaches the program point u. We construct a system of inequalities for this analysis problem:

$$\begin{array}{lll}
\mathcal{D}[start_{\mathsf{main}}] & \sqsupseteq \mathsf{enter}^\sharp(d_0) & \\
\mathcal{D}[start_{\mathsf{f}}] & \sqsupseteq \mathsf{enter}^\sharp(\mathcal{D}[u]) & (u, \mathsf{f}(), v) \text{ call edge} \\
\mathcal{D}[v] & \sqsupseteq \mathsf{combine}^\sharp(\mathcal{D}[u], [\![\mathsf{f}]\!]^\sharp(\mathsf{enter}^\sharp(\mathcal{D}[\mathsf{u}]))) & (u, \mathsf{f}(), v) \text{ call edge} \\
\mathcal{D}[v] & \sqsupseteq [\![k]\!]^\sharp(\mathcal{D}[u]) & k = (u, lab, v) \text{ normal edge}
\end{array}$$

where $d_0 \in \mathbb{D}$ is the analysis information assumed before program execution.

All right sides are monotonic. So, the system of inequalities has a least solution. To prove the correctness of our approach we define the abstract effect $[\![\pi]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ for each v-reaching computation π. This is in analogy to the concrete semantics. The following theorem relates the abstract effects of the v-reaching computations to the abstract values $\mathcal{D}[v]$ obtained by solving the system of inequalities.

Theorem 2.7.1 *Let $\mathcal{D}[\,.\,]$ be the least solution of the interprocedural system of inequalities given above. We then have:*

$$\mathcal{D}[v] \;\sqsupseteq\; [\![\pi]\!]^\sharp\, d_0$$

for each v-reaching computation. □

Example 2.7.1 Let us regard again the program of Example 2.5.1. We obtain for program points 0 to 11:

0	$\{b\}$
1	$\{b\}$
2	$\{b\}$
3	$\{b\}$
4	$\{b\}$
5	$\{b, ret\}$
6	$\{b\}$

7	$\{b, \bullet\}$
8	$\{b, A, \bullet\}$
9	$\{b, A, \bullet\}$
10	$\{b, A, \bullet\}$
11	$\{b, A, \bullet, ret\}$

We conclude that the global variable b can be used instead of the local variable A inside procedure work. □

If all program points are reachable the second phase of the interprocedural analysis also satisfies a conincidence theorem.

Theorem 2.7.2 *Let us assume that for each program point v there exists at least one v-reaching computation. Let us further assume that all effects $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$*

of normal edges as well as the transformation $H^\sharp$ are distributive. Then we have for each program point v,

$$\mathcal{D}[v] = \bigsqcup\{[\![\pi]\!]^\sharp d_0 \mid \pi \in \mathcal{P}_v\}$$

Here, $\mathcal{P}_v$ is the set of all v-reaching computations. □

2.8 Demand-Driven Interprocedural Analysis

In many practical cases, the complete lattice $\mathbb{D}$ is finite, and the required monotonic functions in $\mathbb{D} \to \mathbb{D}$ are compactly representable. This holds in particular for the interprocedural analysis of available assignments, very busy assignments (see Exercises 4 and 5, respectively) or copy propagation. However, one would like to have interprocedural analyses for relevant problems where either the complete lattice is infinite, or the abstract effects do not have compact representations. In these cases, an observation can be exploited made very early by Patrick Cousot as well as by Micha Sharir and Amir Pnueli. Procedures are often only called in constellations that can be classified into a few abstract constellations. In theses cases it may suffice to analyze only a few abstract calls of each procedure whose values are really needed.

To elaborate this idea, we introduce unknowns $\mathcal{D}[\mathsf{f}, \mathsf{a}]$ (f a procedure, $a \in \mathbb{D}$) which should receive the value of the abstract effect of the body of the procedure f, given that it has been entered in the abstract state a. Thus, it corresponds to the value $[\![\mathsf{f}]\!]^\sharp\,\mathsf{a}$. In order to determine the value of the unknown $\mathcal{D}[\mathsf{f}, \mathsf{a}]$, we introduce a distinct unknown $\mathcal{D}[v, a]$ of every program point v of the procedure f and at least conceptually, set up the following system of constraints:

$$\begin{array}{lll}
\mathcal{D}[v, a] & \sqsupseteq a & v \ \text{ entry point} \\
\mathcal{D}[v, a] & \sqsupseteq \mathsf{combine}^\sharp\,(\mathcal{D}[u, a], \mathcal{D}[\mathsf{f}, \mathsf{enter}^\sharp(\mathcal{D}[\mathsf{u}, \mathsf{a}])]) & \\
 & & (u, \mathsf{f}(), v) \text{ call edge} \\
\mathcal{D}[v, a] & \sqsupseteq [\![lab]\!]^\sharp\,(\mathcal{D}[u, a]) & k = (u, lab, v) \text{ normal edge} \\
\mathcal{D}[\mathsf{f}, \mathsf{a}] & \sqsupseteq \mathcal{D}[stop_\mathsf{f}, a] & stop_\mathsf{f} \text{ exit point of } \mathsf{f}
\end{array}$$

The unknown $\mathcal{D}[v, a]$ denotes the abstract state when the program point v within a procedure is reached that was called in the abstract state a. It corresponds to the value $[\![v]\!]^\sharp\, a$ that our analysis of abstract effects of procedures would compute for program point v.Note that in this system of inequalities modeling of procedure calls creates *nested* unknowns; the value of the inner unknown influences for which second component b the value of the variable $\mathcal{D}[\mathsf{f}, \mathsf{b}]$ is queried. This indirect addressing causes the variable dependences to change dynamically during the fixed-point iteration. The variable dependences are, thus, no longer statically known.

This system of inequalities is, in general, very large. Each program point is copied as often as there are elements in $\mathbb{D}$! On the other hand, we do not want to solve it completely. Only the values for those calls should be determined that really *occur*, that is, whose values are demanded during the analysis. Technically, this means that the analysis should constrain itself to those unknowns whose values are accessed during the fixed-point iteration when computing the value $\mathcal{D}[\mathsf{main}, \mathsf{enter}^\sharp(d_0)]$.

The demand-driven evaluation of a system of inequalities needs an adequate fixed-point algorithm. The *local* fixed-point algorithm of Sect. 1.12 can be applied here! It explores the variable space according to the possibly dynamic dependences between variables. Assume that the initial abstract state for program entry is given by $d_0 \in \mathbb{D}$. When started with an initial query for the value of $\mathcal{D}[\mathsf{main}, \mathsf{enter}^\sharp d_0]$, the local fixed-point iteration will explore every procedure f only for that subset of abstract states which are queried when trying to compute the value for the procedure exit of main, when called with the abstract value $\mathsf{enter}^\sharp d_0$.

Let us demonstrate the demand-driven variant of the functional approach with an example analysis. We choose the interprocedural constant propagation. The complete lattice for this analysis is as before:

$$\mathbb{D} = (Vars \to \mathbb{Z}^\top)_\bot$$

This complete lattice is of finite height, but not finite. Interprocedural constant propagation might, therefore, not terminate. The functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ for constant propagation are given by:

$$\begin{array}{ll}
\mathsf{enter}^\sharp\ D & = \begin{cases} \bot & \text{if } D = \bot \\ D \oplus \{A \mapsto \top \mid A \text{ lokal}\} & \text{otherwise} \end{cases} \\
\mathsf{combine}^\sharp(D_1, D_2) & = \begin{cases} \bot & \text{if } D_1 = \bot \vee D_2 = \bot \\ D_1 \oplus \{b \mapsto D_2(b) \mid b \text{ global}\} & \text{otherwise} \end{cases}
\end{array}$$

Together with the intraprocedural abstract edge effects for constant propagation we obtain an analysis that terminates for lattices of finite height $\mathbb{D}$ if and only if the fixed-point algorithm needs to consider only finitely many unknowns $\mathcal{D}[v, a]$ and $\mathcal{D}[f, a]$.

Example 2.8.1 Let us consider a slight modification of the program of Example 2.5.1 with the control-flow graphs of Fig. 2.5. Let d_0 be the variable binding:

$$d_0 = \{A \mapsto \top, b \mapsto \top, ret \mapsto \top\}$$

This leads to the following sequence of evaluations:

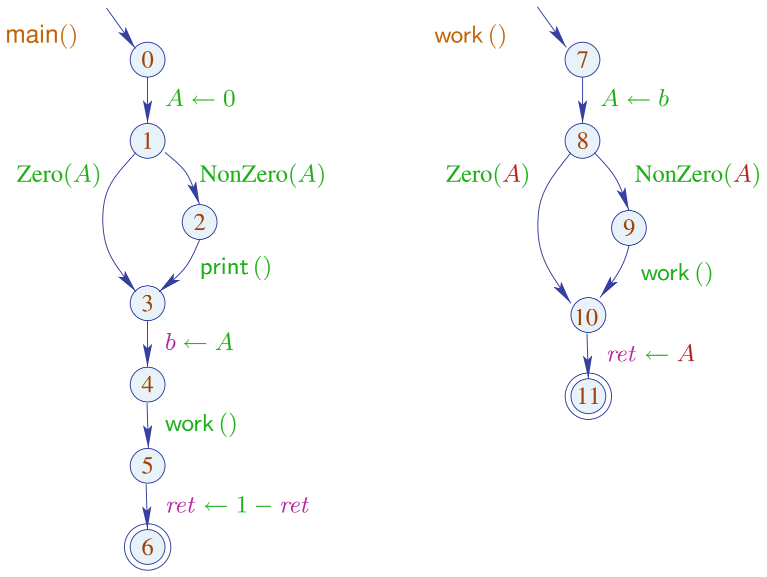


Fig. 2.5 The control-flow graphs for Example 2.8.1

	A	b	ret
$0, d_0$	$\top$	$\top$	$\top$
$1, d_0$	0	$\top$	$\top$
$2, d_0$		$\perp$	
$3, d_0$	0	$\top$	$\top$
$4, d_0$	0	0	$\top$
$7, d_1$	$\top$	0	$\top$
$8, d_1$	0	0	$\top$
$9, d_1$		$\perp$	
$10, d_1$	0	0	$\top$
$11, d_1$	0	0	0
$5, d_0$	0	0	0
$6, d_0$	0	0	1
main, d_0	0	0	1

for $d_1 = \{A \mapsto \top, b \mapsto 0, ret \mapsto \top\}$. The right side of each unknown is evaluated at most once in this example. □

In the example, the analysis terminates after only one iteration. Only one copy needs to be considered for each program point. In general, the analysis needs to calculate with several copies. The lattice does not have infinite ascending chains. Thus, the analysis terminates if each procedure is called only with finitely many arguments during the iteration.

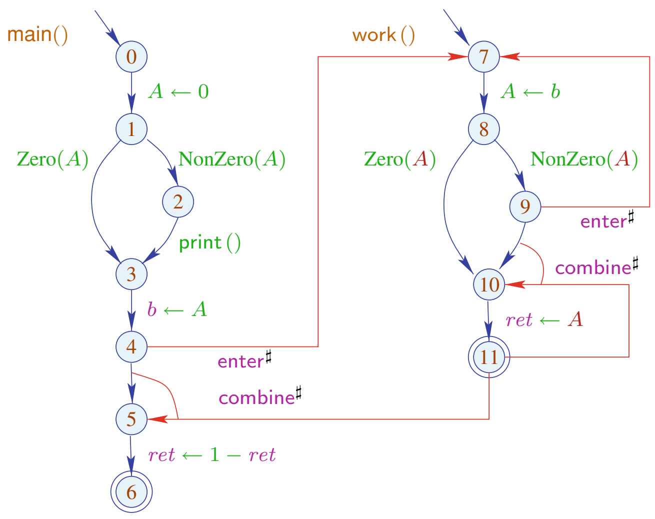


Fig. 2.6 The interprocedural supergraph for the example of Fig. 2.4

2.9 The Call-String Approach

An alternative approach for interprocedural static analysis uses an abstraction of a stack-based operational semantics. The goal is to determine properties of the procedures' behaviors differentiated by the set of *call strings* corresponding to reachable run-time stacks. Here, the call string of a run-time stack retains the sequence of procedures but ignores the values of locals. In general, the set of all reachable run-time stacks as well as the set of potential call strings will be infinite. The trick is to keep distinct information associated with call strings up to a fixed depth d and summarize the property for call strings underneath the top d elements. This idea was presented in Sharir and Pnueli's seminal work.

The complexity of this approach increases drastically with the depth d. In practical applications, call strings of length 1 or even 0 are often used. Using stack depth 0 means to approximate the entry into a procedure f as an *unconditional* jump to the beginning of procedure f and the exit of procedure f as a return jump to the continuation address, that is, the target of the call edge. Connected to the return jump is a restoration of the values that the local variables of the caller had before the call.

