UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Optimal Code Motion: Theory and Practice[1]

Jens Knoop[+], Oliver Rüthing[*]
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preußerstraße 1-9
D–24105 Kiel

Bernhard Steffen
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau

MIP-9310
Dezember 1993

[+] Part of the work was done while the author was supported by the Deutsche Forschungsgemeinschaft grant La 426/9-2. Author's current address: Fakultät für Mathematik und Informatik, Universität Passau, Innstraße 33, D–94032 Passau.

[*] Part of the work was done while the author was supported by the Deutsche Forschungsgemeinschaft grant La 426/11-1.

[1]To appear in *ACM Transactions on Programming Languages and Systems (TOPLAS).*

Abstract

An implementation oriented algorithm for *lazy code motion* is presented that minimizes the number of computations in programs while suppressing any unnecessary code motion in order to avoid superfluous register pressure. In particular, this variant of the original algorithm for lazy code motion works on flowgraphs whose nodes are basic blocks rather than single statements, as this format is standard in optimizing compilers. The theoretical foundations of the modified algorithm are given in the first part, where t-refined flowgraphs are introduced for simplifying the treatment of flowgraphs whose nodes are basic blocks. The second part presents the 'basic block' algorithm in standard notation, and gives directions for its implementation in standard compiler environments.

Keywords

Elimination of partial redundancies, code motion, data flow analysis (bit-vector, unidirectional, bidirectional), nondeterministic flowgraphs, t-refined flow graphs, critical edges, lifetimes of registers, computational optimality, lifetime optimality.

Contents

List of Figures

List of Tables

1 INTRODUCTION

Code motion is a technique for improving the efficiency of a program by avoiding unnecessary recomputations of a value at runtime. This is achieved by replacing the original computations of a program by temporary variables (registers) that are initialized *correctly* at suitable program points.

In order to preserve the semantics of the original program, code motion must also be *safe*, i.e., it must not introduce computations of new values on paths. In fact, under this requirement it is possible to obtain *computationally optimal* results, i.e., results where the number of computations on each program path cannot be reduced anymore by means of safe code motion (cf. Theorem 3.13). The central idea to obtain computational optimality is to place computations *as early as possible* in a program, while maintaining safety (cf. [6, 8, 11, 25, 27, 31]). This strategy, however, moves computations even if it is *unnecessary*, i.e., there is no runtime gain.[1] In fact, this can cause superfluous *register pressure*, which is a major problem in practice.

In [22] we presented a *lazy* computationally optimal code motion algorithm, which suppresses any unnecessary code motion in order to avoid superfluous register pressure. The point of this algorithm is to place computations *as late as possible* in a program, while maintaining computational optimality. In fact, this placing strategy *minimizes* the lifetimes of the temporary that must be introduced for a given expression by any code motion algorithm that places this expression computationally optimally.

Note that we are separately dealing with the different syntactic terms here, which is the state of the art in code motion since Morel/Renvoise's seminal paper [27]. In contrast to the notion of computational optimality, this separation affects lifetime considerations, an observation which to our knowledge did not yet enter the reasoning on the register pressure problem in code motion papers: all these papers, which mainly proposed some heuristics, are based on the 'separation paradigm' of Morel/Renvoise-like code motion (cf. "Related Work"). Our notion of lifetime optimality characterizes the optimal solution to the register pressure problem under these circumstances. We therefore consider lifetime optimality, which can be obtained very efficiently, as an adequate approximation of the expensive (probably NP-hard) global notion of register pressure optimality.

Despite its optimality, the algorithm of [22] has a surprisingly simple structure: it is composed of four purely unidirectional analyses. This guarantees efficiency as the standard bit-vector techniques dealing with all program terms simultaneously can be applied [2, 14, 16, 17, 20, 21, 34–37].

In this paper, we emphasize the practicality of lazy code motion by giving explicit directions for its implementation in standard compiler environments. In particular, we present a version of the algorithm here which works on flowgraphs whose nodes are basic blocks rather than single statements, as this format is standard in optimizing compilers.

Our presentation is split into two parts which can be read and understood independently.

Theoretical Part: Here, the lazy code motion transformation is stepwise developed for flowgraphs whose nodes are basic blocks, and its correctness and optimality are proved. Important is the introduction of *t-refined flowgraphs*, which provide an appropriate frame-

[1] In [8] unnecessary code motions are called *redundant*.

work for concise specifications of program properties and optimality criterea in the presence of basic block nodes. The theoretician may be satisfied reading this part.

Practical Part: Here, the algorithm realizing the transformation specified in the first part is presented in a style which allows a direct and efficient implementation. In fact, the practitioner can find all relevant information for an implementation in this second part.

As in [22] the key observation is the *decomposability* of the known (bidirectional) placement algorithms into a sequence of purely unidirectional components.[2] This conceptual simplification, which can be maintained for lazy code motion, led us to the following two step structure:

1. Obtain computational optimality while maintaining safety by means of the "as early as possible" placement strategy.

2. Avoid unnecessary code motion while maintaining computational optimality by means of the "as late as possible" placement strategy.

Each step only requires two unidirectional data flow analyses. This guarantees the efficiency of the standard bit-vector techniques. However, compared with [22] the earliestness and isolation analyses are modified here in order to decouple the two analyses of each step, which now become independent and therefore parallelizable. Moreover, only minor modifications of implementations based on the algorithm of Morel and Renvoise [27] are necessary, which supports a simple integration into standard compiler environments.

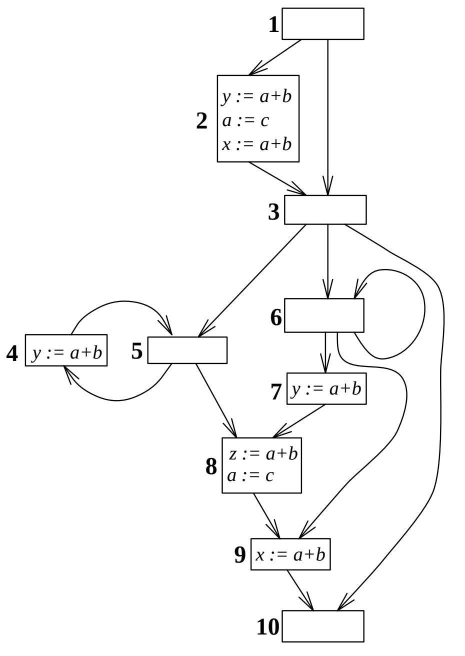

Figure 1: The Motivating Example

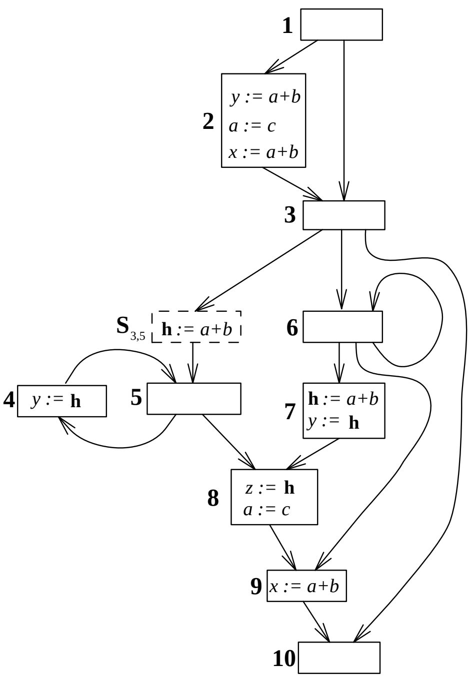

Figure 2: The Lazy Code Motion Transformation

Figure 1 displays an example, which is complex enough to illustrate the essential features of our approach. It will be discussed in more detail in Section 4.2. For now just note that

[2]See "Related Work" below for details.

our algorithm is unique in performing the optimization displayed in Figure 2, which is exceptional for the following reasons: it eliminates the partially redundant computations of "a+b" inside the nodes **4** and **8** by moving them to node **7** and to a new synthetic node which is introduced on the edge from node **3** to **5**, but it does not touch the computations of $a+b$ in node **2** and **9** that cannot be moved with runtime gain. Thus our algorithm only moves computations when it is profitable.

Related Work

In 1979 Morel and Renvoise proposed a bit-vector algorithm for the suppression of partial redundancies [27]. The bidirectionality of their algorithm became model in the field of bit-vector based code motion (cf. $[3, 5–8, 12, 18, 19, 26, 28, 30]$). Bidirectional algorithms, however, are in general conceptually and computationally more complex than unidirectional ones: e.g., in contrast to the unidirectional case, where reducible programs can be dealt with in $O(n\,log(n))$ bit-vector steps, where n characterizes the size of the argument program (e.g., number of statements), the best known estimation for bidirectional analyses is $O(n^2)$ (cf. [8–10]).[3] In [11] Dhamdhere, Rosen, and Zadeck showed that the original transformation of Morel and Renvoise can be solved as easily as a unidirectional problem. However, they do not address the problem of unnecessary code motion. This problem was first addressed in [3,6,8] and more recently in [10]. However, the first three proposals are of heuristic nature, i.e., code is unnecessarily moved or redundancies remain in the program, and the latter one is of limited applicability: it requires the reducibility of the flowgraph under consideration.

