Foreword

The typical development of a successful theory in computer science traverses three sometimes overlapping phases: an *experimental phase*, where phenomena are studied almost in a trial and error fashion, a busy *phase of realization*, where people use the results of the experimental phase in an "uncoordinated" fashion, and a *contemplative phase*, where people look for the essence of what has been previously achieved. In *compiler optimization* these three phases currently coexist. New heuristics are still being proposed and purely evaluated on some benchmarks, and known techniques are still being implemented specifically for a new operating system or variants of programming languages, but increasingly many attempts now try to understand the full picture of compiler optimization in order to develop general frameworks and generators.

This monograph is a typical contribution to third phase activities in that it presents a uniform framework capturing a large class of imperative programming languages and their corresponding transformations, together with directions for cookbook style implementation. Thus besides clarifying appropriateness and limitations of the considered methods it also tries to open these methods even to non-experts.

More technically, the monograph adresses the issue of extension: which principles are stable, i.e., remain valid when extending intraprocedurally successful methods to the interprocedural case, and what needs to be done in order to overcome the problems and anomalies arising from this extension. This investigation characterizes the power and flexibility of procedure mechanisms from the data flow analysis point of view.

Even though all the algorithms considered evolve quite naturally from basic principles, which directly leads to accessible correctness and optimality considerations, they often outperform their "tricky" handwritten counterpart. Thus they constitute a convincing example for the superiority of *concept-driven* software development.

The monograph presents a full formal development for so-called *syntactic* program analysis and transformation methods including complete proofs, which may be quite hard to digest in full detail. This rigorous development, on purpose structurally repetitive, is tailored to stress similarities and differences between the intraprocedural and interprocedural setting, down to the very last detail. However, the reader is not forced to follow the technical level. Rather, details can be consulted on demand, providing students with a deep yet intuitive and accessible introduction to central principles of intraprocedural and interprocedural optimization, compiler experts with precise information about the obstacles when moving from the intraprocedural to the interprocedural case, and developers with concise specifications of easy to implement yet high-performance interprocedural analyses.

Summarizing, this thesis can be regarded as a comprehensive account of what, from the practical point of view, are the most important program analysis and transformation methods for imperative languages. I therefore recommend it to everybody interested in a conceptual, yet far reaching entry into the world of optimizing compilers.

Bernhard Steffen

Prologue

The present monograph is based on the doctoral dissertation of the author [Kn1]. It presents a new framework for optimal *interprocedural program optimization*, which covers the full range of language features of imperative programming languages. It captures programs with (mutually) recursive procedures, global, local, and external variables, value, reference, and procedure parameters. In spite of this unique generality, it is tailored for practical use. It supports the design and implementation of provably optimal program optimizations in a cookbook style. In essence, this is achieved by decomposing the design process of a program optimization and the proof of its optimality with respect to a specific optimality criterion into a small number of elementary steps, which can independently be proved using only knowledge about the specification of the optimization. This contrasts with heuristically based approaches to program optimization, which are still dominant in practice, and often ad hoc. The application of the framework is demonstrated by means of the *computationally* and *lifetime optimal* elimination of *partially redundant computations* in a program, a practically relevant optimization, whose intraprocedural variant is part of many advanced compiler environments. The purpose of considering its interprocedural counterpart is twofold. On the one hand, it demonstrates the analogies between designing intraprocedural and interprocedural optimizations. On the other hand, it reveals essential differences which must usually be faced when extending intraprocedural optimizations interprocedurally. Optimality criteria satisfiable in the intraprocedural setting can impossible to be met in the interprocedural one. Optimization strategies being successful in the intraprocedural setting can fail interprocedurally. The elimination of partially redundant computations is well-suited for demonstration. In contrast to the intraprocedural setting, computational and lifetime optimal results are in general impossible in the interprocedural setting. The placement strategies leading to computationally and lifetime optimal results in the intraprocedural setting, can even fail to guarantee profitability in the interprocedural setting. We propose a natural constraint applying to a large class of programs, which is sufficient for the successful transfer of the intraprocedural elimination techniques to the interprocedural setting. Under this constraint, the resulting algorithm generates interprocedurally computationally and lifetime optimal results, making it unique. It is not only more powerful than its heuristic predecessors but also more efficient, and reduces in the absence of procedures to its intraprocedural counterpart.