Example 2.9.1 Let us consider again the program of Example 2.8.1. The introduction of the jump edges corresponding to procedure entry and return leads to the graph of Fig. 2.6. The graph constructed in this way is called the *interprocedural supergraph*. □

Let $\mathbb{D}$ be the complete lattice for the analysis with the abstract edge effects $[\![lab]\!]^\sharp$ and the functions $\mathsf{enter}^\sharp : \mathbb{D} \to \mathbb{D}$ and $\mathsf{combine}^\sharp : \mathbb{D}^2 \to \mathbb{D}$ for the treatment of procedures. We set up the following system of inequalities to determine the invariants

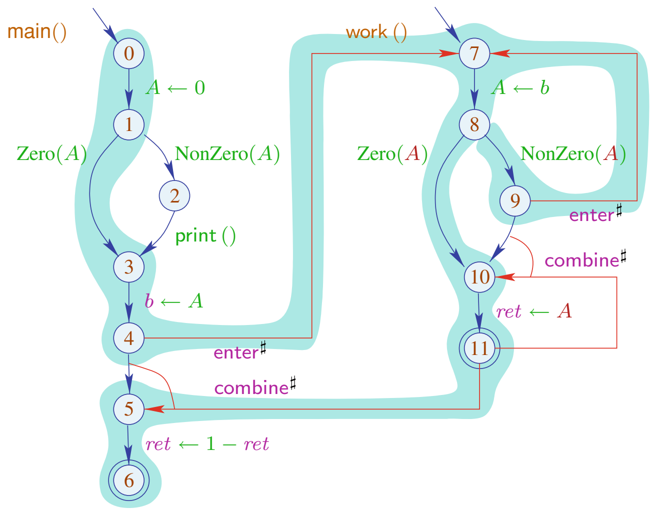


Fig. 2.7 An infeasible path in the interprocedural supergraph of Fig. 2.6

$\mathcal{D}[v]$ at each program point v:

$\mathcal{D}[start_{\mathsf{main}}] \sqsupseteq \mathsf{enter}^\sharp(d_0)$		
$\mathcal{D}[start_{\mathsf{f}}] \sqsupseteq \mathsf{enter}^\sharp(\mathcal{D}[u])$	$(u, \mathsf{f}(), v)$ call edge
$\mathcal{D}[v] \sqsupseteq \mathsf{combine}^\sharp(\mathcal{D}[u], \mathcal{D}[\mathsf{f}])$	$(u, \mathsf{f}(), v)$ call edge
$\mathcal{D}[v] \sqsupseteq [\![lab]\!]^\sharp(\mathcal{D}[u])$	$k = (u, lab, v)$ normal edge
$\mathcal{D}[\mathsf{f}] \sqsupseteq \mathcal{D}[stop_{\mathsf{f}}]$	

Example 2.9.2 Let us again regard the program of Example 2.8.1. The inequalities at the program points 5, 7, and 10 are:

$$\begin{aligned}
\mathcal{D}[5] &\sqsupseteq \mathsf{combine}^\sharp(\mathcal{D}[4], \mathcal{D}[\mathsf{work}])\\
\mathcal{D}[7] &\sqsupseteq \mathsf{enter}^\sharp(\mathcal{D}[4])\\
\mathcal{D}[7] &\sqsupseteq \mathsf{enter}^\sharp(\mathcal{D}[9])\\
\mathcal{D}[10] &\sqsupseteq \mathsf{combine}^\sharp(\mathcal{D}[9], \mathcal{D}[\mathsf{work}])
\end{aligned}$$

□

Correctness of this analysis is proved with respect to the operational semantics. Constant propagation finds, in this example, the same results as full constant propagation. The interprocedural supergraph, however, contains additional paths that the program can, in fact, never take, so-called *infeasible paths*. These infeasible paths can impair the precision of the results. Such an infeasible path is shown in Fig. 2.7.

Only one abstract value is computed for each program point. Termination of the analysis is thus guaranteed if the used complete lattice has only ascending chains that eventually stabilize.

For a comparison, let us have a look at the constraint system for call strings of length 1. This system has the unknowns $\mathcal{D}[v, \gamma]$ where v is a program point and γ is a call string of length at most 1. Then we define:

$$\begin{array}{lll}
\mathcal{D}[start_{\mathsf{main}}, \epsilon] & \sqsupseteq \mathsf{enter}^\sharp(d_0) & \\
\mathcal{D}[start_{\mathsf{f}}, \mathsf{g}] & \sqsupseteq \mathsf{enter}^\sharp(\mathcal{D}[u, \gamma]) & (u, \mathsf{f}(), v) \text{ call edge in } \mathsf{g} \\
\mathcal{D}[v, \gamma] & \sqsupseteq \mathsf{combine}^\sharp\,(\mathcal{D}[u, \gamma], \mathcal{D}[\mathsf{f}, \mathsf{g}]) & (u, \mathsf{f}(), v) \text{ call edge in } \mathsf{g} \\
\mathcal{D}[v, \gamma] & \sqsupseteq [\![lab]\!]^\sharp\,(\mathcal{D}[u, \gamma]) & k = (u, lab, v) \text{ normal edge} \\
\mathcal{D}[\mathsf{f}, \gamma] & \sqsupseteq \mathcal{D}[stop_{\mathsf{f}}, \gamma] &
\end{array}$$

Example 2.9.3 Let us regard again the program of Example 2.8.1. The inequalities at the program points 5, 7, 10 are:

$$\begin{array}{rl}
\mathcal{D}[5, \epsilon] & \sqsupseteq \mathsf{combine}^\sharp\,(\mathcal{D}[4, \epsilon], \mathcal{D}[\mathsf{work}, \mathsf{main}]) \\
\mathcal{D}[7, \mathsf{main}] & \sqsupseteq \mathsf{enter}^\sharp\,(\mathcal{D}[4, \epsilon]) \\
\mathcal{D}[7, \mathsf{work}] & \sqsupseteq \mathsf{enter}^\sharp\,(\mathcal{D}[9, \mathsf{work}]) \\
\mathcal{D}[10, \mathsf{main}] & \sqsupseteq \mathsf{combine}^\sharp\,(\mathcal{D}[9, \mathsf{main}], \mathcal{D}[\mathsf{work}, \mathsf{work}]) \\
\mathcal{D}[10, \mathsf{work}] & \sqsupseteq \mathsf{combine}^\sharp\,(\mathcal{D}[9, \mathsf{work}], \mathcal{D}[\mathsf{work}, \mathsf{work}])
\end{array}$$

□

As for call strings of length 0, the number of unknowns of the constraint system is independent of the complete lattice $\mathbb{D}$ of the analysis. Termination of the analysis is, thus, guaranteed if the used complete lattice has only ascending chains that eventually stabilize.Since the number of contexts which are distinguished is larger than for the constraint system for call string 0, the analysis result is potentially more precise. For constant propagation, e.g., it may find more interprocedural constants than an analysis with call string 0.

2.10 Exercises

1. **Parameter passing**

Describe a general method using global variables for passing call-by-value parameters.

Show that global variables can also be used for returning results of a procedure call.

2. **Reference parameter**

Extend the example programming language by the possibility to store the address &A of a local variable A in another variable or in memory.

(a). Design a simple analysis to determine a superset of the local variables whose address is taken and stored in another variable or in memory.

(b). Improve the precision of your analysis by additionally determining sets of local variables A whose address $\&A$ is taken, but only assigned to local variables

(c). Explain how your analyses can be used in last-call optimization.

3. **Removal of recursion, inlining**
   Regard the program:

```
f_1() {                          f() {              main() {
    if (n ≤ 1) z ← y;                x ← 1;             n ← M[17];
    else {                           y ← 1;             f();
        n ← n − 1;                   f_1();             M[42] ← z;
        z ← x + y;               }                  }
        x ← y;
        y ← z;
        f_1();
    }
}
```

   Remove the recursion of function f_1. Perform inlining.

4. **Interprocedural available assignments**
   Design an interprocedural elimination of redundant assignments.

   - Design an interprocedural analysis of available assignments. For that, take into account that there are global as well as local variables. How should the abstract functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ be defined?
     Is your operator $H^\sharp$ distributive? Is it possible to describe all abstract effects of edges in the control-flow graph by means of functions of the form $f\,x = (x \cup a) \cap b$ for suitable sets a, b?
   - Use the information computed by your availability analysis to remove certain redundancies.

5. **Interprocedural partial redundancies**
   Design an interprocedural elimination of partially redundant assignments.

   (a). Design an interprocedural analysis of very busy assignments.
   How should the abstract functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ be defined? For that, take into account that the analysis should be *backward*! Is your operator $H^\sharp$ distributive? Is it possible to describe all abstract effects of edges in the control-flow graph by means of functions of the form $f\,x = (x \cup a) \cap b$ for suitable sets a, b?

   (b). Use the information computed by your very business analysis to remove partial redundancies.

2.11 Literature

A first approach to the static analysis of programs with procedures is contained in the article by Cousot and Cousot (1977). Independent of this work, Sharir and Pnueli present the functional and the call-string approaches in 1981. This article contains an interprocedural coincidence theorem for procedures without local variables. A generalization for procedures with local variables is contained in Knoop and Steffen (1992). Fecht and Seidl present in 1999 a discussion of several local fixed-point algorithms as they are applicable to Prolog programs.

An application of interprocedural analysis methods for the precise analysis of loops is described by Martin et al. (1998).

An important optimization of object-oriented programs attempts to identify data objects of fixed size that do not escape from a given method call. These do not need to be allocated on the heap, but can be allocated directly on the stack (Choi et al. 1999).

Inlining is very important for object-oriented programs because they often have many small functions. These consist of only a few statements such that the effort needed for a call is greater than for the method itself. However, aggressive inlining finds its limits at dynamic method calls. The static type of an object may deviate from the run-time type of the object. This means that the method actually applied may be statically unknown. An interprocedural analysis is used to determine the exact dynamic type of the object at the call site. For a fast static analysis, as used in *just-in-time* compilers, simple, context-insensitive analysis such as the *Rapid Type Analysis* (Bacon 1997) are used. It only considers the call sites of the program and ignores all variables and assignments. The precision can be improved at the cost of its efficiency by considering assignments to variables and their types (Sundaresan et al. 2000).

Our list of program optimizations is by no means complete. We have not discussed the reduction in operator strength (Paige and Schwartz 1977; Paige 1990; Sheldon et al. 2003) or methods dedicated to programs working with arrays, and concurrent programs.

Chapter 3
Optimization of Functional Programs

In a somewhat naive view, functional programs are imperative programs without assignments.

Example 3.0.1 Consider the following program fragment written in the functional language OCAML.

$$\begin{array}{lcl} \textbf{let rec}\ \ \mathsf{fac2}\ x\ y & = & \textbf{if}\ \ y \leq 1\ \ \textbf{then}\ \ x \\ & & \textbf{else}\ \ \mathsf{fac2}\ (x \cdot y)\ (x - 1) \\ \textbf{in let}\ \ \mathsf{fac}\ x & = & \mathsf{fac2}\ 1\ x \end{array}$$

Some concepts known from imperative languages are missing. There is no sequential control flow and no loop. On the other hand, almost all functions are recursive. □

Besides recursive functions we have some more concepts in functional languages, such as OCAML, SCALA, and HASKELL, which are rarely provided by imperative languages, like pattern matching on structural values, partial application of higher-order functions, or lazy evaluation of function arguments. The type systems often provide polymorphic types, and the implementation attempts to determine types by type inference.

To increase portability of the compiler, some implementations of functional programming languages first compile to an imperative language. The *Glasgow Haskell compiler* ghc, for example, offers the compilation to C as an option. Any compiler for C can then be used to produce executable code. Other compilers for functional languages compile directly to some suitable *virtual machine*. The compiler for SCALA compiles to the JAVA *Virtual Machine*, while the compiler for F# generates .NET instructions. One possibility to optimize functional programs, which are compiled to an imperative intermediate language, is to exploit the optimizations offered by the compiler for the imperative intermediate language.This strategy is not so bad, considering that compilers for functional languages typically generate nontrivial control flow by translating sequences of *let* definitions into sequences of assignments, and tail calls into unconditional jumps. Both calls of fac2 in our example program, for example, are tail calls. Ignoring the allocation of all values on the heap, including the INT values, the imperative program generated for the function fac could look like this:

```
int fac(int x) {
            int a, a_1, b, b_1
            a ← 1;  b ← x;
    fac2 :  if (b ≤ 1)  return a;
            else {
              a_1 ← a · b;  b_1 ← b − 1;
              a ← a_1;  b ← b_1;
              goto fac2;
              }
}
```

The intraprocedural optimizations for imperative programs described so far can, therefore, also be used to improve functional programs. Assignments to dead variables can be removed. Constants or copies can be propagated. In the example, the temporary variables a_1, b_1 can be eliminated; these variables were introduced for the evaluation of the arguments of the recursive applications of fac2.

In general, the control flow resulting from the translation of functional programs into imperative programs is quite confusing, both for human readers and static analysis. Better results in the analysis and the optimization of functional programs can be obtained if the specific properties and sources of inefficiencies of functional programs are taken into account.

3.1 A Simple Functional Programming Language

As in the book *Compiler Design—Virtual Machines* (Wilhelm and Seidl), we restrict ourselves to a small fragment of the functional programming language OCAML. We consider expressions e and patterns p according to the following grammar:

$$\begin{array}{rl} e ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \ldots e_k \mid \mathbf{fun}\, x \to e \\ & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1 \Box_2\, e_2) \mid \\ & \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0 \mid \\ & \mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_k = e_k\ \mathbf{in}\ e \\ & \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\ & \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\ p ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \end{array}$$

where b denotes a value of a base type, x a variable, c a data constructor, and $\Box_i$ an i-place operator, which returns values of base type. Note that all functions are unary. However, OCAML provides *tuples* $(e_1, \ldots, e_k)$ of arbitrary length $k \geq 0$, which can be used to implement multiargument functions. Formal parameters $x_1, \ldots, x_k$ are not listed on the left sides of function definitions. Instead, functional abstraction is always used: **fun** $x_1 \to$ **fun** $x_2 \to \ldots$**fun** $x_k \to \ldots$. We also omit function definitions by cases since these can be equivalently formulated by *match* expressions. Furthermore, we assume that all programs are well-typed.