In [22] the first structurally unrestricted algorithm was presented yielding computationally optimal results while completely avoiding unnecessary code motion.[4] This result as well as the fact that it is composed of *unidirectional* analyses,[5] carry over to the version presented in this paper. Thus, estimations for the worst case time complexity of unidirectional analyses apply (for details see $[2, 14, 16, 17, 20, 21, 34–37]$). Moreover, our algorithm is *conceptually simple*. It only requires the computation of four global predicates *D-Safe*, *U-Safe*, *Delayed*, and *Isolated* in two sequential steps.

2 PRELIMINARIES

2.1 Terms and Flowgraphs

We consider *terms* inductively built of variables, constants and operators, and *directed flowgraphs* $\mathbf{G} = (\mathbf{N}, \mathbf{E}, \mathbf{s}, \mathbf{e})$ with node set $\mathbf{N}$ and edge set $\mathbf{E}$. Nodes $\mathbf{n} \in \mathbf{N}$ represent *basic blocks* (cf. [15]) consisting of a linear sequence of assignments of the form $v := t$, where v is a variable and t a term. An assignment is called a *modification* of t, if

[3]In [9] the complexity of bidirectional problems has been estimated by $O(n * w)$, where w denotes the *width* of a flow graph. In contrast to the well-known notion of *depth* (cf. [15]) traditional estimations are based on, width is not a structural property of a flowgraph, but varies with the bit-vector problem under consideration. In particular, it is larger for bidirectional problems than for unidirectional ones, and in the worst case it is linear in the size of the flowgraph.

[4]A variant of this algorithm which inserts computations on edges rather than in nodes was recently proposed in [13].

[5]Such an algorithm was first proposed in [31] which later on was interprocedurally generalized to programs with procedures, local and global variables and formal value parameters in [25]. Both algorithms realize an "as early as possible" placement.

it assigns to one of t's operands.[6] Edges $(\mathbf{m},\mathbf{n}) \in \mathbf{E}$ represent the nondeterministic branching structure of $\mathbf{G}$,[7] and $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of $\mathbf{G}$, which are assumed not to possess any predecessors and successors, respectively. Moreover, $succ(\mathbf{n}) =_{df} \{\, \mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \mathbf{E} \,\}$ and $pred(\mathbf{n}) =_{df} \{\, \mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \mathbf{E} \,\}$ denote the set of all immediate successors and predecessors of a node $\mathbf{n}$, respectively. A *finite path* of $\mathbf{G}$ is a sequence $\langle \mathbf{n}_1, \ldots, \mathbf{n}_k \rangle$ of nodes such that $\mathbf{n}_{i+1} \in succ(\mathbf{n}_i)$ for all $1 \le i < k$. ϵ denotes the empty path, and the *length* of a path p is denoted by λ_p. For a given path p and an index $1 \le i \le \lambda_p$ the i-th component of a path is addressed by p_i. A path q is said to be a subpath of p, in signs $q \sqsubseteq p$, if there is an index $1 \le i \le \lambda_p$ such that $i + \lambda_q - 1 \le \lambda_p$ and $q_j = p_{i+j-1}$ for all $1 \le j \le \lambda_q$. In particular, for any path p and indices $i,\ j \le \lambda_p$ we denote the subpath which is induced by the sequence of nodes from p_i up to p_j by $p[i,j]$.[8] Moreover, if i or j is excluded from this subpath we will write $p]i,j]$ or $p[i,j[$, respectively. The set of all finite paths of $\mathbf{G}$ leading from a node $\mathbf{n}$ to a node $\mathbf{m}$ is denoted by $\mathbf{P}[\mathbf{n},\mathbf{m}]$. Finally, every node $\mathbf{n} \in \mathbf{N}$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

2.2 Critical Edges

It is well-known that in completely arbitrary graph structures the code motion process may be blocked by *critical edges*, i.e., by edges leading from nodes with more than one successor to nodes with more than one predecessor (cf. [6, 8, 12, 29, 32, 33]).

In Figure 3(a) the computation of "$a + b$" at node **3** is partially redundant with respect to the computation of "$a + b$" at node **1**. However, this partial redundancy cannot safely be eliminated by moving the computation of "$a+b$" to its preceding nodes, because this may introduce a new computation on a path leaving node **2** on the right branch. On the other hand, it can safely be eliminated after inserting a synthetic node $\mathbf{S}_{2,3}$ in the critical edge $(\mathbf{2},\mathbf{3})$, as illustrated in Figure 3(b).

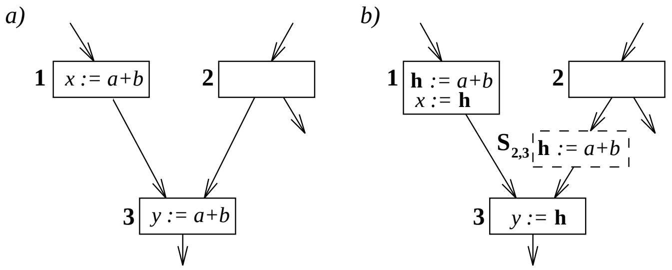

Figure 3: Critical Edges

We will therefore restrict our attention to programs, where every critical edge has been

[6] As usual every modification is assumed to change the value of t.

[7] We do not assume any structural restrictions on $\mathbf{G}$. In fact, every algorithm computing the fixed point solution of a unidirectional bit-vector data flow analysis problem may be used to compute the predicates *U-Safe*, *D-Safe*, *Latest*, and *Isolated* (cf. [15]). However, application of the efficient techniques of [2, 14, 16, 17, 20, 21, 34–37] requires that $\mathbf{G}$ satisfies certain structural restrictions.

[8] If $i > j$ the subpath $p[i,j]$ means ϵ.

split by inserting a synthetic basic block.[9] After this simple transformation every flowgraph $\mathbf{G}$ satisfies the following structural property:

LEMMA 2.1 (CONTROL FLOW LEMMA)

1. $\forall\, \mathbf{n} \in \mathbf{N}.\ |\, pred(\mathbf{n})\,| \geq 2 \Rightarrow succ(pred(\mathbf{n})) = \{\mathbf{n}\}$
2. $\forall\, \mathbf{n} \in \mathbf{N}.\ |\, succ(\mathbf{n})\,| \geq 2 \Rightarrow pred(succ(\mathbf{n})) = \{\mathbf{n}\}$

For the proof of the first property consider a predecessor $\mathbf{m}$ of a node $\mathbf{n}$ with $|\, pred(\mathbf{n})\,| \geq 2$. Now, if $\mathbf{m}$ is a synthetic basic block then $\mathbf{n}$ is the unique successor of $\mathbf{m}$ by construction. Thus we can assume that $\mathbf{m}$ is not synthetic. This implies that the edge $(\mathbf{m}, \mathbf{n})$ is uncritical and hence $\mathbf{m}$ cannot have a successor different from $\mathbf{n}$. The proof of the second property is similar.

2.3 t-refined Flowgraphs

Given a computation t a basic block $\mathbf{n}$ can be divided into two disjoint parts:

- an *entry part* consisting of all statements up to and including the last modification of t,
- an *exit part* consisting of all remaining statements.

Note that a nonempty basic block always has a nonempty entry part, whereas its exit part may be empty (for illustration see Figure 4).

For a fixed term t, the *t-refined flowgraph* $G_t = (N_t, E_t, s_t, e_t)$ is obtained straightforwardly by splitting the nodes of a usual flowgraph into their entry and exit parts. Obviously, s_t and e_t are the entry part of $\mathbf{s}$ and the exit part of $\mathbf{e}$, respectively.

t-refined flowgraphs are not relevant for the purpose of implementation. Rather they provide a convenient framework for the theoretical foundations of code motion. In particular, they simplify the specification and verification of correctness and optimality.

In order to distinguish nodes of the flowgraph $\mathbf{G}$, i.e., basic blocks, and nodes of the flowgraph G_t, i.e., basic block parts, we use bold face notation like $\mathbf{n} \in \mathbf{N}$ for the former and italic notation like $n \in N_t$ for the latter. Finally, to support easy switching between both representations we introduce the following notational convention: for a given basic block $\mathbf{n} \in \mathbf{N}$ the corresponding entry and exit parts in N_t are denoted by n_N^t and n_X^t, respectively, and conversely $n \in N_t$ refers to $\mathbf{n} \in \mathbf{N}$.

2.4 Local Redundancies

In the sequel, we will concentrate on the *global* effects of code motion. Therefore, it is assumed that all *local* redundancies are already eliminated by means of some standard techniques for *common subexpression elimination* on basic blocks [1, 4]. This guarantees the following two properties.