The remainder of this prologue summarizes the background of this monograph, and provides a brief introduction to program optimization intended to make its presentation more easily amenable to novice readers in the field.

Optimizing Compilers. In essence, a *compiler* is a program translating programs of some source language $\mathcal{L}_1$ into semantically equivalent programs of some target language $\mathcal{L}_2$. One of the most typical applications of a compiler is the translation of a source program written in a high-level programming language into a machine program (often simply called "machine code" or just "code"), which can be executed on the computer the compiler is implemented on. Of course, compilers are expected to produce highly efficient code, which has led to the construction of *optimizing compilers* [ASU, WG, Mor].

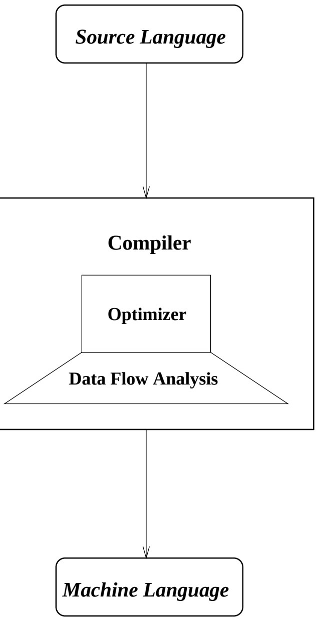

Fig. 1.1. Structure of an optimizing compiler

Figure 1.1 illustrates the general structure of an optimizing compiler. The central component is called an *optimizer*. Basically, this is a program designed for detecting and removing inefficiencies in a program by means of appropriate performance improving transformations. Traditionally, these transformations are called *program optimizations*. This general term, however, is slightly misleading because program optimization cannot usually be expected to transform a program of "bad" performance into a program of "good" or even "optimal" performance. There are two quite obvious reasons for this limitation. First, "bad," "good," and "optimal" are qualitative properties lacking a (precise) quantitative meaning. Second, interpreting the term optimization naively, does not impose any restrictions on the kind of transformations considered possible; restrictions, for example, which are usually imposed by automation requirements. Following the naive interpretation, optimization would require replacing a sorting algorithm of quadratic time complexity by a completely different sorting algorithm where the second factor is replaced by a logarithmic one. Optimizations of this kind would require a profound understanding of the semantics of the program under consideration, which is usually far beyond the capabilities of an automatic analysis.

The original domain of program optimization is different. Usually, it leaves the inherent structure of the algorithms invariant, and improves their performance by *avoiding* or *reducing* the computational effort at run-time, or by *shifting* it from the run-time into the compile-time. Typical examples are *loop invariant code motion*, *strength reduction*, and *constant folding*. Loop invariant code motion moves computations yielding always the same value inside a loop to a program point outside of it, which avoids unnecessary recomputations of the value at run-time. Strength reduction replaces operations that are"expensive" by "cheaper" operations, which reduces the computational effort at run-time. Constant folding evaluates and replaces complex computations, whose operands are known at compile-time, by their values, which shifts the computational effort from the run-time to the compile-time of the program.

In practice, the power of an optimization is often validated by means of benchmark tests, i.e., by measuring the performance gain on a sample of programs in order to provide empirical evidence of its effectivity. The limitations of this approach are obvious. It cannot reveal how "good" an optimization really is concerning the relevant optimization potential. In addition, it is questionable to which extent a performance improvement observed can be considered a reliable prediction in general. This would require that the sample programs are "statistically representative" because the performance gain of a specific optimization depends highly on the program under consideration.

In this monograph, we contrast this empirical approach by a mathematical approach, which focuses on *proving* the effectivity of an optimization. Central is the introduction of *formal optimality criteria*, and proof of the effectivity or even optimality of an optimization with respect to the criteria considered. Usually, these criteria exclude the existence of a certain kind of inefficiencies. Following this approach optimality gets a formal meaning. An optimization satisfying a specific optimality criterion guarantees that a program subjected to it cannot be improved any further with respect to this criterion, or hence with respect to the source of inefficiencies it addresses. Thus, rather than aiming at assuring of a specific percentage of performance improvement, our approach guarantees that a specific kind of inefficiency is proved to be absent after optimization.