A function max computing the maximum of two numbers looks as follows:

$$\begin{array}{lll} \textbf{let}\ \mathsf{max} & = & \textbf{fun}\ x \to\ \textbf{fun}\ y \to\ \textbf{if}\ x_1 < x_2\ \textbf{then}\ x_2 \\ & & \qquad\qquad\qquad\quad \textbf{else}\ x_1 \end{array}$$

3.2 Some Simple Optimizations

This section presents some simple optimizations for functional programs. The basic idea behind all of them is to move evaluations from run-time to compile-time.

A *function application* (**fun** $x \to e_0$) e_1 can be rewritten into the *let* expression, **let** $x = e_1$ **in** e_0. A *case distinction* can be optimized if the expression to which patterns are compared is already partly known at compile-time. Consider the expression

$$\textbf{match}\ c\, e_1 \ldots e_k\ \textbf{with}\ \ldots\ c\, x_1 \ldots x_k \to e\ \ldots$$

where all patterns to the left of $c\, x_1 \ldots x_k$ start with a constructor, which is different from c. The compiler knows then that only the alternative for $c\, x_1 \ldots x_k$ may match. The expression can, therefore, be transformed into

$$\textbf{let}\ x_1 = e_1\ \ldots\ \textbf{in let}\ x_k = e_k\ \textbf{in}\ e$$

Both transformations are semantics-preserving and replace more complex program constructs by *let* expressions.

A *let* expression **let** $x = e_1$ **in** e_0 can be rewritten into $e_0[e_1/x]$, that is, into the main expression e_0, in which each free occurrence of x is replaced by the expression e_1.This transformation corresponds to the β reduction in the λ calculus. It must, however, be taken care that none of the variables occurring free in e_1 is bound by the substitution into e_0.

Example 3.2.1 Consider the expression:

$$\begin{array}{l} \quad\textbf{let}\ x = 17 \\ \textbf{in let}\ \mathsf{f} = \textbf{fun}\ y \to x + y \\ \textbf{in let}\ x = 4 \\ \textbf{in}\ \ \mathsf{f}\ x \end{array}$$

The variable x that is visible in the definition of f represents the value 17, while the variable x that is visible in the application f x represents the value 4. The expression, therefore, evaluates to 21.

The application of the *let* optimization to the second **let** returns the expression:

$$\begin{array}{l} \quad\textbf{let}\ x\ = 17 \\ \textbf{in let}\ x = 4 \\ \textbf{in}\ \ (\textbf{fun}\ y \rightarrow x + y)\ x \end{array}$$

The variable x, now visible in the function as well as in its argument, represents the value 4. The expression evaluates to the value 8. □

There exist several possibilities to solve this problem. The simplest possibility, which we will also use, consists in renaming the variables that are bound in e_0 in such a way that their new names are different from those of variables occurring free in e_1. Renaming of this kind is called α *conversion*.

Example 3.2.2 Consider again the expression of Example 3.2.1. The free variable x of the function **fun** $y \rightarrow x + y$ occurs as bound variable in the expression into which the function is to be substituted. Renaming this occurrence of x results in the expression:

$$\begin{array}{l} \quad\textbf{let}\ x\ \ = 17 \\ \textbf{in let}\ \mathsf{f}\ \ = \textbf{fun}\ y \rightarrow x + y \\ \textbf{in let}\ x' = 4 \\ \textbf{in}\ \ \mathsf{f}\ x' \end{array}$$

The substitution of **fun** $y \rightarrow x + y$ for f produces:

$$\begin{array}{l} \quad\textbf{let}\ x\ \ = 17 \\ \textbf{in let}\ x' = 4 \\ \textbf{in}\ \ (\textbf{fun}\ y \rightarrow x + y)\ x' \end{array}$$

Evaluation of this expression produces the correct result, 21. □

No renaming of variables is necessary if the free variables of the expression e_1 do not occur bound in e_0. This is in particular the case if e_1 does not have any free variables.

The transformation of *let* expressions is only an improvement if its application does not lead to additional evaluations. This is definitely the case in the following three special situations:

- Variable x does not occur in e_0. In this case the evaluation of e_1 is completely avoided by applying the transformation.
- Variable x occurs exactly once in e_0. In this case the evaluation of e_1 is just moved to another position.
- Expression e_1 is just a variable z. In this case all accesses to variable x in e_0 are replaced by accesses to variable z.

But attention! The application of the *let* transformation, including α conversion, preserves the semantics only if the functional language prescribes lazy evaluation for *let* expressions, as is the case in HASKELL. With lazy evaluation the argument e_1

in the application $(\textbf{fun}\ x \rightarrow e_0)\ e_1$ will only be evaluated if and when the value of x is accessed during the evaluation of e_0.

Eager evaluation of *let* expressions, as in OCAML, will evaluate the expression e_1 in any case. The evaluation of the whole *let* expression does not terminate if the evaluation of e_1 does not terminate. If x does not occur in the main expression e_0 or only in a subexpression that is not evaluated, then evaluation of the transformed expression may still terminate although the evaluation of the original expression would not.

Example 3.2.3 Consider the program:

$$\begin{array}{l} \textbf{let rec}\ \mathsf{f} = \textbf{fun}\ x \rightarrow 1 + \mathsf{f}\ x \\ \ \textbf{in let}\ y = \mathsf{f}\ 0 \\ \ \textbf{in}\quad 42 \end{array}$$

With lazy evaluation, the program returns the value 42. With eager evaluation, the program does not terminate since the evaluation of f 0 is started before the value 42 is returned. The variable y does not occur in the main expression. The application of the *let* optimization would remove the evaluation of f 0. Consequently, the evaluation of the optimized program would terminate and return 42. □

One is perhaps not worried about an *improved* termination behavior. The situation is different if the evaluation of e_1 has side effects which are required. This cannot happen in our small example language. OCAML expressions may well raise exceptions or interact with their environment, independently of whether their return value is accessed or not. In this case the application of the transformation must be restricted to expressions e_1 that neither directly nor indirectly cause side effects. This is definitely the case with variables and expressions that directly represent values such as functions.

Further optimizations become possible when *let* definitions are moved in front of the evaluation of an expression.

$$\begin{array}{lll} ((\textbf{let}\ x = e\ \textbf{in}\ e_0)\ e_1) & = & (\textbf{let}\ x = e\ \textbf{in}\ e_0\ e_1), \\ & & \text{if } x \text{ is not free in } e_1 \\ (\textbf{let}\ y = e_1\ \textbf{in let}\ x = e\ \textbf{in}\ e_0) & = & (\textbf{let}\ x = e\ \textbf{in let}\ y = e_1\ \textbf{in}\ e_0), \\ & & \text{if } x \text{ is not free in } e_1 \text{ and } y \text{ is not free in } e \\ (\textbf{let}\ y = \textbf{let}\ x = e\ \textbf{in}\ e_1\ \textbf{in}\ e_0) & = & (\textbf{let}\ x = e\ \textbf{in let}\ y = e_1\ \textbf{in}\ e_0), \\ & & \text{if } x \text{ is not free in } e_0 \end{array}$$

The applicability of these rules is not restricted if no side effects are involved. Even the termination behavior does not change. The application of these rules may move a *let* definition further to the outside creating chances, e.g., for the application of the transformation *Inlining* presented in the next section. Further movements of *let* definitions are discussed in Exercises 1, 2, and 3.

3.3 Inlining

As for imperative programs, inlining for functional programs is performed to save the costs associated with function application. Inlining of a function f means that the body of f is copied at the place of application. This is quite analogous to inlining in imperative languages as treated in Sect. 2.3. A notable difference between the two transformations is that our imperative core language simulated parameter and result passing by copying values between global and local variables and possibly also memory. We, therefore, assumed procedures to have no parameters. Under this assumption, procedure inlining just means to replace the call to a procedure by a copy of its body.

With functional languages, me must make passing of parameters explicit. Let us assume that a function f is defined as **let** f $=$ **fun** $x \to e_0$. Inlining replaces the application f e_1 by:

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_0$$

Example 3.3.1 Consider the program fragment:

```
   let  fmax = fun f → fun x → fun y →
                 if  x > y  then  f x
                 else  f y
in let max   = fmax (fun z → z)
```

Applying inlining to the definition of max results in:

```
let max = let f = fun z → z
          in fun x → fun y → if  x > y  then  f x
                             else  f y
```

Inlining of f then delivers:

```
let max = let f = fun z → z
          in fun x → fun y → if  x > y  then  let z = x
                                              in z
                             else  let z = y
                                   in z
```

Applying the *let* optimizations for variables and useless constant definitions yields:

```
let max = fun x → fun y → if  x > y  then  x
                          else  y
```
□

The inlining transformation can be understood as a combination of a restricted case of the *let* optimization of the preceding section with the optimization of function application. The *let* optimization is partly applied to *let* expressions of the form

let $\mathsf{f} = $ **fun** $x \to e_0$ **in** e where the functional value **fun** $x \to e_0$ is only copied to such occurrences in e at which f is applied to an argument. Subsequently, optimization of function applications is performed at these places. Inlining requires, as did the previous optimizations, some care to guarantee correctness and termination of the transformation. α conversion of the bound variables in the expression e before the application of inlining must guarantee that no free variable in e_0 will be bound by the transformation.

As in the case of imperative languages, inlining is only applied to nonrecursive functions. In our core language, these are the *let*-defined functions. In itself, this does not suffice to guarantee termination of the transformation in all functional languages.

Example 3.3.2 Consider the program fragment:

$$\begin{array}{l} \textbf{let}\ \mathsf{w}\ = \textbf{fun}\ \mathsf{f} \to \textbf{fun}\ \mathsf{y} \to \mathsf{f(y\ f\ y)} \\ \textbf{in let}\ \mathsf{fix} = \textbf{fun}\ \mathsf{f} \to \mathsf{w\ f\ w} \end{array}$$

Neither w nor fix are recursive. We may apply inlining to the body w f w of the function fix. With the definition of w this yields for fix the function:

$$\textbf{fun}\ \mathsf{f} \to \textbf{let}\ \mathsf{f} = \mathsf{f}\ \textbf{in let}\ \mathsf{y} = \mathsf{w}\ \textbf{in}\ \mathsf{f(y\ f\ y)}\ ,$$

which can be simplified to:

$$\textbf{fun}\ \mathsf{f} \to \mathsf{f(w\ f\ w)}\ .$$

Inlining can be repeated. After k repetitions this results in:

$$\textbf{fun}\ \mathsf{f} \to \mathsf{f}^k\mathsf{(w\ f\ w)}\ ,$$

and inlining can again be applied. □

Nontermination as in our example can happen in untyped languages like LISP.The function w is, however, rejected as not typeable in *typed* languages like OCAML and HASKELL. In these languages, inlining of *let*-defined functions always terminates and yields uniquely determined normal forms up to the renaming of bound variables. Still, in untyped languages, the problem of potential nontermination can be pragmatically solved: Whenever inlining lasts too long it may be stopped.

3.4 Specialization of Recursive Functions

Inlining is a technique to improve the efficiency of applications of nonrecursive functions. What can be done to improve the efficiency of recursive functions? Let us look at one particular programming technique often used in functional programming languages. It uses recursive polymorphic functions of higher order as, e.g., the function map. Such functions distill the algorithmic essence of a programming technique and are instantiated for a particular application by supplying the required parameters, including functional parameters.

Example 3.4.1 Consider the following program fragment:

```
let f = fun x → x · x
in let rec map = fun g → fun y → match y
                                 with    [ ] → [ ]
                                      |  x₁ :: z → g x₁ :: map g z
in map f list
```

The actual parameter of the function application map f is the function **fun** $x \rightarrow x \cdot x$. The function application map f *list* thus represents a function that squares all elements of the list *list*. Note that we have as usual written the list constructor :: as infix operator between its two arguments. □

Let f be a recursive function and f v an application of f to an expression v that represents a value, that is, either another function or a constant. Our goal is to introduce a new function h for the expression f v. This optimization is called *function specialization*.

Let f be defined by **let rec** f= **fun**$x \rightarrow e$. We define h by:

$$\textbf{let}\ \mathsf{h} = \textbf{let}\ x = v\ \textbf{in}\ e$$

Example 3.4.2 Let us regard the program fragment of Example 3.4.1.

$$
\begin{array}{l}
\textbf{let}\ \mathsf{h} = \textbf{let}\ \mathsf{g} = \boxed{\textbf{fun}\ x \rightarrow x \cdot x} \\
\qquad\qquad \textbf{in}\ \textbf{fun}\ y \rightarrow \textbf{match}\ y \\
\qquad\qquad\qquad \textbf{with}\quad [\,] \rightarrow [\,] \\
\qquad\qquad\qquad\quad \mid\ x_1 :: z \rightarrow \mathsf{g}\ x_1 :: \boxed{\mathsf{map\ g}}\ z
\end{array}
$$

The function map is recursive. Therefore, the body of the function h contains another application of map. Specialization of map for this application would introduce a function h_1 with the same definition (up to renaming of bound variables) as h. Instead of introducing this new function h_1, the application map g is replaced by the function h. This replacement of the right side of a definition by its left side is called function *folding*. Function folding in the example yields:

```
let rec h = let g = fun x → x · x
            in fun y → match y
                       with    [ ] → [ ]
                            |  x₁ :: z → g x₁ :: h z
```

The definition of h no longer contains any explicit application of the function map and is itself recursive. Inlining of the function g yields:

$$
\begin{array}{l}
\textbf{let rec}\ \mathsf{h} = \textbf{let}\ \mathsf{g} = \textbf{fun}\ x \ \rightarrow\ x \cdot x \\
\qquad\qquad\quad \textbf{in}\ \ \textbf{fun}\ y \ \rightarrow\ \textbf{match}\ y \\
\qquad\qquad\qquad\qquad \textbf{with} \quad [\,] \rightarrow [\,] \\
\qquad\qquad\qquad\qquad\ \ \mid \quad x_1 :: z \ \rightarrow\ (\,\textbf{let}\ x = x_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ x \cdot x\,)\ :: \mathsf{h}\ z
\end{array}
$$

The removal of superfluous definitions and variable-to-variable bindings finally yields:

$$
\begin{array}{l}
\textbf{let rec}\ \mathsf{h} = \ \textbf{fun}\ y \ \rightarrow\ \textbf{match}\ y \\
\qquad\qquad\qquad\quad \textbf{with} \quad [\,] \rightarrow [\,] \\
\qquad\qquad\qquad\quad\ \ \mid \quad x_1 :: z \ \rightarrow x_1 \cdot x_1 :: \mathsf{h}\ z
\end{array}
$$

□

We can, in general, not expect that the recursive calls of the function to be specialized, can be immediately folded to another application of specialization. Worse, it may happen that continuing specialization leads to an infinite number of auxiliary functions and to nontermination. A pragmatic point of view would again stop the endless creation of new functions when the number of auxiliary functions exceeds a given threshold.