[9] This differs from [22], where every edge leading to a node with multiple predecessors has to be split.

1. Every entry part contains at most one computation of t before the first modification, which then is denoted as the *entry computation.*[10] Further computations of t are encapsulated by two modifications, and are therefore irrelevant for global redundancy elimination (cp. Figure 4(a)). The entry computation determines the *insertion point* of the corresponding entry part: it is the point immediately before the entry computation if existent. Otherwise it is immediately before the first modification if existent, or it is at the end of the entry part if there is neither an entry computation nor a modification (cp. Figure 4(c)). This guarantees that the lifetimes of registers are kept locally minimal.

2. Exit parts contain at most one computation of t, which then is denoted as the *exit computation.* The insertion point of an exit part is immediately before the exit computation if existent, or at the end of the exit part under consideration otherwise.

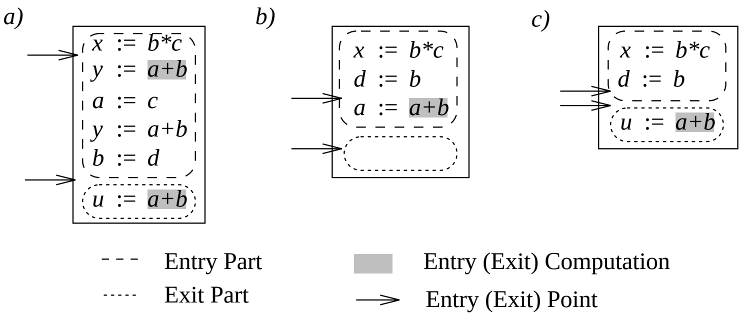

Figure 4: Entry and Exit Parts of a Basic Block wrt "$a + b$"

2.5 Conventions

Following [27], we assume that all right-hand-side terms of assignment statements contain at most one operation symbol. This does not impose any restrictions, because assignment statements can be canonically decomposed into sequences of assignments of this form according to the inductive structure of terms.[11] As a consequence of this assumption it is enough to develop our algorithm for an arbitrary but fixed term here, because a global algorithm dealing with all program terms is just the independent combination of all the "term algorithms" in this case. This allows us to apply the efficient bit-vector algorithms dealing with all terms simultaneously (cf. [2,14,16,17,20,21,34–37]) as well as to interleave code motion with copy propagation using the slotwise approach of [11].

In the following, we therefore develop our algorithm for an arbitrary but fixed flowgraph **G** and a given term t, which avoids a highly parameterized notation. Moreover, we denote the entry and exit computations as *code motion candidates.*

[10] Note, that e.g. in "$a := a + b$" the computation of "$a + b$" is prior to its modification.

[11] Note that rearranging the term structure exploiting e.g., associativity and commutativity of some operators may offer new opportunities for code motion.

3 THEORY

In this section the lazy code motion transformation is stepwise developed for flowgraphs whose nodes are basic blocks, and its correctness and optimality are proved. Important is the introduction of *t-refined flowgraphs*, which provide an appropriate framework for the concise formalization of the relevant aspects of code motion, e.g., admissibility requirements and optimality criterea.

The section is organized in three subsections. Section 3.1 presents the basic concepts of code motion. Central are the notions of *safety* and *correctness*, because they guarantee that code motion preserves the semantics of the argument program. *Computational* and *lifetime optimality*, which are discussed afterwards, characterize the quality of code motion. Sections 3.2 and 3.3 specify transformations for computationally and lifetime optimal code motion, respectively. As we will show, there is a variety of computationally optimal code motions, among which there is a unique lifetime optimal representative. The known code motion algorithms focus on computational optimality, and treat the lifetime aspect at most heuristically. In fact, lifetime optimality was first achieved in [22].

3.1 CODE MOTION

A code motion transformation CM is completely characterized by two predicates on nodes in N.

- $Insert_{CM}$, determining in which nodes computations have to be inserted and
- $Repl_{CM}$, specifying the nodes where code motion candidates have to be replaced.

It is assumed that $Repl_{CM}$ implies $Comp$. Moreover, in order to avoid the introduction of new local redundancies $Repl_{CM}$ must be implied by the conjunction of $Insert_{CM}$ and $Comp$. Obviously this does not impose any restriction on our approach. We only avoid transformations that keep a code motion candidate even after an insertion into the corresponding node.

The transformation itself comprises the three steps summarized in Table 1. We will denote the set of code motion transformations with respect to t by $\mathcal{CM}$.

1. Introduce a new temporary variable $\mathbf{h}_{CM}$ for t.
2. Insert assignments $\mathbf{h}_{CM} := t$ at the insertion points of all $n \in N$ satisfying $Insert_{CM}$.
3. Replace the (unique) code motion candidate of t by $\mathbf{h}_{CM}$ in every $n \in N$ satisfying $Repl_{CM}$.

Table 1: General Pattern of a Code Motion Transformation

For each $n \in N$ we define two local predicates:

- $Comp(n)$: n contains a code motion candidate of t.

- $\mathit{Transp}(n)$: n is transparent, i.e., it does not contain a modification of t.

These local predicates are the basis for the definition of all global predicates introduced in this paper, which always refer to the insertion point of the part under consideration. In order to address program points that have a computation of t after applying a code motion transformation CM, we use the following indexed version of the predicate Comp:

$$\mathit{Comp}_{\mathit{CM}}(n) =_{\mathit{df}} \mathit{Insert}_{\mathit{CM}}(n) \vee \mathit{Comp}(n) \wedge \neg \mathit{Repl}_{\mathit{CM}}(n)$$

Logical formulas are built according to the following priority ϕ of operations and quantifiers: $\phi(\neg) > \phi(\wedge) > \phi(\vee) > \phi(\Rightarrow) = \phi(\Leftrightarrow) > \phi(\forall) = \phi(\exists)$. Finally, any local or global predicate *Predicate* defined for nodes $n \in N$ is extended to path arguments in two ways:[12]

- $\mathit{Predicate}^{\forall}(p) \;\Leftrightarrow\; \forall\, 1 \leq i \leq \lambda_p.\ \mathit{Predicate}(p_i)$
- $\mathit{Predicate}^{\exists}(p) \;\Leftrightarrow\; \exists\, 1 \leq i \leq \lambda_p.\ \mathit{Predicate}(p_i)$

3.1.1 Safety and Correctness

In order to guarantee that the semantics of the argument program is preserved, we require that code motion transformations are *admissible*. Intuitively, this means that every insertion of a computation is *safe*, i.e., on no program path the computation of a new value is introduced at initialization sites, and that every substitution of a code motion candidate by $\mathbf{h}_{\mathit{CM}}$ is *correct*, i.e., $\mathbf{h}_{\mathit{CM}}$ always represents the same value as t at use sites. This requires that $\mathbf{h}_{\mathit{CM}}$ is initialized on every program path leading to some use site in a way such that no modification occurs afterwards.

This gives rise to the following definition, which states when it is safe to insert a computation of t at the insertion point of $n \in N$, or when it is correct to replace a code motion candidate of t by $\mathbf{h}_{\mathit{CM}}$ in n, respectively.

Definition 3.1 (Safety & Correctness)
Let $n \in N$. We define:

1. $\mathit{Safe}(n) \;\Leftrightarrow_{\mathit{df}}\; \forall\, p \in \mathbf{P}[s,e]\ \forall\, i \leq \lambda_p.\ p_i = n \Rightarrow$
   $$\underbrace{\exists\, j < i.\ \mathit{Comp}(p_j) \wedge \mathit{Transp}^{\forall}(\,p[j,i[\,)}_{i)} \;\vee\; \underbrace{\exists\, j \geq i.\ \mathit{Comp}(p_j) \wedge \mathit{Transp}^{\forall}(\,p[i,j[\,)}_{ii)}$$
2. *Let $\mathit{CM} \in \mathcal{CM}$. Then:*
   $\mathit{Correct}_{\mathit{CM}}(n) \;\Leftrightarrow_{\mathit{df}}\; \forall\, p \in \mathbf{P}[s,n]\ \exists\, i \leq \lambda_p.\ \mathit{Insert}_{\mathit{CM}}(p_i) \wedge \mathit{Transp}^{\forall}(\,p[i,\lambda_p[\,)$

Restricting Definition 3.1(1) to either condition (i) or (ii) leads to the notion of *up-safe* and *down-safe* basic block parts, respectively.

[12] Note that $\neg\mathit{Predicate}^{\forall}(p)$ and $\neg\mathit{Predicate}^{\exists}(p)$ are now abbreviations for $\exists\, 1 \leq i \leq \lambda_p.\ \neg\mathit{Predicate}(p_i)$ and $\forall\, 1 \leq i \leq \lambda_p.\ \neg\mathit{Predicate}(p_i)$, respectively.