Data Flow Analysis. Optimization must preserve semantics. It is thus usually preceded by a static analysis of the argument program, usually called *data flow analysis* (*DFA*), which checks the side-conditions under which an optimization is applicable. For imperative programming languages like Algol, Pascal, or Modula, an important classification of DFA techniques is derived from the treatment of programs with procedures. *Intraprocedural* DFA is characterized by a separate and independent investigation of the procedures of a program making explicit worst-case assumptions for procedure calls. *Interprocedural* DFA takes the semantics of procedure calls into account, and is thus theoretically and practically much more ambitious than intraprocedural DFA. In contrast, *local* DFA considering (maximal sequences of) straight-line code only, so-called *basic blocks*, which are investigated separately and independently, is considerably simpler, but also less powerful than intraprocedural and interprocedural DFA. In distinction to local DFA, intraprocedural and interprocedural DFA are also called *global* DFA. Figure 1.2 illustrates this classification of DFA techniques, which carries over to program optimization, i.e., local, intraprocedural, and interprocedural optimization are based on local, intraprocedural, and interprocedural DFA, respectively.

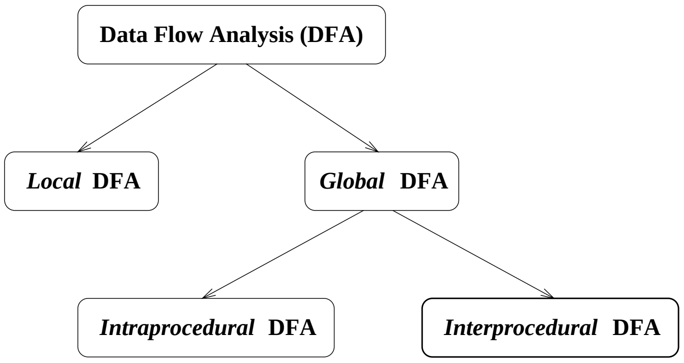

Fig. 1.2. Taxonomy of data flow analysis

DFA is usually performed on an intermediate program representation. A flexible and widely used representation is the *control flow graph* (*CFG*) of a program. This is a directed graph, whose nodes and edges represent the statements and the branching structure of the underlying program. Figure 1.3 shows an illustrative example. In order to avoid undecidability of DFA the branching structure of a CFG is usually nondeterministically interpreted. This means, whenever the control reaches a branch node, it is assumed that the program execution can be continued with any successor of the branch node within the CFG. Programs containing several procedures can naturally be represented by systems of CFGs. The control flow caused by procedure calls can be made explicit by combining them to a single graph, the *inter-*

procedural flow graph; intuitively, by connecting the call sites with the flow graphs representing the called procedures.