3.5 An Improved Value Analysis

Inlining and function specialization optimize function applications $f\ e$ where we assume that the function f is defined in an enclosing *let*- or *letrec*-expression. Functional languages, however, allow functions to be passed as arguments or be returned as results. The applicability of inlining and function specialization therefore relies on an analysis that for each variable determines a *superset* of its potential run-time values. This is the goal of the next analysis.

In a first step, the analysis identifies the subset E of the set of expressions occurring in a given program whose values should be determined. This set E consists of all subexpressions of the program that are variables, function applications, *let*-, *letrec*-, *match*-, or *if*-expressions.

Example 3.5.1 Consider the following program:

$$
\begin{array}{l}
\textbf{let rec}\ \mathsf{from} = \textbf{fun}\ i \ \rightarrow i :: \mathsf{from}\ (i+1) \\
\quad \textbf{and}\ \mathsf{first} = \textbf{fun}\ l \rightarrow \textbf{match}\ l\ \textbf{with}\ x :: xs \rightarrow x \\
\textbf{in}\ \ \mathsf{first}\ (\mathsf{from}\ 2)
\end{array}
$$

The set E consists of the expressions:

$$E = \{\mathsf{from}, i, \mathsf{from}\,(i+1), \mathsf{first}, l, x, \mathsf{from}\ 2, \mathsf{first}\ (\mathsf{from}\ 2), \\ \mathbf{match}\, l\, \mathbf{with} \ldots, \mathbf{let\ rec}\, \mathsf{from} = \ldots\}$$

$\square$

Let V be the set of the remaining subexpressions of the program. The expressions in V, thus, are either values such as function abstractions or constants, or they provide at least the outermost constructor or outermost operator application. In the program of Example 3.5.1, the set V therefore is:

$$V = \{\mathbf{fun}\ i\ \to \ldots, i :: \mathsf{from}\,(i+1), i+1, \mathbf{fun}\ l \to \ldots, 2\}$$

Each subexpression e in V can be decomposed in a unique way into an upper part, in which only constructors, values, or operators occur, and the maximal subexpressions $e_1, \ldots, e_k$ from the set E below those.

The upper part is represented by a k-place *pattern*, that is, a term in which the pattern variables $\bullet_1, \ldots, \bullet_k$ at the leaves stand for the expressions $e_1, \ldots, e_k$. The expression e has the form $e \equiv t[e_1/\bullet_1, \ldots, e_k/\bullet_k]$, or in shorter form, $e \equiv t[e_1, \ldots, e_k]$.

In our example, the expression $e \equiv (i :: \mathsf{from}\,(i+1))$ can be decomposed into $e \equiv t[i, \mathsf{from}\,(i+1)]$ for expressions i, $\mathsf{from}\,(i+1)$ from E and the pattern $t \equiv (\bullet_1 :: \bullet_2)$.

Our goal consists in identifying for each expression $e \in E$ a subset of expressions from V into which e may develop. We first explain in which sense a relation $G \subseteq E \times V$ for each expression from E defines a set of *value expressions* and then present a method to compute such subsets from V. A *value expression* is an expression v that is formed according to the following grammar:

$$v \quad ::= \quad b \mid \mathbf{fun}\ x \to e \mid c\, v_1 \ldots v_k \mid (v_1, \ldots, v_k) \mid \square_1\ v \mid v_1 \square_2 v_2$$

for basic values b, arbitrary expressions e, constructors c and unary and binary operators $\square_1, \square_2$, respectively, that return basic values.

Let $G \subseteq E \times V$ be a relation between expressions from E and V. Expressions $e \in E$ are associated with the set $[\![e]\!]^\sharp_G$ of all value expressions that can be derived from e using G. Each pair $(e, t[e_1, \ldots, e_k]) \in G$ for expressions $e, e_1, \ldots, e_k \in E$ and patterns t can be seen as the inequality

$$[\![e]\!]^\sharp_G \supseteq t[[\![e_1]\!]^\sharp_G, \ldots, [\![e_k]\!]^\sharp_G]$$

Here, we interpret the application of the pattern t to sets $V_1, \ldots, V_k$ as the set

$$t[V_1, \ldots, V_k] = \{t[v_1, \ldots, v_k] \mid v_i \in V_i\}$$

The sets $[\![e]\!]^\sharp_G$, $e \in E$, are defined as the least solution of this system of inequalities.

Example 3.5.2 Let G be the relation

$$\{(i, 2), (i, i+1)\} .$$

The set $[\![i]\!]^\sharp_G$ consists of all expressions of the form 2 or $(\ldots(2+1)\ldots)+1$. Note that in this analysis, operator applications are treated in the same way as data constructors.

The set $[\![\mathsf{from}\ (i+1)]\!]^\sharp_{G'}$ on the other hand is empty for

$$G' = \{(\mathsf{from}\ (i+1), i :: \mathsf{from}\ (i+1))\}$$

$\square$

A relation G can be seen as a *regular tree grammar* with the set of nonterminals E and constants and function abstractions as 0-ary terminal symbols, and operators and constructors as multiplace terminal symbols. For an expression $e \in E$, i.e., a nonterminal, the set $[\![e]\!]^\sharp_G$ denotes the set of terminal expressions derivable from e according to the grammar (see Exercises 4 and 5). The sets $[\![e]\!]^\sharp_G$ are, in general, *infinite*. The relation G, however, is a *finite* representation of these sets, which makes it possible to decide simple properties of the sets $[\![e]\!]^\sharp_G$. The most important question is whether $[\![e]\!]^\sharp_G$ contains a certain term v. This question can be easily answered if v is a function abstraction $\mathbf{fun}\, x \to e'$. Each pair (e, u) from G has a right side u that is either a constant, a function, or the application of a constructor or an operator. Therefore, $(\mathbf{fun}\, x \to e') \in [\![e]\!]^\sharp_G$ holds if and only if $(e, \mathbf{fun}\, x \to e') \in G$.

Further examples of properties that can be easily decided are:

- Is $[\![e]\!]^\sharp_G$ nonempty?
- Is $[\![e]\!]^\sharp_G$ finite, and if yes, of which elements does this set consist?

The goal of our value analysis is to construct a relation G for a program such that for each expression e of the program, the set $[\![e]\!]^\sharp_G$ contains all values to which e may develop during runtime, relative to the bindings for the free variables in e. The relation $G \subseteq E \times V$ is defined by means of axioms and derivation rules. For convenience, we will not define the relation G itself but the relation $\Rightarrow$ which is the relation G, extended with all pairs (v, v) for $v \in V$. Also for convenience, we write the relation $\Rightarrow$ in infix notation. Axioms represent those relationships that hold without preconditions:

$$v \Rightarrow v \qquad (v \in V)$$

i.e, expressions from the set V are related to themselves. We supply rules for each program construct:

Function application. Let $e \equiv (e_1\ e_2)$. We have the rules:

$$\frac{e_1 \Rightarrow \mathbf{fun}\, x \to e_0 \quad e_0 \Rightarrow v}{e \Rightarrow v} \qquad \frac{e_1 \Rightarrow \mathbf{fun}\, x \to e_0 \quad e_2 \Rightarrow v}{x \Rightarrow v}$$

If the function expression of a function application evaluates to a function $\mathbf{fun}\, x \to e_0$ and the main expression e_0 of the function evaluates to a value v, then the function application may develop into v. If, on the other hand, the argument of the function application evaluates to a value v , then v is a potential value of the formal parameter x of the function.

let-**Definition**. Let $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0$. We then have the rules:

$$\frac{e_0 \Rightarrow v}{e \Rightarrow v} \qquad \frac{e_1 \Rightarrow v}{x \Rightarrow v}$$

If the main expression e_0 of a *let*-expression evaluates to a value v, then so does the whole *let*-expression. Each value for the expression e_1 represents a potential value for the local variable x. A similar consideration justifies the rules for *letrec*-expressions.

letrec **Definition**. For $e \equiv \mathbf{let\ rec}\ x_1 = e_1 \ldots \mathbf{and}\ x_k = e_k\ \mathbf{in}\ e_0$, we have:

$$\frac{e_0 \Rightarrow v}{e \Rightarrow v} \qquad \frac{e_i \Rightarrow v}{x_i \Rightarrow v}$$

Case distinctions. Let $e \equiv \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \to e_1 \mid \ldots \mid p_m \to e_m$.

If p_i is a basic value we have the rules:

$$\frac{e_i \Rightarrow v}{e \Rightarrow v}$$

If on the other hand, $p_i \equiv c\, y_1 \ldots y_k$, we have:

$$\frac{e_0 \Rightarrow c\, e'_1 \ldots e'_k \quad e_i \Rightarrow v}{e \Rightarrow v} \quad \frac{e_0 \Rightarrow c\, e'_1 \ldots e'_k \quad e'_j \Rightarrow v}{y_j \Rightarrow v} \quad (j = 1, \ldots, k)$$

If finally p_i is a variable y, we have:

$$\frac{e_i \Rightarrow v}{e \Rightarrow v} \qquad \frac{e_0 \Rightarrow v}{y \Rightarrow v}$$

If an alternative evaluates to a value, then the whole case distinction may evaluate to that value, as long as the corresponding pattern cannot be statically excluded for the evaluation of the expressions e_0. The analysis does not track the exact values of operator applications. Therefore, basic values are always assumed to be possible. This is different for patterns $c\, y_1 \ldots y_k$. Such a pattern matches the value of e_0 only if $e_0 \Rightarrow v$ holds, where c is the outermost constructor of v. In this case v has the form $v = c\ e'_1 \dots e'_k$, and the values for e'_i are potential values for the variables x_i.

Conditional expressions. Let $e \equiv$ **if** e_0 **then** e_1 **else** e_2. For $i = 1, 2$ we have:

$$\frac{e_i \Rightarrow v}{e \Rightarrow v}$$

These rules are similar to the rules for *match*-expressions where basic values are used as patterns.

Example 3.5.3 Consider again the program

$$\begin{array}{l} \textbf{let rec}\ \mathsf{from} = \textbf{fun}\ i \to i :: \mathsf{from}\ (i+1) \\ \quad \textbf{and}\ \mathsf{first} = \textbf{fun}\ l \to \textbf{match}\ l\ \textbf{with}\ x :: xs \to x \\ \textbf{in}\ \ \mathsf{first}\ (\mathsf{from}\ 2) \end{array}$$

This program terminates only with lazy evaluation like in HASKELL. In OCAML, however, with eager evaluation, the application from 2 and with it the whole program does not terminate. One possible derivation of the relation $x \Rightarrow 2$ would look as follows:

$$\dfrac{\dfrac{\mathsf{first} \Rightarrow \textbf{fun}\ l \to \dots \qquad \dfrac{\mathsf{from} \Rightarrow \textbf{fun}\ i \to i :: \mathsf{from}\ (i+1)}{\mathsf{from}\ 2 \Rightarrow i :: \mathsf{from}\ (i+1)}}{l \Rightarrow i :: \mathsf{from}\ (i+1)} \qquad \dfrac{\mathsf{from} \Rightarrow \textbf{fun}\ i \to i :: \mathsf{from}\ (i+1)}{i \Rightarrow 2}}{x \Rightarrow 2}$$

We have left out occurrences of axioms $v \Rightarrow v$. For $e \in E$ let $G(e)$ be the set of all expressions $v \in V$, for which $e \Rightarrow v$ can be derived.

The analysis of this program delivers for the variables and function applications:

$$\begin{array}{ll} G(\mathsf{from}) & = \{\textbf{fun}\, i\ \to i :: \mathsf{from}\ (i+1)\} \\ G(\mathsf{from}\ (i+1)) & = \{i :: \mathsf{from}\ (i+1)\} \\ G(\mathsf{from}\ 2) & = \{i :: \mathsf{from}\ (i+1)\} \\ G(i) & = \{2, i+1\} \\ G(\mathsf{first}) & = \{\textbf{fun}\, l\ \to \textbf{match}\ l \dots\} \\ G(l) & = \{i :: \mathsf{from}\ (i+1)\} \\ G(x) & = \{2, i+1\} \\ G(xs) & = \{i :: \mathsf{from}\ (i+1)\} \\ G(\mathsf{first}\ (\mathsf{from}\ 2)) & = \{2, i+1\} \end{array}$$

We conclude that the evaluation of the expressions from 2 and from $(i + 1)$ never delivers a finite value. On the other hand, the variable i will potentially be bound to expressions with values $2, 2+1, 2+1+1, \dots$. According to the analysis, the main expression evaluates to one of the values $2, 2+1, 2+1+1, \dots$. □

The sets $G(e)$ can be computed by fixed-point iteration. A more clever implementation does not calculate with sets of expressions, but propagates expressions $v \in V$ individually. When v is added to a set $G(e)$, the applicability conditions for further rules might be satisfied, which may add more expressions v' to sets $G(e')$. This is the idea of the algorithm of Heintze (1994).