Definition 3.2 (Up-Safety & Down-Safety)

1. $\forall\, n \in N.\ \textit{U-Safe}(n) \ \Leftrightarrow_{df}\ \forall\, p \in \mathbf{P}[s,n]\ \exists\, i < \lambda_p.\ \textit{Comp}\,(p_i) \wedge \textit{Transp}^{\forall}(\,p[i,\lambda_p[\,)$
2. $\forall\, n \in N.\ \textit{D-Safe}(n) \ \Leftrightarrow_{df}\ \forall\, p \in \mathbf{P}[n,e]\ \exists\, i \leq \lambda_p.\ \textit{Comp}\,(p_i) \wedge \textit{Transp}^{\forall}(\,p[1,i[\,)$

We have:

Lemma 3.3 (Safety Lemma)

$$\forall\, n \in N.\ \textit{Safe}\,(n) \Leftrightarrow \textit{D-Safe}(n) \vee \textit{U-Safe}(n).$$

Whereas the backward direction is obvious, the converse implication is essentially a consequence of the fact that we are only considering nondeterministic branching: every path leading to a node can be completed by every path starting at that node. In particular, every path violating the up-safety condition at a node n, can be linked to every path violating the down-safety condition at n. This yields a path violating the safety condition at n, and proves the contrapositive of the forward implication.

Now we can define the class of *admissible* code motion transformations, which are guaranteed to preserve the semantics of the argument program.

Definition 3.4 (Admissible Code Motion)
A code motion transformation $CM \in \mathcal{CM}$ is admissible *if and only if for every $n \in N$ the following two conditions are satisfied:*

- $\textit{Insert}_{CM}(n) \Rightarrow \textit{Safe}\,(n)$
- $\textit{Repl}_{CM}(n) \Rightarrow \textit{Correct}_{CM}(n)$

The set of all admissible code motion transformations is denoted by $\mathcal{CM}_{Adm}$.

We have:

Lemma 3.5 (Correctness Lemma)

$$\forall\, CM \in \mathcal{CM}_{Adm}\ \forall\, n \in N.\ \textit{Correct}_{CM}(n) \Rightarrow \textit{Safe}\,(n)$$

The proof can be found in Section A.1.

3.1.2 Computational Optimality

The primary goal of code motion is to minimize the number of computations on every program path. This intent is reflected by the following relation. A code motion transformation $CM \in \mathcal{CM}_{Adm}$ is *computationally better*[13] than a code motion transformation $CM' \in \mathcal{CM}_{Adm}$ if and only if

$$\forall\ p \in \mathbf{P}[s,e].\ \mid \{i \mid \textit{Comp}_{CM}(p_i)\} \mid\ \leq\ \mid \{i \mid \textit{Comp}_{CM'}(p_i)\} \mid$$

Definition 3.6 (Computationally Optimal Code Motion)
An admissible code motion transformation $CM \in \mathcal{CM}_{Adm}$ is computationally optimal *if and only if it is computationally better than any other admissible code motion transformation. We denote the set of computationally optimal code motion transformations by $\mathcal{CM}_{CmpOpt}$.*

[13]Note that this relation is reflexive. In fact, *computationally at least as good* would be the more precise but uglier term.

3.1.3 Lifetime Optimality

The secondary goal is to avoid unnecessary code motion, which can cause superfluous register pressure. This requires to minimize the lifetimes of temporary variables that are introduced by a computationally optimal code motion transformation. Locally, i.e., within a basic block part, this is guaranteed when using the insertion points (cf. Section 2.4) for initialization. Globally, lifetime optimality means minimal lifetime ranges for temporary variables in terms of paths through the flowgraph. Formally, it is defined relative to the following (reflexive) relation: $CM \in \mathcal{CM}$ is *lifetime better* than a code motion transformation $CM' \in \mathcal{CM}$ if and only if

$$\forall\, p \in LtRg\,(CM)\ \exists\, q \in LtRg\,(CM').\ p \sqsubseteq q$$

where $LtRg\,(CM)$ denotes the set of *lifetime ranges* of CM, which is defined by:

$$LtRg\,(CM) =_{df} \{p \mid Insert_{CM}(p_1)\ \wedge Repl_{CM}(p_{\lambda_p}) \wedge \neg Insert^{\exists}_{CM}(\,p]1, \lambda_p]\,)\}$$

Definition 3.7 (Lifetime Optimal Code Motion)
A computationally optimal code motion transformation $CM \in \mathcal{CM}_{CmpOpt}$ *is* lifetime optimal *if and only if it is lifetime better than any other computationally optimal code motion transformation. The set of lifetime optimal code motion transformations is denoted by* $\mathcal{CM}_{LtOpt}$.

Intuitively, lifetime optimality guarantees that no computation has unnecessarily been moved, i.e., without runtime gain. Hence, any superfluous register pressure due to unnecessary code motion is avoided. Clearly, a simultaneous investigation of all variables of a program may allow to reduce the register pressure in a program even further. However, to our knowledge, this problem has not yet been investigated (cf. Section 3.3.1).

In contrast to computational optimality, which in fact can be achieved by different code motion transformations, there exists at most one lifetime optimal code motion transformation.

Theorem 3.8 (Uniqueness of Lifetime Optimal Code Motion)

$$|\,\mathcal{CM}_{LtOpt}\,| \leq 1$$

Proof. Let $CM,\ CM' \in \mathcal{CM}_{LtOpt}$. To prove that $CM = CM'$ we must show:

$$\forall\, n \in N.\ Insert_{CM}(n) \ \Leftrightarrow\ Insert_{CM'}(n) \tag{I}$$

$$\forall\, n \in N.\ Repl_{CM}(n) \ \Leftrightarrow\ Repl_{CM'}(n) \tag{II}$$

For symmetry reasons it is sufficient to prove only one direction of the logical equivalences. Starting with (I), the computational optimality of CM delivers the following proposition which is immediate from the fact that any other kind of initialization could be suppressed without affecting the admissibility of CM:

$$Insert_{CM}(n) \Rightarrow \exists\, m \in N\ \exists\, p \in \mathbf{P}[n,m].\ p \in LtRg\,(CM)$$

Due to the lifetime optimality of CM there is a path $q \in LtRg(CM')$ such that $p \sqsubseteq q$. Now, assuming $q_1 \neq n$ the lifetime optimality of CM' would yield a lifetime range $p' \in LtRg(CM)$ with $q \sqsubseteq p'$, which according to the definition of lifetime ranges would imply $\neg Insert_{CM}(n)$ in contradiction to the premise of the implication under consideration. Thus, $n = q_1$, which directly yields the validity of $Insert_{CM'}(n)$.

(II) is a consequence of the following chain of implications:

$$\begin{array}{rcl} & & Repl_{CM}(n) \\ (CM \in \mathcal{CM}_{Adm}) & \Rightarrow & Comp(n) \wedge Correct_{CM}(n) \\ \text{(I)} & \Rightarrow & Comp(n) \wedge Correct_{CM'}(n) \\ (CM' \in \mathcal{CM}_{CmpOpt}) & \Rightarrow & Repl_{CM'}(n) \end{array}$$

□

Finally, we introduce the set of *first-use-lifetime ranges* of a code motion transformation CM, which are particularly useful when arguing about computation points of different code motion transformations (cp. Lemma 3.12):

$$FU\text{-}LtRg(CM) =_{df} \{p \in LtRg(CM) \mid \forall\, q \in LtRg(CM).\ (q \sqsubseteq p) \Rightarrow (q = p)\}$$

Obviously, we have:

Lemma 3.9 (First-Use-Lifetime Range Lemma)
Let $CM \in \mathcal{CM}$, $p \in \mathbf{P}[s,e]$, i_1, i_2, j_1, j_2 indices such that $p[i_1, j_1] \in FU\text{-}LtRg(CM)$, and $p[i_2, j_2] \in FU\text{-}LtRg(CM)$. Then

- *either $p[i_1, j_1]$ and $p[i_2, j_2]$ coincide, i.e., $i_1 = i_2$ and $j_1 = j_2$, or*
- *$p[i_1, j_1]$ and $p[i_2, j_2]$ are disjoint, i.e., $j_1 < i_2$ or $j_2 < i_1$.*

A formal proof is given in Section A.2.

3.2 BUSY CODE MOTION

In this section we present the busy code motion transformation which, as in [22], realizes the "as early as possible strategy" by means of a backward and a forward analysis.

However, the formulation given here characterizes *earliest* program points in terms of up-safety and down-safety.[14] This reduces the computation of earliestness to two well-known data flow analyses that are likely to be already implemented in many commercial compiler systems, and therefore simplifies the implementation of our algorithm. Moreover, this new characterization makes our approaches for busy and lazy code motion conceptually symmetric: whereas earliest placement points can be seen as the entry points into safe program areas, latest placement points are the exit points of program areas where a computationally optimal placement is possible.

We will now proceed by first characterizing earliestness in terms of up-safety and down-safety, and subsequently presenting the busy code motion transformation together with its main properties.

[14] A similar modification was also proposed by K.-H. Drechsler and M. P. Stadel in [13].