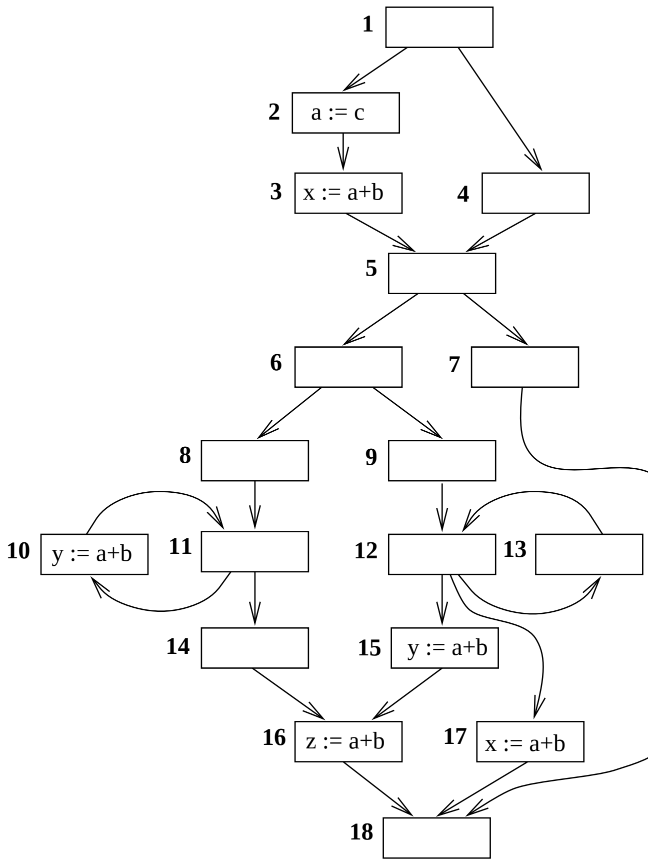

Fig. 1.3. Control flow graph

Code Motion: A Practically Relevant Optimization. Code motion is one of the most widely used program optimizations in practice, for which there are two quite natural optimization goals concerning the number of computations performed at run-time, and the lifetimes of temporaries, which are unavoidably introduced as a side-effect of the transformation. Code motion is thus well suited for demonstrating the practicality of our optimization framework because it is designed for supporting the construction of provably optimal optimizations. The code motion transformation we develop (interprocedurally with respect to a natural side-condition) satisfies both optimality criteria informally sketched above: it generates programs which are *computationally* and *lifetime optimal*. The corresponding transformation to meet these criteria is not only unique, it is even more efficient than its heuristic predecessors. In the following we illustrate the central idea underlying this transformation in the intraprocedural context.

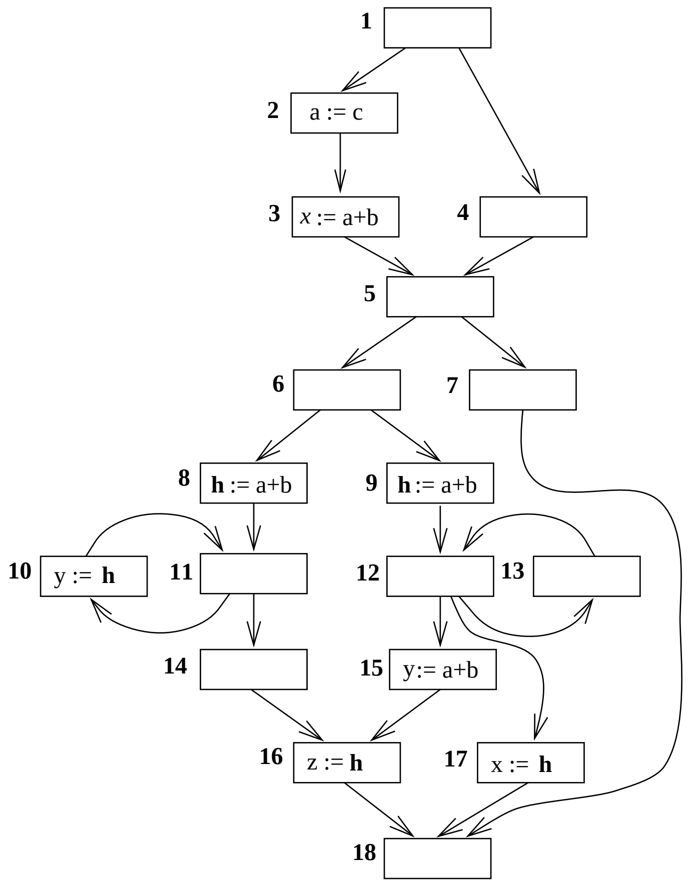

Fig. 1.4. A first code motion optimization

In essence, code motion improves the efficiency of a program by avoiding unnecessary recomputations of values at run-time. For example, in the program of Figure 1.3 the computation of $a + b$ at node **10** always yields the same value. Thus, it is unnecessarily recomputed if the loop is executed more than once at run-time. Code motion eliminates unnecessary recomputations by replacing the original computations of a program by temporaries (or registers), which are correctly initialized at appropriate program points. For example, in the program of Figure 1.3 the original computations of $a + b$ occurring at the nodes **10**, **16**, and **17** can be replaced by a temporary **h**, which is initialized by $a + b$ at the nodes **8** and **9** as illustrated in Figure 1.4.