Correctness of this analysis can be shown using an operational semantics for programs with delayed evaluation. We do not present this proof here, but like to mention that the derivation rules for the relation $e \Rightarrow v$ are quite analogous to the corresponding rules of the operational semantics, with the following notable exceptions:

- Operators $\Box_1, \Box_2$ on base types are not further evaluated;
- At case distinctions depending on basic values, all possibilities are explored non-deterministically;
- In case distinctions, the order of the patterns is not taken into account;
- The computation of the return value of a function is decoupled from the determination of the potential actual parameters of the function.

The analysis described is also correct for programs of a programming language with eager expression evaluation. For these, the precision of the analysis can be increased by requiring additional constraints for rules applications:

- The set $[\![e_2]\!]^\sharp_G$ should not be empty at function applications with argument e_2;
- At *let*- and *letrec*-expressions, the sets $[\![e_i]\!]^\sharp_G$ for the right sides e_i of locally introduced variables should not be empty;
- At conditional expressions, the sets $[\![e_0]\!]^\sharp_G$ for the condition e_0 should not be empty;
- Analogously, at case distinctions **match** $e_0 \ldots$, the set $[\![e_0]\!]^\sharp_G$ should not be empty. In the rules for patterns $c\ y_1 \ldots y_k$, the sets $[\![e'_j]\!]^\sharp_G$ for $j = 1, \ldots, k$ for the value $c\, e'_1 \ldots e'_k$ in $[\![e_0]\!]^\sharp_G$ should not be empty.

In the example, if holds that:

$$[\![l]\!]^\sharp_G = [\![x]\!]^\sharp_G = [\![xs]\!]^\sharp_G = [\![\textbf{match}\ l \ldots]\!]^\sharp_G = [\![\mathsf{first}\ (\mathsf{from}\ 2)]\!]^\sharp_G = \emptyset$$

The analysis therefore finds out that eager evaluation of the application first (from 2) does not terminate.

The value analysis just presented, produces amazingly precise results. It can be extended to an analysis of the exceptions possibly thrown during the evaluation of an expression (see Exercise 7) and an analysis of the set of side effects possibly occurring during expression evaluation (see Exercise 8). Imprecision, however, can come in since in the analysis of functions the approximation of the potential parameter values is decoupled from the determination of the return values. In imperative languages, this corresponds to an interprocedural analysis which uses call strings of length 0. In the analysis of polymorphic functions this means that arguments of different types are not differentiated.

3.6 Elimination of Intermediate Data Structures

One of the most important data structures offered by functional programming languages is lists. Functional programs compute results from the values in lists, collect intermediate results in lists, and apply functions to all elements in lists. Program libraries contain higher-order functions on list arguments supporting this programming style. Examples of such higher-order functions are:

```
map = fun f → fun l → match l
                      with [] → []
                      | h :: t → f x :: map f t
filter = fun p → fun l → match l
                         with [] → []
                         | h :: t → if p h then h :: filter p t
                                    else filter p t)

fold_left = fun f → fun a → fun l → match l with [] → a
                    | h :: t → fold_left f (f a h) t)
```

Functions can be composed by function composition:

```
comp = fun f → fun g → fun x → f (g x)
```

The next example shows how quite complex functions can be constructed out of these few components.

Example 3.6.1 The following program fragment supplies functions to compute the sum of all elements of a list, to determine the length of a list, and to compute the standard deviation of the elements of the list.

```
   let sum    = fold_left (+) 0
in let length = comp sum (map (fun x → 1))
in let der    = fun l →    let s1   = sum l
                        in let n    = length l
                        in let mean = s1/n
                        in let s2   = sum (
                                      map (fun x → x · x) (
                                      map (fun x → x − mean) l))
                        in     s2/n
```

Here, $(+)$ denotes the function **fun** $x \to$ **fun** $y \to x + y$. The definition above does not show recursion explicitly. However, it is implicitly used to define the functions map and fold_left. The definition of length does not need explicit functional abstraction **fun** $\ldots \to$. On the one hand, the implementation is clearer. On the other hand, this programming style leads to programs that create data structures for intermediate results, which actually could be avoided. Function length can directly be implemented by:

$$\textbf{let}\ \mathsf{length} = \mathsf{fold\_left}\ (\textbf{fun}\ a \to \textbf{fun}\ y \to a + 1)\ 0$$

This implementation avoids the creation of the auxiliary list, which contains one 1 for each element of the input. □

The following rules allow for the elimination of some apparently superfluous auxiliary data structures.

$$\begin{array}{lll}
\mathsf{comp}\ (\mathsf{map}\ \mathsf{f})\ (\mathsf{map}\ \mathsf{g}) & = & \mathsf{map}\ (\mathsf{comp}\ \mathsf{f}\ \mathsf{g}) \\
\mathsf{comp}\ (\mathsf{fold\_left}\ \mathsf{f}\ a)\ (\mathsf{map}\ \mathsf{g}) & = & \mathsf{fold\_left}\ (\textbf{fun}\ a \to \mathsf{comp}\ (\mathsf{f}\ a)\ \mathsf{g})\ a \\
\mathsf{comp}\ (\mathsf{filter}\ \mathsf{p}_1)\ (\mathsf{filter}\ \mathsf{p}_2) & = & \mathsf{filter}\ (\textbf{fun}\ x \to\ \textbf{if}\ \mathsf{p}_2\ x\ \textbf{then}\ \mathsf{p}_1\ x \\
 & & \qquad\qquad\qquad\ \textbf{else}\ \mathsf{false}) \\
\mathsf{comp}\ (\mathsf{fold\_left}\ \mathsf{f}\ a)\ (\mathsf{filter}\ \mathsf{p}) & = & \mathsf{fold\_left}\ (\textbf{fun}\ a \to \textbf{fun}\ x \to\ \textbf{if}\ \mathsf{p}\ x\ \textbf{then}\ \mathsf{f}\ a\ x \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ a)\ a
\end{array}$$

The evaluation of the left sides always needs an auxiliary data structure, while the evaluation of the right sides does not. Applying such rules for eliminating intermediate data structures is called *deforestation*. Deforestation allows to optimize the function length of Example 3.6.1. However, left sides and right sides are now no longer equivalent under all circumstances. In fact, these rules may only be applied if the functions f, g, p_1, p_2 that occur have no side effects. Another problem of this optimization consists in recognizing *when* it can be applied. Programmers often do not use explicit function composition to sequentially apply functions. They instead use directly nested function applications. An example is the definition of function der in Example 3.6.1. For this case, the transformation rules should be written as:

$$\begin{array}{lll}
\mathsf{map}\ \mathsf{f}\ (\mathsf{map}\ \mathsf{g}\ l) & = & \mathsf{map}\ (\textbf{fun}\ z \to \mathsf{f}\ (\mathsf{g}\ z))\ l \\
\mathsf{fold\_left}\ \mathsf{f}\ a\ (\mathsf{map}\ \mathsf{g}\ l) & = & \mathsf{fold\_left}\ (\textbf{fun}\ a \to \textbf{fun}\ z \to \mathsf{f}\ a\ (\mathsf{g}\ z))\ a\ l \\
\mathsf{filter}\ \mathsf{p}_1\ (\mathsf{filter}\ \mathsf{p}_2\ l) & = & \mathsf{filter}\ (\textbf{fun}\ x \to\ \textbf{if}\ \mathsf{p}_2\ x\ \textbf{then}\ \mathsf{p}_1\ x \\
 & & \qquad\qquad\qquad\ \textbf{else}\ \mathsf{false})\ l \\
\mathsf{fold\_left}\ \mathsf{f}\ a\ (\mathsf{filter}\ \mathsf{p}\ l) & = & \mathsf{fold\_left}\ (\textbf{fun}\ a \to \textbf{fun}\ x \to\ \textbf{if}\ p\ x\ \textbf{then}\ \mathsf{f}\ a\ x \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ a)\ a\ l
\end{array}$$

Example 3.6.2 The application of these rules to the definition of function der of Example 3.6.1 leads to:

```
   let sum    = fold_left (+) 0
in let length = fold_left (fun a → fun z → a + 1) 0
in let der    = fun l →    let s_1  = sum l
                        in let n    = length l
                        in let mean = s_1/n
                        in let s_2  = fold_left (fun a → fun z →
                                        (+) a (
                                           (fun x → x · x) (
                                           (fun x → x − mean) z))) 0 l
                        in     s_2/n
```

Applying the optimization of function application repeatedly and performing *let* optimization leads to:

```
   let sum    = fold_left (+) 0
in let length = fold_left (fun a → fun z → a + 1) 0
in let der    = fun l →    let s_1  = sum l
                        in let n    = length l
                        in let mean = s_1/n
                        in let s_2  = fold_left (fun a → fun z →
                                             let x = z − mean
                                          in let y = x · x
                                          in     a + y) 0 l
                        in     s_2/n
```

All intermediate data structures have disappeared. Only applications of the functions fold_left remain. The function fold_left is tail recursive such that the compiler can generate code that is as efficient as loops in imperative languages. □

Sometimes, a first list of intermediate results is produced by *tabulation* of a function. Tabulation of n values of a function f : **int** → τ produces a list:

$$[\mathsf{f}\ 0; \ldots;\ \mathsf{f}\ (n-1)]$$

A function tabulate to compute this list can be defined in OCAML by:

```
let tabulate   = fun n → fun f →
    let rec tab = fun j → if j ≥ n then [ ]
                          else (f j) :: tab (j + 1)
    in  tab 0
```

Under the conditions that all occurring functions terminate and have no side effects it holds that:

$$\begin{array}{lcl}
\mathsf{map\ f\ (tabulate}\ n\ \mathsf{g)} & = & \mathsf{tabulate}\ n\ \mathsf{(comp\ f\ g)} \\
 & = & \mathsf{tabulate}\ n\ (\textbf{fun}\ j \rightarrow \mathsf{f\ (g}\ j))\\
\mathsf{fold\_left\ f}\ a\ \mathsf{(tabulate}\ n\ \mathsf{g)} & = & \mathsf{loop}\ n\ (\textbf{fun}\ a \rightarrow \mathsf{comp\ (f}\ a\mathsf{)\ g)}\ a \\
 & = & \mathsf{loop}\ n\ (\textbf{fun}\ a \rightarrow \textbf{fun}\ j \rightarrow (\mathsf{f}\ a\ (\mathsf{g}\ j))\ a
\end{array}$$

Here we have:

```
let loop     = fun $n$ → fun f → fun $a$ →
    let rec doit = fun $a$ → fun $j$ → if $j \geq n$ then $a$
                                       else doit (f $a$ $j$) $(j + 1)$
    in  doit $a$ 0
```

The tail-recursive function loop corresponds to a *for* loop: The local data are collected in the accumulating parameter a while the functional parameter f determines how the new value for a after the jth iteration is computed from the old values of a and j. The function loop computes its result without a list as auxiliary data structure.

The applicability of the rules essentially depends on whether composition of the functions fold_left f a, map f, filter p are recognized. This structure, however, may occur rather indirectly in a given program. Subexpressions may be contained in *let* definitions or may be passed as parameters to the appropriate position. Application of the *let* optimizations of Sect. 3.2 may help here, or in more complex situations the value analysis of Sect. 3.5.

The principle of deforestation can be generalized in different directions:

- Other functions on lists can be considered besides the considered functions, like the function rev, which reverses the order of the elements of a list, the tail-recursive version rev_map of the function map, and the function fold_right (see Exercise 10).
- The suppression of intermediate data structures is also possible for index-dependent versions of the functions map and fold_left (see Exercise 11).
  Let l denote the list $[x_0;\ \ldots\ ; x_{n-1}]$ of type $'b$ **list**. The index-dependent version of map receives as argument a function f of type **int** $\rightarrow' b \rightarrow' c$ and returns for l the list:
  $$[\mathsf{f}\ 0\ x_0;\ \ldots\ ; \mathsf{f}\ (n-1)\ x_{n-1}]$$
  In analogy, the index-dependent version of the function fold_left receives as argument a function f of type **int** $\rightarrow' a \rightarrow' b \rightarrow' a$, an initial value a of type $'a$ and computes the value
  $$\mathsf{f}\ (n-1)\ (\ldots\ \mathsf{f}\ 1\ (\mathsf{f}\ 0\ a\ x_0)\ x_1\ \ldots)\ x_{n-1}$$
- The functions map and fold_left can, in full generality, be defined for user-defined functional data types, although this is not too frequently done. At least in principle, the same optimizations can be applied as we presented them for lists (see Exercise 12).

3.7 Improving the Evaluation Order: Strictness Analysis

Functional programming languages such as HASKELL delay the evaluation of expressions until their evaluation is strictly necessary. They evaluate the defining expressions of *let*-defined variables in the same way as the actual parameters of functions, namely only when an access to the values happens. Such a *lazy* evaluation allows for an elegant treatment of (potentially) infinite data structures. Each execution of a program working on such potentially infinite data structures will, in fact, only use a finite part for the computation of its result. The lazy evaluation of an expression e causes additional costs since a *closure* for e has to be constructed allowing for a later evaluation of e.