3.2.1 Earliestness

The busy code motion transformation computes t at those program points that are safe, and where an "earlier" computation of t would not be safe. Thus, *earliest* program points are defined as follows:

DEFINITION 3.10 (EARLIESTNESS)

$$\forall\, n \in N.\ \mathit{Earliest}\,(n) =_{df} \mathit{Safe}\,(n) \wedge \begin{cases} \mathit{true} & \text{if } n = s \\ \bigvee_{m \in pred(n)} \neg\,\mathit{Transp}\,(m) \vee \neg \mathit{Safe}\,(m) & \text{otherwise} \end{cases}$$

Lemma 3.11 summarizes some basic properties of earliestness. Whereas the first part is essential for proving the correctness of the busy code motion transformation, the remaining two parts give alternative characterizations of earliest computation points.

The characterization in the second part states that the safety condition and the disjunction over properties of the predecessors can be strengthened to down-safety and a conjunction, respectively. Intuitively, the first strengthening is clear, because it does not make sense to initialize after usage, and the second strengthening is essentially a consequence of the Control Flow Lemma 2.1.

The characterization in the third part directly reflects the intuition of earliestness, i.e., no admissible code motion transformation can initialize strictly earlier than in program points that are earliest (in the sense of Definition 3.10).

LEMMA 3.11 (EARLIESTNESS LEMMA)
Let $n \in N$. Then we have:

1. $\mathit{Safe}\,(n) \Rightarrow \forall\, p \in \mathbf{P}[s,n]\ \exists\, i \leq \lambda_p.\ \mathit{Earliest}\,(p_i) \wedge \mathit{Transp}^{\forall}(\,p[i,\lambda_p[\,)$
2. $\mathit{Earliest}\,(n) \ \Leftrightarrow\ \textit{D-Safe}(n) \wedge \bigwedge_{m \in pred(n)} (\neg\,\mathit{Transp}\,(m) \vee \neg\,\mathit{Safe}\,(m))$
3. $\mathit{Earliest}\,(n) \ \Leftrightarrow\ \mathit{Safe}\,(n) \wedge\ \forall\, \mathit{CM} \in \mathcal{CM}_{Adm}.\ \mathit{Correct}_{CM}(n) \Rightarrow \mathit{Insert}_{CM}(n)$

The proof can be found in Section A.3.

3.2.2 The Busy Code Motion Transformation

The busy code motion transformation (*BCM*-transformation) is specified in Table 2.

• $\mathit{Insert}_{BCM}(n) =_{df} \mathit{Earliest}\,(n)$
• $\mathit{Repl}_{BCM}(n) =_{df} \mathit{Comp}\,(n)$

Table 2: The Busy Code Motion Transformation

The intuition behind this transformation is to move up computations as far as possible while maintaining safety. This suggests that first-use-lifetime ranges of the *BCM*-transformation are suitable for characterizing the computation points of any admissible code motion transformation. The following lemma provides such characterizations.[15]

LEMMA 3.12 (BUSY CODE MOTION LEMMA)
Let $p \in \mathbf{P}[s,e]$. Then we have:

1. $\forall\, i \leq \lambda_p.\ Insert_{BCM}(p_i) \ \Leftrightarrow\ \exists\, j \geq i.\ p[i,j] \in \textit{FU-LtRg}\,(BCM)$
2. $\forall\, CM \in \mathcal{CM}_{Adm}\ \forall\, i,\, j \leq \lambda_p.\ p[i,j] \in \textit{LtRg}\,(BCM) \Rightarrow Comp^{\exists}_{CM}(\,p[i,j]\,)$
3. $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, i \leq \lambda_p.\ Comp_{CM}(p_i) \Rightarrow \exists\, j \leq i \leq l.\ p[j,l] \in \textit{FU-LtRg}\,(BCM)$

Details of the proof are given in Section A.4. We are now able to prove the main result about the *BCM*-transformation.

THEOREM 3.13 (BUSY CODE MOTION THEOREM)
The BCM-transformation is computationally optimal, i.e., $BCM \in \mathcal{CM}_{CmpOpt}$.

PROOF. The proof proceeds in two steps showing 1) $BCM \in \mathcal{CM}_{Adm}$ and 2) $BCM \in \mathcal{CM}_{CmpOpt}$.

The definition of earliestness and Lemma 3.11(1) yield that $Insert_{BCM}$ implies *Safe* and $Repl_{BCM}$ implies $Correct_{BCM}$, respectively, which proves 1). For 2) let $CM \in \mathcal{CM}_{Adm}$ and $p \in \mathbf{P}[s,e]$. Then we have as desired:

$$\begin{array}{rcl} & & |\,\{i \,|\, Comp_{BCM}(p_i)\}\,| \\ (\text{Def. } BCM) & = & |\,\{i \,|\, Insert_{BCM}(p_i)\}\,| \\ (\text{Lemma 3.9 \& 3.12(1)}) & = & |\,\{i \,|\, \exists\, j.\ p[i,j] \in \textit{FU-LtRg}\,(BCM)\}\,| \\ (\text{Lemma 3.12(2)}) & \leq & |\,\{i \,|\, Comp_{CM}(p_i)\}\,| \end{array}$$

□

3.3 LAZY CODE MOTION

Busy code motion optimally minimizes the number of computations, however, lifetimes of temporary variables are not taken into account. In fact, as a consequence of Lemma 3.12(3) we obtain that the lifetimes introduced by the *BCM*-transformation are even maximal.

LEMMA 3.14 $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, p \in \textit{LtRg}\,(CM)\ \exists\, q \in \textit{LtRg}\,(BCM).\ p \sqsubseteq q.$

The formal proof is given in Section A.5.

In this section, we modify the *BCM*-transformation along the lines of [22] in order to minimize the lifetimes of temporary variables. The idea behind this modification is to successively move the insertions of the *BCM*-transformation as far as possible in the

[15] In [22], where first-use-lifetime ranges of the *BCM*-transformation are called safe-earliest first-use paths (SEFU-paths), this lemma is called SEFU-Lemma.

direction of control flow while maintaining computational optimality. It will be shown that this leads to a lifetime optimal code motion transformation, which according to Theorem 3.8 is even unique.

The section is organized in three parts. In the first part the predicate *Latest* is introduced, which exactly characterizes how far the insertions of the *BCM*-transformation can be moved in the direction of control flow without violating computational optimality. Subsequently, the corresponding code motion transformation is presented, which is lifetime optimal, except for trivial uses of registers, i.e., registers may be used just to transfer a value to the next statement. The lazy code motion transformation, which is presented in the third part suppresses also these trivial uses. This suffices to prove our main result: the lifetime optimality of the lazy code motion transformation.

3.3.1 Latestness

Intuitively, the initializations of the *BCM*-transformation can be delayed on every program path reaching e as long as computational optimality is preserved. This leads to the notion of delayability:

Definition 3.15 (Delayability)
$\forall\, n \in N.\ \mathit{Delayed}(n) \ \Leftrightarrow_{df} \forall\, p \in \mathbf{P}[s,n]\ \exists\, i \leq \lambda_p.\ \mathit{Earliest}(p_i) \wedge \neg \mathit{Comp}^{\exists}(\, p[i,\lambda_p[\,)$

The following properties of delayability are important for the proof of Theorem 3.18, which characterizes the "maximally delayed" code motion transformation.

Lemma 3.16 (Delayability Lemma)

1. $\forall\, n \in N.\ \mathit{Delayed}(n) \ \Rightarrow\ \mathit{D\text{-}Safe}(n)$
2. $\forall\, p \in \mathbf{P}[s,e]\ \forall i \leq \lambda_p.\ \mathit{Delayed}(p_i) \Rightarrow \exists\ j \leq i \leq l.\ \ p[j,l]\ \in \mathit{FU\text{-}LtRg}(\mathit{BCM})$
3. $\forall\, \mathit{CM} \in \mathcal{CM}_{\mathit{CmpOpt}}\ \forall\, n \in N.\ \mathit{Comp}_{\mathit{CM}}(n) \Rightarrow \mathit{Delayed}(n)$

It is worth noting that Lemma 3.16(3) for the first time exactly characterizes the program points, where computationally optimal placements may insert computations. In fact, the set of computation points of a computationally optimal placement is a subset of the set of delayed nodes. This observation supports the construction of computationally optimal code motion transformations that satisfy additional optimality criteria, e.g., code is not unnecessarily moved, the number of inserted computations is minimal, i.e., the static code length becomes minimal, or the overall register occupancy (cf. Section 3.1.3) is minimized.

In the following this is demonstrated for the goal of avoiding any unnecessary code motion. Intuitively, this means that computations are only moved if this yields some runtime gain, which requires to minimize the individual lifetimes of temporaries introduced during the code motion process. Here the predicate *Latest*, which characterizes the maximally delayed program parts, is central:

$$\forall\, n \in N.\ \mathit{Latest}(n) =_{df} \mathit{Delayed}(n) \wedge (\mathit{Comp}(n) \vee \bigvee_{m \in succ(n)} \neg \mathit{Delayed}(m))$$

The proof of the following properties of latest program points can be found in Section A.7.