Admissible Code Motion

Code motion must preserve the semantics of the argument program. This leads to the notion of *admissible* code motion. Intuitively, admissibility requires that the temporaries introduced for replacing the original computations of a program are correctly initialized at certain program points as illustrated above. In addition, it requires that the initializations of the temporaries do not introduce computations of new values on paths because this could introduce new run-time errors. Illustrating this by means of the program of Figure 1.3, the second requirement would be violated by initializing the temporary **h** at node **5** as shown in Figure 1.5. This introduces a computation of $a + b$ on the path $(\mathbf{1}, \mathbf{4}, \mathbf{5}, \mathbf{7}, \mathbf{18})$, which is free of a computation of $a + b$ in the original program. Under the admissibility requirement, we can obtain *computationally* and *lifetime optimal* results as indicated below.

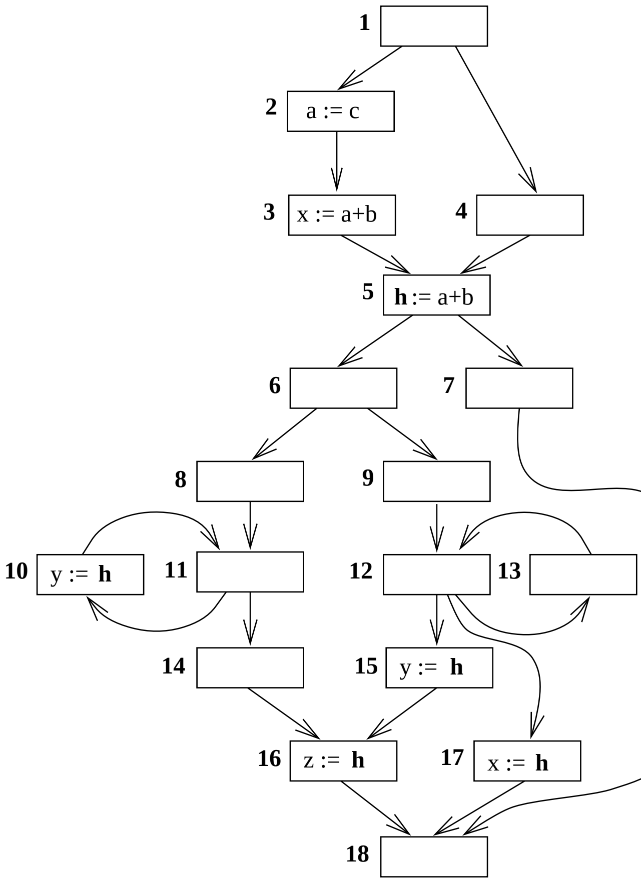

Fig. 1.5. No admissible code motion optimization

Computationally Optimal Code Motion

Intuitively, an admissible code motion is *computationally optimal*, if the number of computations on every program path cannot be reduced any further by means of admissible code motion. Achieving computationally optimal results is the primary goal of code motion. The central idea to meet this goal is to place computations

– *as early as possible*, while maintaining admissibility.

This is illustrated in Figure 1.6 showing the program, which results from the program of Figure 1.3 by means of the "as-early-as-possible" placing strategy. All unnecessary recomputations of $a + b$ are avoided by storing the value of $a+b$ in the temporary **h** and replacing all original computations of $a+b$ by **h**. Note that this program cannot be improved any further. It is computationally optimal.

Lifetime Optimal Code Motion

The "as-early-as-possible" placing strategy moves computations even if there is no run-time gain. In the running example this is particularly obvious when considering the computation of $a + b$ at node **3**, which is moved without any run-time gain. Though unnecessary code motion does not increase the number of computations on a path, it can be the source of superfluous *register pressure*, which is a major problem in practice. The secondary goal of code motion therefore is to avoid any unnecessary motions of computations while maintaining computational optimality. This is illustrated in Figure 1.7 for the running example of Figure 1.3.

Like the program of Figure 1.6, it is computationally optimal. However, computations are only moved, if it is profitable: the computations of $a + b$ at nodes **3** and **17**, which cannot be moved with run-time gain, are not touched at all. The problem of unnecessary code motions is addressed by the criterion of lifetime optimality. Intuitively, a computationally optimal code motion transformation is *lifetime optimal*, if the lifetimes of temporaries cannot be reduced any further by means of computationally optimal code motion. Intuitively, this means that in any other program resulting from a computationally optimal code motion transformation, the lifetimes of temporaries are at least as long as in the lifetime optimal one. The central idea to achieve lifetime optimality is to place computations

– *as late as possible*, while maintaining computational optimality.