Example 3.7.1 Consider the following program:

$$\begin{array}{lllll}
\textbf{let rec}\ \mathsf{from} = \textbf{fun}\ n & \to & n :: \mathsf{from}\ (n+1) & & \\
\quad\textbf{and}\ \mathsf{take} = \textbf{fun}\ k & \to & \textbf{fun}\ s & \to & \textbf{if}\ k \leq 0\ \textbf{then}\ [\,] \\
& & & & \textbf{else}\ \textbf{match}\ s\ \textbf{with}\ [\,]\ \to\ [\,] \\
& & & & \qquad\mid\ h :: t\ \to\ h :: \mathsf{take}\ (k-1)\ t
\end{array}$$

Lazy evaluation of the expression $\mathsf{take}\ 5\ (\mathsf{from}\ 0)$ produces the list [0; 1; 2; 3; 4], while eager evaluation using call-by-value passing of parameters causes nontermination. □

Lazy evaluation, however, has its disadvantages. Even tail-recursive functions might not always only consume constant amount of space.

Example 3.7.2 Consider the following program fragment:

$$\begin{array}{lll}
\textbf{let rec}\ \mathsf{fac2} = \textbf{fun}\ x\ \to\ \textbf{fun}\ a\ \to & \textbf{if}\ x \leq 0\ \textbf{then}\ a \\
& \textbf{else}\ \mathsf{fac2}\ (x-1)\ (a \cdot x)
\end{array}$$

Lazy evaluation creates one closure for each multiplication in the accumulating parameter. The nested sequence of closures is only evaluated when the recursion arrives at the application $\mathsf{fac2}\ x\ 1$. It would be much more efficient to immediately evaluate the multiplication. □

It is often more efficient to eagerly evaluate an expression, thereby avoiding the construction of a closure, instead of delaying it. This is the aim of the following optimization.

For a simplification, we start with programs that neither use composed data structures nor higher-order functions. In addition, we assume that all functions are defined on the top level. To describe the transformation, we introduce a construct

$$\textbf{let\#}\ x = e_1\ \textbf{in}\ e_0$$

that forces the evaluation of the expression e_1 whenever the value of e_0 is needed. The goal of the optimization is to replace as many *let*-expressions as possible by *let#*-expressions without changing the termination properties of the program. *Strictness analysis* determines the necessary information about the termination properties of expressions. A k-place function f is called *strict* in its jth argument, $1 \leq j \leq k$, if the evaluation of the expression f $e_1 \ldots e_k$ does not terminate whenever the evaluation of e_j does not terminate. The evaluation of the jth argument e_j can be forced without changing the termination behavior if the function is strict in its jth argument. The compiler may then replace f $e_1 \ldots e_k$ by

$$\textbf{let\#}\ x = e_j\ \textbf{in}\ \mathsf{f}\ e_1 \ldots e_{j-1}\ x\ e_{j+1} \ldots e_k$$

Analogously, the compiler can replace a *let*-expression **let** $x = e_1$ **in** e_0 by the expression

$$\textbf{let\#}\ x = e_1\ \textbf{in}\ e_0$$

if the evaluation of e_0 does not terminate whenever the computation of e_1 does not terminate.

The simplest form of a strictness analysis only distinguishes whether the evaluation of an expression does definitely not terminate or maybe terminates and delivers a value. Let **2** be the finite lattice consisting of the two values 0 and 1, where $0 < 1$. The value 0 is associated with an expression whose evaluation does definitely not terminate. The value 1 denotes possible termination. A k-place function f is described by an abstract k-place function:

$$[\![\mathsf{f}]\!]^\sharp : \mathbf{2} \to \ldots \to \mathbf{2} \to \mathbf{2}$$

The fact $[\![\mathsf{f}]\!]^\sharp\ 1 \ldots 1\ 0\ 1 \ldots 1 = 0$ (0 in the jth argument) allows us to derive that an application of function f definitely does not terminate if the evaluation of the jth argument does not terminate. The function f, therefore, is *strict* in its jth argument.

We construct a system of equations to determine abstract descriptions $\mathsf{f}^\sharp$ for all functions f of the program. This needs the abstract evaluation of expressions as an auxiliary function. This abstract evaluation is defined with respect to a value binding ρ for the free variables of base types and a mapping ϕ of functions to their actual abstract descriptions:

$$\begin{array}{ll}
[\![b]\!]^\sharp\ \rho\ \phi & = 1 \\
[\![x]\!]^\sharp\ \rho\ \phi & = \rho\ x \\
[\![\Box_1\ e]\!]^\sharp\ \rho\ \phi & = [\![e]\!]^\sharp\ \rho\ \phi \\
[\![e_1\ \Box_2\ e_2]\!]^\sharp\ \rho\ \phi & = [\![e_1]\!]^\sharp\ \rho\ \phi\ \wedge\ [\![e_2]\!]^\sharp\ \rho\ \phi \\
[\![\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]^\sharp\ \rho\ \phi & = [\![e_0]\!]^\sharp\ \rho\ \phi\ \wedge\ ([\![e_1]\!]^\sharp\ \rho\ \phi\ \vee\ [\![e_2]\!]^\sharp\ \rho\ \phi) \\
[\![\mathsf{f}\ e_1\ \ldots\ e_k]\!]^\sharp\ \rho\ \phi & = \phi(\mathsf{f})\ ([\![e_1]\!]^\sharp\ \rho\ \phi)\ \ldots\ ([\![e_k]\!]^\sharp\ \rho\ \phi) \\
[\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ e]\!]^\sharp\ \rho\ \phi & = [\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto [\![e_1]\!]^\sharp\ \rho\})\ \phi \\
[\![\textbf{let\#}\ x_1 = e_1\ \textbf{in}\ e]\!]^\sharp\ \rho\ \phi & = ([\![e_1]\!]^\sharp\ \rho\ \phi)\ \wedge\ ([\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto 1\})\ \phi)
\end{array}$$

The abstract evaluation function $[\![.]\!]^\sharp$ interprets constants as the value 1. Variables are looked up in ρ. Unary operators $\Box_1$ are approximated by the identity function since the evaluation of an application does not terminate whenever the evaluation of the argument does not terminate. Binary operators are, analogously, interpreted as conjunction. The abstract evaluation of an *if*-expression is given by $b_0 \wedge (b_1 \vee b_2)$, where b_0 represents the abstract value of the condition and b_1, b_2 represent the abstract values for the two alternatives. The intuition behind this definition is that when evaluating a conditional expression the condition needs to be evaluated in any case while only one of the two alternatives must be evaluated. For a function application, the actual abstract value of the function is looked up in the function binding ϕ is and applied to the values determined recursively for the argument expressions. For a *let*-defined variable x in an expression e_0, first the abstract value for x is determined and then the value of the main expression e_0 with respect to this value. If the variable x is *let#*-defined, it must be ensured that the overall expression obtains the abstract value 0 when the value obtained for x is 0.

Example 3.7.3 Consider the expression e, given by

$$\begin{array}{l}\textbf{if } x \leq 0 \textbf{ then } a \\ \textbf{else } \ \mathsf{fac2}\ (x-1)\ (a \cdot x)\end{array}$$

For values $b_1, b_2 \in \mathbf{2}$, let ρ be the variable binding $\rho = \{x \mapsto b_1, a \mapsto b_2\}$. Mapping ϕ associates the function $\mathsf{fac2}$ with the abstract function $\textbf{fun}\ x \to \textbf{fun}\ a \to x \wedge a$. The abstract evaluation of e produces the value:

$$\begin{aligned}[\![e]\!]^\sharp\ \rho\ \phi &= (b_1 \wedge 1) \wedge (b_2 \vee (\phi\,\mathsf{fac2})\ (b_1 \wedge 1)\ (b_2 \wedge b_1)) \\ &= b_1 \wedge (b_2 \vee (b_1 \wedge b_2)) \\ &= b_1 \wedge b_2\end{aligned}$$

□

The abstract expression evaluation constructs for each function $\mathsf{f} = \textbf{fun}\ x_1 \to \ldots \to \textbf{fun}\ x_k \to e$ defined in the program the equations

$$\phi(\mathsf{f})\ b_1\ \ldots\ b_k = [\![e_i]\!]^\sharp\ \{x_j \mapsto b_j \mid j = 1, \ldots, k\}\ \ \phi$$

for all $b_1, \ldots, b_k \in \mathbf{2}$. The right side depends monotononically on the abstract values $\phi(\mathsf{f})$. Therefore, this system of equations possesses a least solution, denoted by $[\![\mathsf{f}]\!]^\sharp$.

Example 3.7.4 For the function $\mathsf{fac2}$ of Example 3.7.2 one obtains the equations

$$[\![\mathsf{fac2}]\!]^\sharp\ b_1\ b_2 = b_1 \wedge (b_2 \vee [\![\mathsf{fac2}]\!]^\sharp\ b_1\ (b_1 \wedge b_2))$$

Fixed-point iteration successively delivers for $[\![\mathsf{fac2}]\!]^\sharp$ the abstract functions:

0	$\textbf{fun}\, x \to \textbf{fun}\, a \to 0$
1	$\textbf{fun}\, x \to \textbf{fun}\, a \to x \wedge a$
2	$\textbf{fun}\, x \to \textbf{fun}\, a \to x \wedge a$

Note that the occurring abstract functions have been represented by boolean expressions instead of by their value tables. We conclude that function fac2 is strict in both its arguments. The definition of fac2 can thus be transformed into

```
let rec fac2 = fun x → fun a → if x ≤ 0 then a
                                else    let# x' = x − 1
                                     in let# a' = x · a
                                       in    fac2 x' a'
```

□

The analysis produces the precise results for this example. Correctness of the analysis follows since the abstract expression evaluation is an abstraction of the concrete expression evaluation as provided by the *denotational* semantics of our functional language. The denotational semantics uses a partial order for the integer numbers that consists of the set of numbers $\mathbb{Z}$ together with a special value $\perp$ that represents a nonterminating evaluation where the order relation is given by $\perp \sqsubset z$ for all $z \in \mathbb{Z}$. The *abstract* denotational semantics which we have constructed, however, interprets basic values and operators over the lattice **2**. Our description relation between concrete and abstract values is

$$\perp \,\Delta\, 0 \quad \text{and} \quad z \,\Delta\, 1 \quad \text{for } z \in \mathbb{Z}$$

The proof that the given analysis always delivers correct values is by induction over the fixed-point iteration.

In the following, the analysis is extended beyond values of base types to structured data. So far, the only distinction made is whether a function's argument is totally needed or not needed at all for computing the result. With structured data, functions may access their arguments to different depths.

Example 3.7.5 The function

```
let hd = fun l → match l with h :: t → h
```

only visits the topmost list constructor of its argument and returns its first element. The function length of Example 3.6.1, in contrast, needs all list constructors and the empty list at the end of the argument list to compute its result. □

We now consider programs that, besides basic values, manipulate lists and tuples. Accordingly, we extend the syntax of expressions by admitting the corresponding constructors for structured data.

$$\begin{array}{rcl} e & ::= & \ldots \mid [\,] \mid e_1 :: e_2 \mid \textbf{match}\ e_0\ \textbf{with}\ [\,]\ \to\ e_1\ \mid\ h :: t\ \to\ e_2 \\ & & \mid (e_1, \ldots, e_k) \mid \textbf{match}\ e_0\ \textbf{with}\ (x_1, \ldots, x_k)\ \to\ e_1 \end{array}$$

The first question we like to answer is whether functions access topmost constructors of their arguments or not. A function f is *root-strict* in its ith argument if the topmost constructor of the ith argument is needed to compute the topmost constructor of an application of function f. For basic values, root-strictness agrees with strictness, as considered so far. As with strictness for basic values, we use a construct **let#** $x = e_1$ **in** e_0 that evaluates the value of x up to the root constructor before evaluating the root constructor of e_0.

As with strictness properties of functions on basic values we describe root-strictness using boolean functions. The value 0 represents only the concrete value $\perp$ (nonterminating computation), the value 1, in contrast, represents all other values, for instance, the list [1; 2] as well as the partially computed lists $[1; \perp]$ and $1 :: \perp$.

Beyond the strictness analysis for basic values, we extend the abstract evaluation function $[\![e]\!]^\sharp\ \rho\ \phi$ by rules for lists, tuples, and case distinction;

$$\begin{array}{l} [\![\textbf{match}\ e_0\ \textbf{with}\ [\,]\ \to\ e_1\ \mid\ h :: t\ \to\ e_2]\!]^\sharp\ \rho\ \phi = \\ \qquad\qquad [\![e_0]\!]^\sharp\ \rho\,\phi\ \wedge\ ([\![e_1]\!]^\sharp\ \rho\,\phi \vee\ [\![e_2]\!]^\sharp\ (\rho \oplus \{h, t \mapsto 1\}))\ \phi \\ [\![\textbf{match}\ e_0\ \textbf{with}\ (x_1, \ldots, x_k)\ \to\ e_1]\!]^\sharp\ \rho\ \phi = \\ \qquad\qquad\qquad\qquad [\![e_0]\!]^\sharp\ \rho\ \phi\ \wedge\ [\![e_1]\!]^\sharp\ (\rho \oplus \{x_1, \ldots, x_k \mapsto 1\})\ \phi \\ [\![[\,]]\!]^\sharp\ \rho\ \phi\ =\ [\![e_1 :: e_2]\!]^\sharp\ \rho\ \phi\ =\ [\![(e_1, \ldots, e_k)]\!]^\sharp\ \rho\ \phi = 1 \end{array}$$

The abstract evaluation of an expression returns the abstract value 1 if the expression already provides the topmost constructor of the result. A *match*-expression for lists is abstractly evaluated in analogy to an *if*-expression. In the case of a composed list, the analysis does not know anything about the values of the two newly introduced variables. These are, therefore, described by the value 1. The abstract evaluation of a *match*-expression for tuples corresponds to the conjunction of the abstract value of the expression e_0 with the value for the body e_1, where the newly introduced variables are bound to 1.