Lemma 3.17 (Latestness Lemma)

1. $\forall\, p \in LtRg(BCM)\ \exists\, i \le \lambda_p.\ Latest(p_i)$
2. $\forall\, p \in LtRg(BCM)\ \forall\, i \le \lambda_p.\ Latest(p_i) \Rightarrow \neg Delayed^{\exists}(\,p]i, \lambda_p]\,)$

3.3.2 The Almost Lazy Code Motion Transformation

Like earliestness also latestness induces a code motion transformation, called the *almost lazy code motion transformation* (*ALCM*-transformation), which replaces all code motion candidates (see Table 3).

• $Insert_{ALCM}(n) =_{df} Latest(n)$
• $Repl_{ALCM}(n) =_{df} Comp(n)$

Table 3: The Almost Lazy Code Motion Transformation

The *ALCM*-transformation moves the initializations of the *BCM*-transformation in the direction of control flow as far as possible without losing computational optimality. As we will see, this is almost sufficient to yield lifetime optimality except for the presence of unnecessary trivial lifetime ranges. In fact, the following, slightly weaker notion of lifetime optimality holds.

$CM \in \mathcal{CM}_{CmpOpt}$ is called *almost lifetime optimal*, if

$$\forall\, p \in LtRg(CM).\ \lambda_p \ge 2 \Rightarrow \forall\, CM' \in \mathcal{CM}_{CmpOpt}\ \exists\, q \in LtRg(CM').\ p \sqsubseteq q$$

Note that this definition coincides with the definition of lifetime optimality except for the treatment of lifetime ranges of length 1, which are not considered here at all. Thus, lifetime optimality implies 'almost lifetime optimality'. Denoting the set of almost lifetime optimal code motion transformations by $\mathcal{CM}_{ALtOpt}$, we obtain:

Theorem 3.18 (Almost Lazy Code Motion Theorem)
The ALCM-transformation is almost lifetime optimal, i.e., $ALCM \in \mathcal{CM}_{ALtOpt}$.

Proof. We proceed in three steps: 1) $ALCM \in \mathcal{CM}_{Adm}$, 2) $ALCM \in \mathcal{CM}_{CmpOpt}$, and 3) $ALCM \in \mathcal{CM}_{ALtOpt}$.

Ad 1) Obviously, due to the definition of latestness and Lemma 3.16(1) every $n \in N$ with $Insert_{ALCM}(n)$ is safe. Thus it remains to show:

$$\forall\, n \in N.\ Repl_{ALCM}(n) \Rightarrow Correct_{ALCM}(n)$$

Clearly, $Repl_{ALCM}$ implies $Repl_{BCM}$, which because of the admissibility of the *BCM*-transformation implies $Correct_{BCM}$. Hence, applying Lemma 3.17(1) we also obtain $Correct_{ALCM}(n)$ as desired.

Ad 2) For any $p \in \mathbf{P}[s,e]$ the following sufficient sequence of inequations holds:

$$\begin{array}{rcl} & & |\{i \mid \mathit{Comp}_{ALCM}(p_i)\}| \\ (\text{Def. } ALCM) & = & |\{i \mid \mathit{Insert}_{ALCM}(p_i)\}| \\ (*) & \leq & |\{(i,j) \mid \exists\, p[i,j] \in \mathit{FU\text{-}LtRg}(BCM)\}| \\ (\text{Def. } \mathit{LtRg}(BCM), \text{ Lemma 3.9}) & = & |\{i \mid \mathit{Insert}_{BCM}(p_i)\}| \\ (\text{Def. } BCM) & = & |\{i \mid \mathit{Comp}_{BCM}(p_i)\}| \end{array}$$

The inequation marked by $(*)$ needs a short explanation. Due to Lemma 3.16(2) we are able to map every index i with $\mathit{Insert}_{ALCM}(p_i)$ (which in particular implies $\mathit{Delayed}(p_i)$) to a lifetime range $p[j,l] \in \mathit{FU\text{-}LtRg}(BCM)$ with $j \leq i \leq l$. Following 3.17(2) this mapping is injective, which justifies $(*)$.

Ad 3) Let $p \in \mathit{LtRg}(ALCM)$ with $\lambda_p > 1$, and $CM \in \mathcal{CM}_{CmpOpt}$. Then we have:

$$\begin{array}{rcl} & & p \in \mathit{LtRg}(ALCM) \\ (\text{Lemma 3.14}) & \Rightarrow & \exists\, q \in \mathit{LtRg}(BCM).\ p \sqsubseteq q \\ (\text{Lemma 3.17(2)}) & \Rightarrow & \neg \mathit{Delayed}^{\exists}(\,p]1,\lambda_p]\,) \\ (\lambda_p > 1 \ \& \ \text{Lemma 3.16(3)}) & \Rightarrow & \mathit{Repl}_{CM}(p_{\lambda_p}) \wedge \neg \mathit{Insert}^{\exists}_{CM}(\,p]1,\lambda_p]\,) \\ (CM \in \mathcal{CM}_{Adm}) & \Rightarrow & \exists\, q \in \mathit{LtRg}(CM).\ p \sqsubseteq q \end{array}$$

□

3.3.3 The Lazy Code Motion Transformation

The $ALCM$-transformation still replaces every code motion candidate. It therefore has to insert initializations, even if they can be used only immediately after the insertion point itself. In order to avoid this drawback, such trivial lifetime ranges must be detected. This leads to the following definition.

Definition 3.19 (CM-Isolation)
$\forall\, CM \in \mathcal{CM}\ \forall\, n \in N.$

$$\mathit{Isolated}_{CM}(n) \ \Leftrightarrow_{df} \ \forall\, p \in \mathbf{P}[n,e]\ \forall\, 1 < i \leq \lambda_p.\ \mathit{Repl}_{CM}(p_i) \Rightarrow \mathit{Insert}^{\exists}_{CM}(\,p]1,i]\,)$$

The following lemma states two important properties of isolation. The first part gives an alternative characterization of isolation by means of the notion of lifetime ranges, and the second part deals with the parameter CM in the definition of the isolation predicate. For latest program points the isolation property coincides for all computationally optimal code motion transformations CM. This allows us to choose the earliest-dependent isolation predicate for our lazy code motion transformation, which is advantageous in practice (cf. "Pragmatics" at the end of Section 4.1). The proof of Lemma 3.20 can be found in Section A.8.

LEMMA 3.20 (ISOLATION LEMMA)

1. $\forall\, CM \in \mathcal{CM}\ \forall\, n \in N.\ \mathit{Isolated}_{CM}(n) \;\Leftrightarrow\; \forall\, p \in \mathit{LtRg}(CM).\ \langle n \rangle \sqsubseteq p \Rightarrow \lambda_p = 1$
2. $\forall\, CM \in \mathcal{CM}_{CmpOpt}\ \forall\, n \in N.\ \mathit{Latest}(n) \Rightarrow (\mathit{Isolated}_{CM}(n) \;\Leftrightarrow\; \mathit{Isolated}_{BCM}(n))$

The lazy code motion transformation (*LCM*-transformation) is specified in Table 4. In fact, it is the unique lifetime optimal code motion transformation (Corollary 3.22).

• $\mathit{Insert}_{LCM}(n) =_{df} \mathit{Latest}(n) \wedge \neg \mathit{Isolated}_{BCM}(n)$
• $\mathit{Repl}_{LCM}(n) =_{df} \mathit{Comp}(n) \wedge \neg(\mathit{Latest}(n) \wedge \mathit{Isolated}_{BCM}(n))$

Table 4: The Lazy Code Motion Transformation

THEOREM 3.21 (LAZY CODE MOTION THEOREM)
The LCM-transformation is lifetime optimal, i.e., $LCM \in \mathcal{CM}_{LtOpt}$.

PROOF. Here the proof is divided into four parts, namely 1) $LCM \in \mathcal{CM}$, 2) $LCM \in \mathcal{CM}_{Adm}$, 3) $LCM \in \mathcal{CM}_{CmpOpt}$, and 4) $LCM \in \mathcal{CM}_{LtOpt}$.

Ad 1) In contrast to the *BCM*- and the *ALCM*-transformation, where all code motion candidates are replaced, it is not obvious that the *LCM*-transformation does not introduce local redundancies. Thus, we have to show that the conjunction of Insert_{LCM} and Comp implies Repl_{LCM}. Obviously, Insert_{LCM} implies $\neg\mathit{Isolated}_{BCM}$, and together with Comp this finally yields Repl_{LCM}.

Ad 2) The same argument as for the *ALCM*-transformation yields that Insert_{LCM} implies Safe. Thus, we are left with showing that Repl_{LCM} implies $\mathit{Correct}_{LCM}$, which is proved by the following case analysis.