The "as-late-as-possible" placing strategy transforms computationally optimal programs into a unique lifetime optimal program. This is an important difference to computational optimality. Whereas computationally optimal results can usually be achieved by several transformations, lifetime optimality is achieved by a single transformation only.

Figures 1.8 and 1.9 illustrate the lifetime ranges of the temporary **h** for the programs of Figures 1.6 and 1.7, respectively.

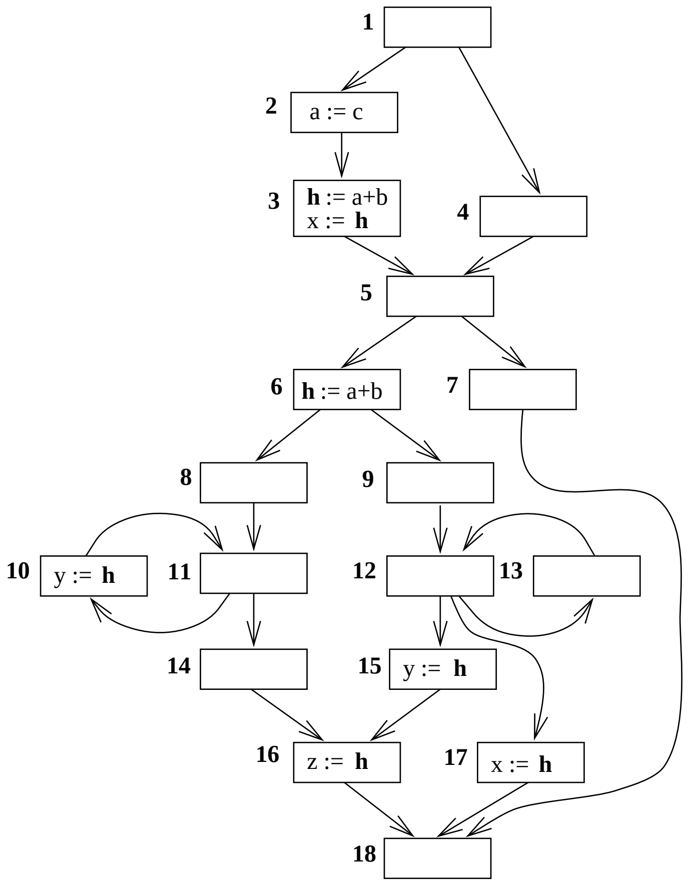

Fig. 1.6. A computationally optimal program

Summarizing, the "as-early-as-possible" code motion transformation of Figure 1.6 moves computations as far as *possible* in order to achieve computationally optimal results; the "as-late-as-possible" code motion transformation of Figure 1.7 moves computations only as far as *necessary*. Therefore, we call the first transformation the *busy* code motion transformation and the second one the *lazy* code motion transformation, or for short the *BCM*- and *LCM*-transformation.

In this monograph, we will show how to construct intraprocedural and interprocedural program optimizations like the *BCM*- and *LCM*-transformation systematically. However, we also demonstrate that usually essential differences have to be taken into account when extending intraprocedural optimizations interprocedurally. We illustrate this by developing the interprocedural counterparts of the *BCM*- and *LCM*-transformation for programs with recursive procedures, global, local, and external variables, value, reference and procedure parameters. We show that interprocedurally computationally and