Example 3.7.6 Let us check our analysis with the example function app, which concatenates two lists:

$$\begin{array}{l} \textbf{let rec}\ \mathsf{app} = \textbf{fun}\ x\ \to\ \textbf{fun}\ y\ \to\ \textbf{match}\ x\ \textbf{with}\ [\,]\ \to\ y \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid\ h :: t\ \to\ h :: \mathsf{app}\ t\ y \end{array}$$

Abstract interpretation establishes the equations:

$$\begin{array}{rl} [\![\mathsf{app}]\!]^\sharp\ b_1\ b_2 & = b_1\ \wedge\ (b_2 \vee 1) \\ & = b_1 \end{array}$$

for values $b_1, b_2 \in \mathbf{2}$. We conclude that the root constructor of the first argument is definitely needed for the computation of the root constructor of the result. □

In many applications, not only is the root constructor of the result value needed, but the whole value. A strictness analysis concentrating on that property tries to find out which arguments of a function are *totally* needed if the result of the function is *totally* needed. This generalization of strictness on basic values to structured values is called *total* strictness. The abstract value 0 now describes all concrete values that definitely contain a $\perp$, while 1 still describes all values.

The total-strictness properties of functions are again described by boolean functions. The rules for the evaluation of strictness for expressions without structured data are again extended to constructors for tuples, lists, and *match*-expressions:

$$\begin{array}{lll}
[\![\mathbf{match}\, e_0\, \mathbf{with}\, [\,] \to e_1 \mid h :: t \to e_2]\!]^\sharp\ \rho\ \phi & = & \mathbf{let}\ b = [\![e_0]\!]^\sharp\ \rho\ \phi \\
& & \mathbf{in}\ b \wedge [\![e_1]\!]^\sharp\ \rho\ \phi \\
& & \vee\ [\![e_2]\!]^\sharp\ (\rho \oplus \{h \mapsto b, t \mapsto 1\}\ \phi \\
& & \vee\ [\![e_2]\!]^\sharp\ (\rho \oplus \{h \mapsto 1, t \mapsto b\}\ \phi \\
[\![\mathbf{match}\, e_0\, \mathbf{with}\, (x_1, \ldots, x_k) \to e_1]\!]^\sharp\ \rho\ \phi & = & \mathbf{let}\ b = [\![e_0]\!]^\sharp\ \rho\ \phi \\
& & \mathbf{in}\ [\![e_1]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto b, x_2, \ldots, x_k \mapsto 1\})\ \phi \\
& & \vee \ldots \vee\ [\![e_1]\!]^\sharp\ (\rho \oplus \{x_1, \ldots, x_{k-1} \mapsto 1, x_k \mapsto b\})\ \phi \\
[\![\,[\,]\,]\!]^\sharp\ \rho\ \phi & = & 1 \\
[\![e_1 :: e_2]\!]^\sharp\ \rho\ \phi & = & [\![e_1]\!]^\sharp\ \rho\ \phi\ \wedge\ [\![e_2]\!]^\sharp\ \rho\ \phi \\
[\![(e_1, \ldots, e_k)]\!]^\sharp\ \rho\ \phi & = & [\![e_1]\!]^\sharp\ \rho\ \phi\ \wedge \ldots \wedge\ [\![e_k]\!]^\sharp\ \rho\ \phi \\
[\![\mathbf{let\#}\ x_1 = e_1\ \mathbf{in}\ e]\!]^\sharp\ \rho\ \phi & = & [\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto [\![e_1]\!]^\sharp\ \rho\ \phi\})\ \phi
\end{array}$$

In the analysis of total strictness, constructor applications need to be treated differently from how they were treated in the analysis of root-strictness. The application of a data constructor is now interpreted as the *conjunction* of the abstract values obtained for the components. For the evaluation of *let#*-expressions, we recall that the eagerly evaluated expression is only evaluated to its root constructor. The value obtained this way might contain $\perp$ without causing nontermination of the evaluation of the whole expression. The abstract evaluation of a *let#*-expression is, therefore, not different from the abstract evaluation of a *let*-expression. The decomposition of the value of an expression by applying a *match*-construct has also changed: The abstract value of the expression e_0, to which the pattern is matched, is not 0 if it evaluates to the empty list. This case thus corresponds to the conjunction of the abstract values of e_0 and the abstract value of the expression for the case of an empty list. Two cases must be considered if the expression e_0 evaluates to a composed list.

If the expression e_0 produces the value 1 only this value 1 can be assumed for all components of the list.

If the abstract evaluation of e_0 returns 0, either the first element or the rest of the list must contain $\perp$. So, either the local variable h or the local variable t must obtain the value 0. Let b be the value of the expression e_0. These two cases can be compactly combined by disjunction of the results obtained through abstract evaluation of the expression for composed lists, where for the newly introduced local variables h, t

the values b, 1 and 1, b, resp., are substituted. A similar disjunction is employed in the abstract evaluation of *match*-expressions for tuples. If a tuple is described by 0, then this is the case when it contains $\perp$, so at least one of its components must also contain $\perp$. This component can be described by 0. If a tuple is described by 1, then nothing is known about its components, and therefore all of them must be described by 1.

Example 3.7.7 We test our approach to the analysis of total strictness again with the function app of Example 3.7.6. Abstract interpretation establishes the equations:

$$\begin{aligned}[\![\mathsf{app}]\!]^\sharp\, b_1\, b_2 &= b_1 \wedge b_2 \vee b_1 \wedge [\![\mathsf{app}]\!]^\sharp\, 1\, b_2 \vee 1 \wedge [\![\mathsf{app}]\!]^\sharp\, b_1\, b_2 \\ &= b_1 \wedge b_2 \vee b_1 \wedge [\![\mathsf{app}]\!]^\sharp\, 1\, b_2 \vee [\![\mathsf{app}]\!]^\sharp\, b_1\, b_2\end{aligned}$$

for $b_1, b_2 \in \mathbf{2}$. Fixed-point iteration produces the following approximations of the least fixed point:

0	$\mathbf{fun}\,x \to \mathbf{fun}\,y \to 0$
1	$\mathbf{fun}\,x \to \mathbf{fun}\,y \to x \wedge y$
2	$\mathbf{fun}\,x \to \mathbf{fun}\,y \to x \wedge y$

We conclude that both arguments are totally needed if the result is totally needed. □

Whether the value of an expressions is *totally* needed depends on the context of the expression. For a function f, which possibly occurs in such a context, a variant f# is required that computes its result possibly more efficiently than f. For simplicity, we only consider functions f that totally need the values of *all* their arguments to compute the result totally. For the implementation of the function f#, we then assume that their arguments have already been totally evaluated. The implementation must then guarantee that this also holds for all recursive applications of any variants g# and that the result is also already totally evaluated.

For the function app the variant app# can be implemented in the following way:

```
let rec app# = fun x → fun y → match x with [ ] → y
                                 | h :: t →    let# t1 = app# t y
                                             in let# r = h :: t1
                                             in  r
```

We assume here that no closures are constructed for variables. A general transformation that systematically exploits information about total strictness is the subject of Exercise 13.

The programming language fragment for which we have developed several strictness analyses is very restricted. In the following, we briefly sketch how to at least partly lift these restrictions.

A first assumption was that all functions are defined on the topmost level. Each program of our OCaml-fragment can be transformed such that this property holds, see Exercise 15. Alternatively, we associate all free variables of a local function with the value 1 (don't know) in the analysis of this local function. This may produce imprecise information, but at least gives correct results.

Further, we constrained ourselves to k-place functions without functional arguments and results, and without partial applications. This is because the complete lattice of the k-place monotonic abstract functions $\mathbf{2} \to \ldots \to \mathbf{2}$ possesses properly ascending chains whose length is exponential in k, that is, the number of arguments. This is still acceptable since the number of arguments is often not very large. The number of elements of this lattice, however, is even doubly exponential in k. And this number matters if higher order functions are to be analyzed. One way out of this dilemma consists in abstracting the abstract-function domains radically by smaller complete lattices. For example, one could use the boolean lattice $\mathbf{2}$ for function domains: 0 represents the constant 0-function. This strong abstraction will not lead to precise strictness information for programs that systematically employ higher-order functions. Some of the higher-order functions, however, may be removed by function specialization as shown in Sect. 3.4 and thereby improve the chances for strictness analysis.

Strictness analysis, as we have considered it, is only applicable to monomorphically typed functions and monomorphic instances of polymorphically typed functions.

In general, the programmer might have a hard time finding out when the compiler is able to determine and exploit strictness information, and when it fails. The programming language HASKELL therefore provides *annotations*, which allow the programmer to force the evaluation of expressions whenever he considers it important for efficiency reasons.

3.8 Exercises

1. *let*-**Optimization**
   Consider the following equation:

$$(\mathbf{fun}\ y \to \mathbf{let}\ x = e\ \mathbf{in}\ e_1) = (\mathbf{let}\ x = e\ \mathbf{in}\ \mathbf{fun}\ y \to e_1)$$

if y does not occur free in e.

(a) Give conditions under which the expressions on both sides are semantically equivalent.
(b) Give conditions under which the application of this equation from left to right may contribute to an increase in efficiency of evaluation.

2. *letrec*-**Optimization**
   This exercise tries to extend the optimizations of *let*-expressions to optimizations of *letrec*-expressions.

   (a) Give rules how *let*-definitions can be moved out of *letrec*-expressions.

   (b) Give constraints under which your transformations are semantics preserving.
   (c) Give constraints under which your transformations lead to an improvement in efficiency.
   (d) Test your optimizations with some example programs.

3. ***let*-Optimization of *if*-expressions**
   What do you think of the following rules?

$$(\textbf{if let } x = e \textbf{ in } e_0 \textbf{ then } e_1 \textbf{ else } e_2) = (\textbf{let } x = e \textbf{ in if } e_0 \textbf{ then } e_1 \textbf{ else } e_2)$$
$$(\textbf{if } e_0 \textbf{ then let } x = e \textbf{ in } e_1 \textbf{ else } e_2) = (\textbf{let } x = e \textbf{ in if } e_0 \textbf{ then } e_1 \textbf{ else } e_2)$$

   where x does not occur free in the expressions e_0, e_1, and e_2.
4. **Regular tree grammar**
   A *regular tree grammar* is a tuple $G = (N, T, P)$, where N is a finite set of *nonterminal symbols*, T is a finite set of *terminal constructors*, and P is a set of *rules* of the form $A \Rightarrow t$, and where t is a term that is built from nonterminal symbols in N using constructors in T. The *language* $\mathcal{L}_G(A)$ of the regular tree grammar G for a nonterminal A is the set of all terminal expressions t derivable from nonterminal A using rules from P. An expression is called *terminal* if no nonterminal occurs in it.
   Give regular tree grammars for the following set of trees:

   (a) all lists (inner nodes : "::") with an even number of elements from {0, 1, 2};
   (b) all lists with elements from {0,1,2} such that the sum of the elements is even;
   (c) all terms with inner nodes :: and leaves {0,1,2} or [ ] that are of type list list int.

5. **Tree grammar (cont.)**
   Let G be a regular tree grammar of size n and A a nonterminal symbol of G. Show:

   (a) $\mathcal{L}_G(A) \neq \emptyset$ if and only if $t \in \mathcal{L}_G(A)$ for a t of depth $\leq n$;
   (b) $\mathcal{L}_G(A)$ is infinite if and only if $t \in \mathcal{L}_G(A)$ for a t of depth d where $n \leq d < 2n$.
   (In particular, define the "size of a grammar" in such a way that these claims hold.)

6. **Value analysis: case distinction**
   Modify the algorithm for the value analysis in such a way that it respects the given order of the patterns in a case distinction.
7. **Value analysis: exceptions**
   Consider the functional core language, extended by the constructs:

$$e \;::=\; \ldots \mid \textbf{raise } e \mid (\textbf{try } e \textbf{ with } p_1 \to e_1 \mid \ldots \mid p_k \to e_k)$$

The expression **raise** e throws an exception of value e, while a *try*-expression evaluates the main expression e, and if the computation ends with an exception then v catches this exception if one of the patterns p_i matches v, and throws the same exception, otherwise.
How should one modify the value analysis to identify the set of exceptions which the evaluation of an expression may possibly throw?

8. **Value analysis: references**
Consider the functional core language, extended by destructively modifiable references:

$$e \ ::= \ \ldots \mid \textbf{ref}\ e \mid (e_1 := e_2) \mid !e$$

Extend the value analysis to apply it to this extended language. How could one find out by help of this analysis whether an expression is *pure*, i.e., that its evaluation does not modify references?

9. **Simplification rules for the identity**
Let id= **fun** $x \rightarrow x$. Give a set of rules to simplify expressions containing id!

10. **Simplification rules for rev**
The list of functions considered by deforestation can be further extended.

    (a) Define functions rev, fold_right, rev_map, rev_tabulate and rev_loop, where rev reverses a list. The following equalities hold for the other functions:

    $$\begin{aligned}\text{fold\_right f } a &= \text{comp (fold\_left f a) rev}\\ \text{rev\_map f} &= \text{comp (map f) rev}\\ \text{rev\_tabulate } n &= \text{comp rev tabulate } n\end{aligned}$$

    rev_loop n should behave like loop n, but the iteration should run from $n - 1$ to 0 instead of the other way round.

    (b) Design rules for the composition of these functions and for these functions with map, fold_left, filter, and tabulate. Exploit that comp rev rev is the identity.

    (c) Explain under which circumstances these rules are applicable, and why they improve the efficiency.

11. **Simplification rules for index-dependent functions**
Define index-dependent variants of the functions map and fold_left. Design simplification rules and argue under which conditions these are applicable!
How do the new functions behave at composition with map, fold_left, filter and tabulate?