Case 1: $\mathit{Latest}(n)$ holds. By definition (cp. Table 4) the conjunction of $\mathit{Repl}_{LCM}(n)$ and $\mathit{Latest}(n)$ implies $\neg\mathit{Isolated}_{BCM}(n)$ and therefore $\mathit{Insert}_{LCM}(n)$, which trivially establishes $\mathit{Correct}_{LCM}(n)$.

Case 2: $\mathit{Latest}(n)$ does not hold. Clearly, Repl_{LCM} implies Repl_{ALCM}. The admissibility of the *ALCM*-transformation together with $\neg\mathit{Latest}(n)$ delivers

$$\forall\, p \in \mathbf{P}[s,n]\ \exists\, i < \lambda_p.\ \mathit{Insert}_{ALCM}(p_i) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,)$$

Choosing i as the maximal index satisfying $\mathit{Insert}_{ALCM}(p_i)$ this can be strengthened as follows:

$$\forall\, p \in \mathbf{P}[s,n]\ \exists\, i < \lambda_p.\ p[i, \lambda_p] \in \mathit{LtRg}(ALCM) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,)$$

Applying Isolation Lemma 3.20(1) yields

$$\forall\, p \in \mathbf{P}[s,n]\ \exists\, i < \lambda_p.\ \mathit{Latest}(p_i) \wedge \neg\mathit{Isolated}_{ALCM}(p_i) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,)$$

which also implies according to the second part of the Isolation Lemma 3.20:

$$\forall\, p \in \mathbf{P}[s,n]\ \exists\, i < \lambda_p.\ \mathit{Latest}(p_i) \wedge \neg \mathit{Isolated}_{BCM}(p_i) \wedge \mathit{Transp}^{\forall}(\,p[i,\lambda_p[\,)$$

This finally establishes $\mathit{Correct}_{LCM}(n)$ as desired.

Ad 3) Obviously, the number of computations in every $n \in N$ is the same for both the *LCM*- and the *ALCM*-transformation.

Ad 4) Since Insert_{LCM} implies Insert_{ALCM}, and Repl_{LCM} implies Repl_{ALCM} for any $n \in N$, we have:

$$\mathit{LtRg}(LCM) \ \subseteq\ \mathit{LtRg}(ALCM)$$

Thus, the "almost lifetime optimality" of $ALCM$ carries over to LCM (cp. Theorem 3.18). It remains to show for any $CM \in \mathcal{CM}_{CmpOpt}$:

$$\forall\, \langle n \rangle \in \mathit{LtRg}(LCM)\ \exists\, p \in \mathit{LtRg}(CM).\ \langle n \rangle \sqsubseteq p \qquad \text{(I)}$$

Let $\langle n \rangle \in \mathit{LtRg}(LCM)$. According to the definition of the *LCM*-transformation $\neg \mathit{Isolated}_{BCM}(n)$ holds. Thus Lemma 3.20(2) implies $\neg \mathit{Isolated}_{ALCM}(n)$ and Lemma 3.20(1) guarantees the existence of a lifetime range $q \in \mathit{LtRg}(ALCM)$ satisfying

$$\lambda_q \geq 2 \wedge \langle n \rangle \sqsubseteq q.$$

Now the almost lifetime optimality of the *ALCM*-transformation completes the proof. □

Applying Theorem 3.8 the previous theorem can even be strengthened as follows:

Corollary 3.22
The LCM-transformation is the unique lifetime optimal code motion transformation, i.e., $\mathcal{CM}_{LtOpt} = \{LCM\}$

4 PRACTICE

In the previous section, transformations for busy and lazy code motion were presented with respect to t-refined flowgraphs. The goal of this section is to present practise-oriented algorithms realizing the *BCM*- and *LCM*-transformation, which do not require the construction of t-refined flowgraphs. This is achieved in three steps. First, the implementation-relevant information about the algorithms is summarized in a concise and self-contained way. Second, a detailed discussion of the introductory example is given. And third, the gap between the theoretical and the practical presentations of the transformations is closed, showing that the practical versions do in fact satisfy the optimality criterea of Section 3.

4.1 Implementing Lazy Code Motion

In this section, we present our algorithms for busy and lazy code motion using standard notation (cf. [27]), where "+", "·" and "overlining" denote disjunction, conjunction and negation, respectively. Moreover, predicates defined on complete basic blocks will be distinguished from predicates defined on basic block parts using bold face notation. Working directly on the argument flowgraph requires the refinement of the predicates introduced in Section 3, e.g., *Comp* defined for basic block parts is split into an entry-predicate N-COMP and an exit-predicate X-COMP for nodes $\mathbf{n} \in \mathbf{N}$. The predicates for the local properties of a basic block $\mathbf{n}$ are:[16]

- $\mathsf{TRANSP}_{\mathbf{n}}$: $\mathbf{n}$ is *transparent* for t.
- $\mathsf{N\text{-}COMP}_{\mathbf{n}}$: $\mathbf{n}$ has an *entry computation* of t.
- $\mathsf{X\text{-}COMP}_{\mathbf{n}}$: $\mathbf{n}$ has an *exit computation* of t.

Finally, like in Section 3 (cp. Table 1) the insertion and replacement predicates of Table 6 and Table 7 specify a code motion transformation in the way described in Table 5. Remember that all predicates of Table 6 and Table 7 refer to the uniquely determined *insertion points* of a basic block (cf. Section 2.4).

1. Introduce a new temporary variable $\mathbf{h}_{CM}$ for t.
2. (a) Insert assignments $\mathbf{h}_{CM} := t$ at the insertion point of the entry part of all $\mathbf{n} \in \mathbf{N}$ satisfying $\mathsf{N\text{-}INSERT}^{\mathsf{CM}}$
   (b) Insert assignments $\mathbf{h}_{CM} := t$ at the insertion point of the exit part of all $\mathbf{n} \in \mathbf{N}$ satisfying $\mathsf{X\text{-}INSERT}^{\mathsf{CM}}$
3. (a) Replace the (unique) entry computation of t by $\mathbf{h}_{CM}$ in every $\mathbf{n} \in \mathbf{N}$ satisfying $\mathsf{N\text{-}REPLACE}^{\mathsf{CM}}$
   (b) Replace the (unique) exit computation of t by $\mathbf{h}_{CM}$ in every $\mathbf{n} \in \mathbf{N}$ satisfying $\mathsf{X\text{-}REPLACE}^{\mathsf{CM}}$

Table 5: General Pattern of a Code Motion Transformation

Pragmatics

The two analyses involved in the *BCM*-transformation are completely independent of each other. Thus, they can be computed in parallel in order to achieve a further speedup of the algorithm. In fact, the choice of the earliest-dependent isolation predicate also allows the parallel computation of the other two analyses[17] of the *LCM*-transformation (cp. Section 3.3). Thus, the *LCM*-transformation can be computed in only two sequential steps, which improves on our presentation in [22], where all four computation steps must be determined sequentially.

Remark 4.1 It is worth noting that the delay analysis of the *LCM*-transformation can also be fed with the insertion points delivered by the original algorithm of Morel and Renvoise [27]. Due to the specialities of their framework, which assumes that the result of every computation is written to a (symbolic) register,[18] computations cannot be

[16] In [27] N-COMP is called ANTLOC and a predicate COMP is used in place of the slightly different X-COMP.
[17] Note that Table 7(3) does not specify a data flow analysis!
[18] The isolation analysis determines, whether it is advantegous to initialize a register, which is justified whenever the considered value may be used twice.

1. Safety Analyses:

a) Down-Safety Analysis

$$\mathsf{N\text{-}D\text{-}SAFE}_{\mathbf{n}} = \mathsf{N\text{-}COMP}_{\mathbf{n}} + \mathsf{TRANSP}_{\mathbf{n}} \cdot \mathsf{X\text{-}D\text{-}SAFE}_{\mathbf{n}}$$

$$\mathsf{X\text{-}D\text{-}SAFE}_{\mathbf{n}} = \mathsf{X\text{-}COMP}_{\mathbf{n}} + \begin{cases} \mathit{false} & \mathit{if}\ \mathbf{n} = \mathbf{e} \\ \prod_{\mathbf{m} \in succ(\mathbf{n})} \mathsf{N\text{-}D\text{-}SAFE}_{\mathbf{m}} & \mathit{otherwise} \end{cases}$$

$\leadsto$ Greatest fixed point solution: $\mathsf{N\text{-}D\text{-}SAFE}^{\star}$ and $\mathsf{X\text{-}D\text{-}SAFE}^{\star}$

b) Up-Safety Analysis

$$\mathsf{N\text{-}U\text{-}SAFE}_{\mathbf{n}} = \begin{cases} \mathit{false} & \mathit{if}\ \mathbf{n} = \mathbf{s} \\ \prod_{\mathbf{m} \in pred(\mathbf{n})} (\mathsf{X\text{-}COMP}_{\mathbf{m}} + \mathsf{X\text{-}U\text{-}SAFE}_{\mathbf{m}}) & \mathit{otherwise} \end{cases}$$

$$\mathsf{X\text{-}U\text{-}SAFE}_{\mathbf{n}} = \mathsf{TRANSP}_{\mathbf{n}} \cdot (\mathsf{N\text{-}COMP}_{\mathbf{n}} + \mathsf{N\text{-}U\text{-}SAFE}_{\mathbf{n}})$$