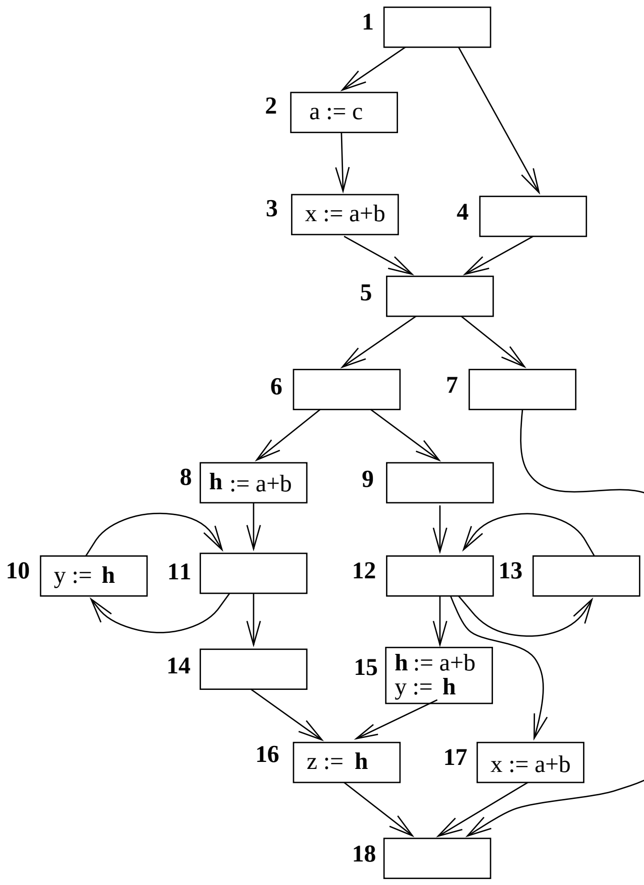

Fig. 1.7. The computationally and lifetime optimal program

lifetime optimal results are in general impossible. Therefore, we propose a natural constraint which is sufficient to meet both criteria for a large class of programs. The resulting algorithms are unique in achieving interprocedurally computationally and lifetime optimal results for this program class. Their power is illustrated by a complex example in Section 10.6. Additionally, a detailed account of the example considered in the prologue for illustrating the intraprocedural versions of busy and lazy code motion can be found in Section 3.5.

Acknowledgements

I am greatly indebted to many people, in particular, my academic teachers, colleagues, friends, and parents, who contributed in various, quite different

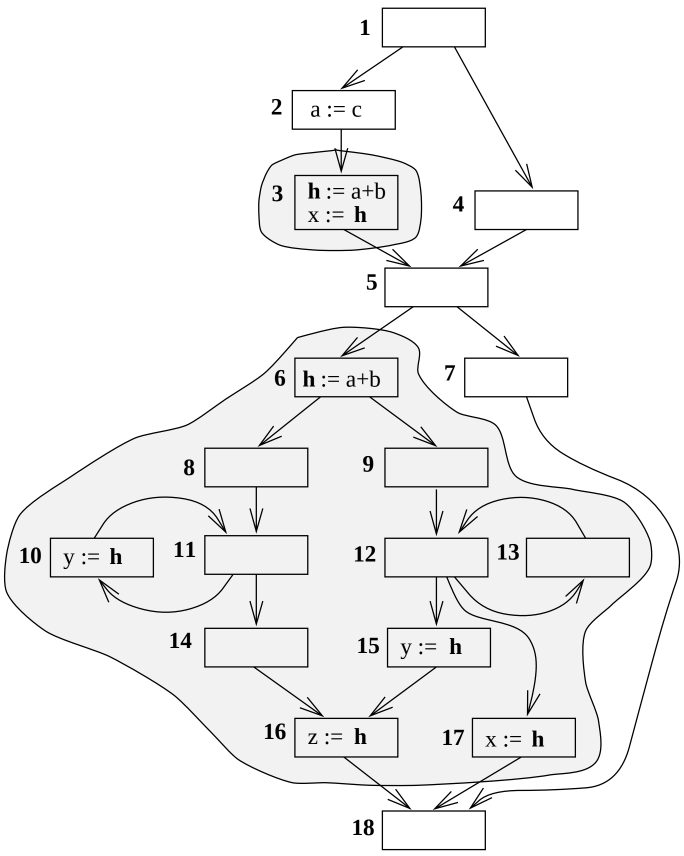

Fig. 1.8. Lifetime ranges after the *BCM*-transformation

ways to the origin of this monograph, and I wish to express my deepest gratitude to all of them.