12. **Simplification rules for general data structures**
Elimination of intermediate data structures may not only pay off for lists.

    (a) Design simplification rules for functions map and fold_left on tree-like data structures. The function map is to apply a functional argument to all data elements contained in the data structure. fold_left is to combine all elements in the data structure in one value using its functional argument.

(b) Give examples for your proposed functions and discuss their applicability. What could a generalization of tabulate from lists to tree-like data structures look like?

(c) Do you encounter an analogy to the list function filter? Define functions to_list and from_list that convert your data structure into a list and reconstruct a list from your data structure, respectively. What simplification rules exist for these functions?

13. **Optimization for total strictness**
Develop a transformation that optimizes an expression whose value is definitely completely needed.

14. **Combination of total and root strictness**
The strictness analyses which we have designed are instances of a neededness analysis which more precisely determines how much of the arguments of a function is needed for satisfying various demands onto the function's result.

(a) Define a strictness analysis that simultaneously determines total-strictness and root-strictness information. Use a complete lattice $\mathbf{3} = \{0 < 1 < 2\}$.

(b) Define a description relation between concrete values and abstract values from **3** and define the necessary abstract expression evaluation.

(c) Test your analysis at the function app.

(d) Generalize your analysis to an analysis that determines, for a given $k \geq 1$, up to which depth $\leq k - 1$ the arguments need to be evaluated to or whether they need to be completely evaluated if the result needs to be evaluated up to a depth $0 \leq j \leq k - 1$ or completely.

15. **Moving local functions to the outermost level**
Transform a given OCAML program in such a way that all functions are defined on the outermost level.

16. **Monotone functions over 2**
Construct the following complete lattices of monotonic functions:

(a) $\mathbf{2}\to\mathbf{2}$;
(b) $\mathbf{2}\to\mathbf{2}\to\mathbf{2}$;
(c) $(\mathbf{2}\to\mathbf{2})\to\mathbf{2}\to\mathbf{2}$.

17. **Strictness analysis for higher-order functions**
Analyze total strictness properties of monomorphic instances of the functions map and fold_left with types:

$$\begin{array}{ll} \mathsf{map} & : (\mathbf{int} \to \mathbf{int}) \to \mathsf{list}\ \mathbf{int} \to \mathsf{list}\ \mathbf{int} \\ \mathsf{fold\_left} & : (\mathbf{int} \to \mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathsf{list}\ \mathbf{int} \to \mathsf{list}\ \mathbf{int} \end{array}$$

3.9 Literature

The λ calculus with β-reduction and α-conversion is the theoretical basis for functional programming languages. It is based on work about the foundations of mathematics by Church and Kleene from the 1930s. The book by Barendregt (1984) is still the standard textbook presenting important properties of the λ-calculus.

Jones and Santos (1998) gives a survey of optimizations implemented in the HASKELL compiler. This description also treats optimizations of nested *let*-expressions (Jones and Santos 1996).

fold/*unfold*-transformations are first treated in Burstall and Darlington (1977). Inlining and function specialization are simple forms of *partial evaluation* of programs (Secher and Sørensen 1999). Our value analysis follows the approach taken by Heintze (1994). A type-based analysis of side effects is proposed by Amtoft et al. (1997).

The idea to systematically suppress intermediate data structures was proposed by Wadler (1990). An extension to programs with higher-order functions is described in Seidl and Sørensen (1998). The particularly simple variant for programs with lists presented here was introduced by Gill et al. (1993). Generalizations to arbitrary algebraic data structures are studied by Takano and Meijer (1995).

The idea to use strictness analysis to support a conversion from CBN to CBV originates with Mycroft (1980). A generalization to monomorphic programs with higher-order functions is presented by Burn et al. (1986). The method for the analysis of total strictness for programs with structured data described here is a simplification of the method of Sekar et al. (1990).

Not treated in this chapter are optimizations that are based on the representation of functional programs in continuation-passing style. An extensive presentation of this technique is given by Appel (2007).

References

P. Anderson, D. Binkley, G. Rosay, T. Teitelbaum, Flow insensitive points-to sets. Inf. Softw. Technol. **44**(13), 743–754 (2002)

W.A. Appel, M. Ginsburg, *Modern Compiler Implementation in C* (Cambridge University Press, Cambridge, 2004)

S. Abramsky, C. Hankin (Hrsg.), *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, 1987)

A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd revised edn. (Addison-Wesley, New York, 2007)

T. Amtoft, F. Nielson, H.R. Nielson, Type and behaviour reconstruction for higher-order concurrent programs. J. Funct. Program. **7**(3), 321–347 (1997)

W.A. Appel, *Compiling with Continuations* (Cambridge University Press, Cambridge, 2007)

D.F. Bacon, Fast and effective optimization of statically typed object-oriented languages. Ph.D. thesis, Berkeley, 1997

H.P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, Volume 103 of Studies in Logic and the Foundations of Mathematics, revised edition (North Holland, Amsterdam, 1984)

R.M. Burstall, J. Darlington, A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)

G.L. Burn, C. Hankin, S. Abramsky, Strictness analysis for higher-order functions. Sci. Comput. Program. **7**(3), 249–278 (1986)

P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in *2nd International Symposium on Programming*, pp. 106–130. Dunod, Paris, France, 1976

P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *4th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 238–252, 1977a

P. Cousot, R. Cousot, Static determination of dynamic properties of recursive procedures, ed. by E.J. Neuhold (Hrsg.), in *IFIP Conference on Formal Description of Programming Concepts*, pp. 237–277. (North Holland, Amsterdam, 1977b)

P. Cousot, R. Cousot, Systematic design of program transformation frameworks by abstract interpretation, in *29th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 178–190, 2002

J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, S. Midkiff, Escape analysis for Java. SIGPLAN Not. **34**(10), 1–19 (1999)

P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in *5th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 84–97, 1978

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd edn. (MIT Press, Cambridge, 2009)

K.D. Cooper, L. Torczon, *Engineering a Compiler* (Morgan Kaufmann, Massachusetts, 2004)

M. Fähndrich, J. Rehof, M. Das, Scalable context-sensitive flow analysis using instantiation constraints. SIGPLAN Not. **35**(5), 253–263 (2000)

C. Fecht, H. Seidl, A faster solver for general systems of equations. Sci. Comput. Program. (SCP) **35**(2), 137–161 (1999)

A.J. Gill, J. Launchbury, S.L.P. Jones, A short cut to deforestation, in *Functional Programming and Computer Architecture (FPCA)*, pp. 223–232, 1993

R. Giegerich, U. Möncke, R. Wilhelm, Invariance of approximate semantics with respect to program transformations. In *GI Jahrestagung*, pp. 1–10, 1981

P. Granger, Static analysis of linear congruence equalities among variables of a program, in *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, pp. 169–192. LNCS 493 (Springer, Heidelberg, 1991)

T. Gawlitza, H. Seidl, Precise fixpoint computation through strategy iteration, in *European Symposium on Programming (ESOP)*, pp. 300–315. LNCS 4421 (Springer, Heidelberg, 2007)

M.S. Hecht, *Flow Analysis of Computer Programs* (North Holland, Amsterdam, 1977)

N. Heintze, Set-based analysis of ML programs. SIGPLAN Lisp Pointers **VII**(3), 306–317 (1994)

M. Hofmann, A. Karbyshev, H. Seidl, in *Verifying a Local Generic Solver in Coq*, ed. by R. Cousot, M. Martel. Static Analysis, Volume 6337 of Lecture Notes in Computer Science (Springer, Heidelberg, 2010), pp. 340–355

M. Hofmann, A. Karbyshev, H. Seidl, *What is a Pure Functional?* ed. by S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, P.G. Spirakis (Hrsg.), ICALP (2), Volume 6199 of Lecture Notes in Computer Science (Springer, Heidelberg, 2010), pp. 199–210

S.L.P. Jones, W. Partain, A. Santos, Let-floating: moving bindings to give faster programs, in *International Conferece on Functional Programming (ICFP)*, pp. 1–12, 1996

L. Simon, P. Jones, A.L.M. Santos, A Transformation-based optimiser for Haskell. Sci. Comput. Program. **32**(1–3), 3–47 (1998)

M. Karr, Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)

G.A. Kildall, A unified approach to global program optimization, in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 194–206, 1973

J. Knoop, *Optimal Interprocedural Program Optimization, A New Framework and Its Application*, LNCS 1428. (Springer, Berlin, 1998)

J. Knoop, O. Rüthing, B. Steffen, Optimal code motion: theory and practice. ACM Trans. Program. Lang. Syst. **16**(4), 1117–1155 (1994)

J. Knoop, O. Rüthing, B. Steffen, Partial dead code elimination, in *ACM Conference on Programming Languages Design and Implementation (PLDI)*, pp. 147–158, 1994

J. Knoop, B. Steffen, The interprocedural coincidence theorem, in *4th International Conference on Compiler Construction (CC)*, pp. 125–140. LNCS 541 (Springer, Heidelberg, 1992)

S. Kundu, Z. Tatlock, S. Lerner, Proving optimizations correct using parameterized program equivalence, in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009

J.B. Kam, J.D. Ullman, Global data flow analysis and iterative algorithms. J. ACM **23**(1), 158–171 (1976)

J.B. Kam, J.D. Ullman, Monotone data flow analysis frameworks. Acta Inf. **7**, 305–317 (1977)

X. Leroy, Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)

S. Lerner, T.D. Millstein, C. Chambers, Automatically proving the correctness of compiler optimizations, in *ACM SIGPLAN Conference on Programming Language Design and Implementatio (PLDI)*, pp. 220–231, 2003

S. Lerner, T. Millstein, E. Rice, C. Chambers, Automated soundness proofs for dataflow analyses and transformations via local rules, in *32nd ACM Symp. on Principles of Programming Languages (POPL)*, pp. 364–377, 2005

D. Liang, M. Pennings, M.J. Harrold, Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java, in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE),* pp. 73–79, 2001

F. Martin, M. Alt, R. Wilhelm, C. Ferdinand, Analysis of loops, in *7th International Conference on Compiler Construction (CC)*, pp. 80–94. LNCS 1383, Springer, 1998.

S.S. Muchnick, N.D. Jones (Hrsg.), *Program Flow Analysis: Theory and Application* (Prentice Hall, Englewood Cliffs, 1981)

M. Müller-Olm, H. Seidl, Precise interprocedural analysis through linear algebra, in *31st ACM Symposium on Principles of Programming Languages (POPL)*, pp. 330–341, 2004.

M. Müller-Olm, H. Seidl, A generic framework for interprocedural analysis of numerical properties, in *Static Analysis, 12th International Symposium (SAS)*, pp. 235–250. LNCS 3672 (Springer, Heidelberg, 2005)

M. Müller-Olm, H. Seidl, Analysis of modular arithmetic. ACM Trans. Program. Lang. Syst. **29**(5), 2007

S.S. Muchnick, *Advanced Compiler Design and Implementation* (Morgan Kaufmann, Massachusetts, 1997)

A. Mycroft, The theory and practice of transforming call-by-need into call-by-value, in *Symposium on Programming: Fourth 'Colloque International sur la Programmation'*, pp. 269–281. LNCS 83 (Springer, Heidelbeg, 1980)

F. Nielson, H.R. Nielson, C. Hankin, *Principles of Program Analysis* (Springer, Heidelberg, 1999)

R. Paige, Symbolic finite differencing—Part I, in *3rd European Symposium on Programming (ESOP)*, pp. 36–56. LNCS 432 (Springer, Heidelberg, 1990)

R. Paige, J.T. Schwartz, Reduction in strength of high level operations, in *4th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 58–71, 1977

G. Ramalingam, On loops, dominators, and dominance frontiers. ACM Trans. Program. Lang. Syst. (TOPLAS) **24**(5), 455–490 (2002)

V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, C. Godin, Practical virtual method call resolution for Java, in *15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 264–280, 2000

A. Simon, *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities* (Springer, Heidelberg, 2008)

J. Sheldon, W. Lee, B. Greenwald, S.P. Amarasinghe, Strength reduction of integer division and modulo operations, in *Languages and Compilers for Parallel Computing, 14th International Workshop (LCPC). Revised Papers*, pp. 254–273. LNCS 2624 (Springer, Heidelberg, 2003)

M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, ed. by S.S. Muchnick, N.D. Jones (Hrsg.). *Program Flow Analysis: Theory and Application*, pp. 189–234 (Prentice Hall, Englewood Cliffs, 1981)

R.C. Sekar, S. Pawagi, I.V. Ramakrishnan, Small domains spell fast strictness analysis, in *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 169–183, 1990

M. Sagiv, T.W. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, in *26th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 105–118, 1999

M.Sagiv, T.W. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. (TOPLAS) **24**(3), 217–298 (2002)

H. Seidl, M.H. Sørensen, Constraints to stop deforestation. Sci. Comput. Program. **32**(1–3), 73–107 (1998)

J.P. Secher, M.H. Sørensen, On perfect supercompilation, in *3rd Int. Andrei Ershov Memorial Conference: Perspectives of System Informatics (PSI)*, pp. 113–127. LNCS 1755 (Springer, Heidelberg, 1999)

Y.N. Srikant, P. Shankar (Hrsg.), *The Compiler Design Handbook: Optimizations and Machine Code Generation* (CRC Press, Boca Raton, 2003)

B. Steensgaard, Points-to analysis in almost linear time, in *23rd ACM Symposium on Principles of Programming Languages (POPL)*, pp. 32–41, 1996

J.-B. Tristan, X. Leroy, Verified validation of lazy code motion, in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 316–326, 2009

A. Takano, E. Meijer, Shortcut deforestation in calculational form, in *SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 306–313, 1995

P. Wadler, Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (1990)

Index