Both I and my research, and as a consequence the present monograph, owe a lot to my academic teachers and my colleagues at Kiel and Passau University, above all to Hans Langmaack and to Bernhard Steffen. Professor Langmaack introduced me to the foundations of computer science and the specifics of compiler construction from the very beginnings of my studies, and later on I conducted my research for the doctoral dissertation underlying this monograph as a member of his research group. I am very grateful for the valuable and inspiring advice he gave, for his constant motivation and support. These thanks belong to the same extent to Professor Steffen. It was Bernhard who aroused my interest for the theory of abstract interpretation and its application to program optimization, the topic of this monograph, and also the general theme of in the meantime more than 10 years of most intensive, enjoyable, and fruitful collaboration going far beyond the joint publications we accomplished over the years. In particular, I would like to

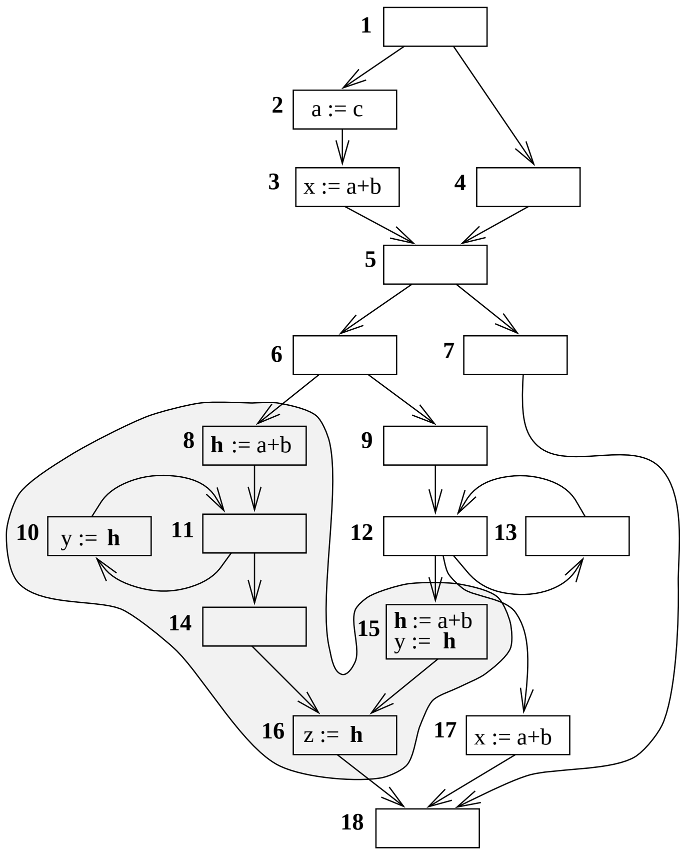

Fig. 1.9. Lifetime ranges after the *LCM*-transformation

thank Bernhard for writing the foreword to this monograph. Moreover, I am also very grateful to Oliver Rüthing, my close colleague at Kiel University, and collaborator on several joint publications in the field of program analysis and optimization. Not just because we shared an office, Oliver accompanied the development of my doctoral dissertation at very close range, and I would like to thank him cordially for many discussions on technical and nontechnical topics, and invaluable comments, which helped improving it.

I would also like to thank Dhananjay M. Dhamdhere, Rajiv Gupta, Flemming Nielson, Robert Paige, Thomas Reps, Barry K. Rosen, and F. Kenneth Zadeck for many stimulating discussions, mostly during conferences and Professor Nielson's guest professorship at Kiel University. I am also grateful to Hardi Hungar for his hints on recent references to decidability and complexity results concerning formal reachability, and to Thomas Noll and Gerald Lüttgen for their careful proof-reading of a preliminary version of this monograph. My special thanks belong to Reinhard Wilhelm for taking over the third report on the underlying thesis.

Moreover, I greatly acknowledge the financial support of the Deutsche Forschungsgemeinschaft for providing me a research fellowship at Kiel University for several years, and also its generous support for joining several conferences, in particular, the ACM SIGPLAN'92 Conference on Programming Languages Design and Implementation in San Francisco, where the intraprocedural version of the lazy code motion transformation considered as a running example in this monograph was originally presented. Without this support, conducting my research would have been much more difficult.

I am also especially grateful to Gerhard Goos, editor-in-chief of the series of Lecture Notes in Computer Science, and an anonymous referee for reviewing my doctoral dissertation for publication in this series. I greatly acknowledge their helpful comments and suggestions for improving the current presentation. Last but not least, I would cordially like to thank Alfred Hofmann at Springer-Verlag for the smooth and competent cooperation, his assistance and thoroughness in publishing this monograph, and particularly for his patience in awaiting the final version of the manuscript.

My deepest gratitude, finally, belongs to my parents for their continuous encouragement, support, and love.

Passau, May 1998 Jens Knoop