$\leadsto$ Greatest fixed point solution: $\mathsf{N\text{-}U\text{-}SAFE}^{\star}$ and $\mathsf{X\text{-}U\text{-}SAFE}^{\star}$

2. Computation of Earliestness: (No data flow analysis!)

$$\mathsf{N\text{-}EARLIEST}_{\mathbf{n}} =_{df} \mathsf{N\text{-}D\text{-}SAFE}^{\star}_{\mathbf{n}} \cdot \prod_{\mathbf{m} \in pred(\mathbf{n})} (\overline{\mathsf{X\text{-}U\text{-}SAFE}^{\star}_{\mathbf{m}} + \mathsf{X\text{-}D\text{-}SAFE}^{\star}_{\mathbf{m}}})$$

$$\mathsf{X\text{-}EARLIEST}_{\mathbf{n}} =_{df} \mathsf{X\text{-}D\text{-}SAFE}^{\star}_{\mathbf{n}} \cdot \overline{\mathsf{TRANSP}_{\mathbf{n}}}$$

3. Insertion and Replacement Points of the Busy Code Motion Transformation:

$$\mathsf{N\text{-}INSERT}^{\mathsf{BCM}}_{\mathbf{n}} =_{df} \mathsf{N\text{-}EARLIEST}_{\mathbf{n}}$$
$$\mathsf{X\text{-}INSERT}^{\mathsf{BCM}}_{\mathbf{n}} =_{df} \mathsf{X\text{-}EARLIEST}_{\mathbf{n}}$$

$$\mathsf{N\text{-}REPLACE}^{\mathsf{BCM}}_{\mathbf{n}} =_{df} \mathsf{N\text{-}COMP}_{\mathbf{n}}$$
$$\mathsf{X\text{-}REPLACE}^{\mathsf{BCM}}_{\mathbf{n}} =_{df} \mathsf{X\text{-}COMP}_{\mathbf{n}}$$

Table 6: The Busy Code Motion Transformation

isolated by definition, and hence, the isolation analysis can be dropped. Minor modifications of the delay- and latestness analysis due to the different choice of where to insert computations in a basic block are straightforward and similar to those of [13]. Hence, the effect of lazy code motion can be achieved by completely retaining a possibly existing implementation of Morel and Renvoise's algorithm. Note, however, that the overall

1. *Perform steps 1) and 2) of Table 6.*

2. *Delayability Analysis:*

$$\mathsf{N\text{-}DELAYED}_{\mathbf{n}} = \mathsf{N\text{-}EARLIEST}_{\mathbf{n}} + \begin{cases} \mathit{false} & \mathit{if} \quad \mathbf{n} = \mathbf{s} \\ \prod_{\mathbf{m} \in pred(\mathbf{n})} \overline{\mathsf{X\text{-}COMP}_{\mathbf{m}}} \cdot \mathsf{X\text{-}DELAYED}_{\mathbf{m}} & \mathit{otherwise} \end{cases}$$

$$\mathsf{X\text{-}DELAYED}_{\mathbf{n}} = \mathsf{X\text{-}EARLIEST}_{\mathbf{n}} + \mathsf{N\text{-}DELAYED}_{\mathbf{n}} \cdot \overline{\mathsf{N\text{-}COMP}_{\mathbf{n}}}$$

$\rightsquigarrow$ Greatest fixed point solution: $\mathsf{N\text{-}DELAYED}^{\star}$ and $\mathsf{X\text{-}DELAYED}^{\star}$

3. *Computation of Latestness: (No data flow analysis!)*

$$\mathsf{N\text{-}LATEST}_{\mathbf{n}} =_{df} \mathsf{N\text{-}DELAYED}^{\star}_{\mathbf{n}} \cdot \mathsf{N\text{-}COMP}_{\mathbf{n}}$$

$$\mathsf{X\text{-}LATEST}_{\mathbf{n}} =_{df} \mathsf{X\text{-}DELAYED}^{\star}_{\mathbf{n}} \cdot \big(\mathsf{X\text{-}COMP}_{\mathbf{n}} + \sum_{\mathbf{m} \in succ(\mathbf{n})} \overline{\mathsf{N\text{-}DELAYED}^{\star}_{\mathbf{m}}} \big)$$

4. *Isolation Analysis:*

$$\mathsf{N\text{-}ISOLATED}_{\mathbf{n}} = \mathsf{X\text{-}EARLIEST}_{\mathbf{n}} + \mathsf{X\text{-}ISOLATED}_{\mathbf{n}}$$

$$\mathsf{X\text{-}ISOLATED}_{\mathbf{n}} = \prod_{\mathbf{m} \in succ(\mathbf{n})} \mathsf{N\text{-}EARLIEST}_{\mathbf{m}} + \overline{\mathsf{N\text{-}COMP}_{\mathbf{m}}} \cdot \mathsf{N\text{-}ISOLATED}_{\mathbf{m}}$$

$\rightsquigarrow$ Greatest fixed point solution: $\mathsf{N\text{-}ISOLATED}^{\star}$ and $\mathsf{X\text{-}ISOLATED}^{\star}$

5. *Insertion and Replacement Points of the Lazy Code Motion Transformation:*

$$\begin{aligned} \mathsf{N\text{-}INSERT}^{\mathsf{LCM}}_{\mathbf{n}} &=_{df} \mathsf{N\text{-}LATEST}_{\mathbf{n}} \cdot \overline{\mathsf{N\text{-}ISOLATED}^{\star}_{\mathbf{n}}} \\ \mathsf{X\text{-}INSERT}^{\mathsf{LCM}}_{\mathbf{n}} &=_{df} \mathsf{X\text{-}LATEST}_{\mathbf{n}} \cdot \overline{\mathsf{X\text{-}ISOLATED}^{\star}_{\mathbf{n}}} \\ \\ \mathsf{N\text{-}REPLACE}^{\mathsf{LCM}}_{\mathbf{n}} &=_{df} \mathsf{N\text{-}COMP}_{\mathbf{n}} \cdot \overline{\mathsf{N\text{-}LATEST}_{\mathbf{n}} \cdot \mathsf{N\text{-}ISOLATED}^{\star}_{\mathbf{n}}} \\ \mathsf{X\text{-}REPLACE}^{\mathsf{LCM}}_{\mathbf{n}} &=_{df} \mathsf{X\text{-}COMP}_{\mathbf{n}} \cdot \overline{\mathsf{X\text{-}LATEST}_{\mathbf{n}} \cdot \mathsf{X\text{-}ISOLATED}^{\star}_{\mathbf{n}}} \end{aligned}$$

Table 7: The Lazy Code Motion Transformation

complexity of an upgraded implementation of their algorithm is worse than that of our *LCM*-transformation, since their algorithm requires the computation of the predicates up-safe, down-safe,[19] partial available, and placement possible, which is even bidirectional,

[19]Up-safety and down-safety are called availability and anticipability in [27], respectively.

whereas our *LCM*-transformation is only based on the (unidirectional) computation of up-safety and down-safety.

4.2 Example

In this section we illustrate the computation of the entry- and exit-predicates defined in Section 4.1 for the introductory example of Figure 1. As described in Section 2.2, we assume that every critical edge of the original flowgraph is split by inserting a synthetic node. This step transforms the flowgraph of Figure 5 to the one of Figure 6. Here a synthetic node inserted on an edge leading from a basic block $\mathbf{n}$ to a basic block $\mathbf{m}$ is represented by a dashed box with name $\mathbf{S_{n,m}}$.

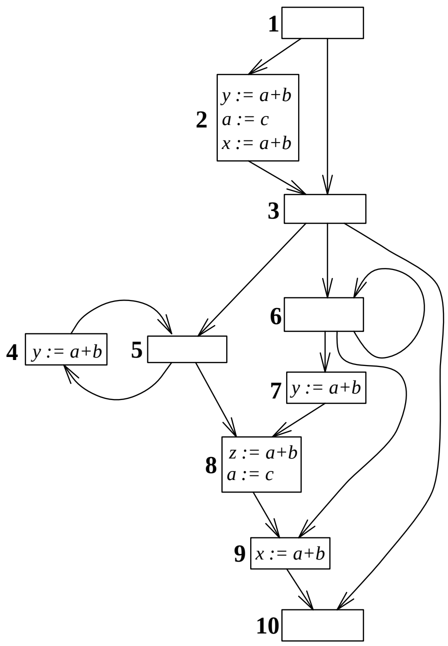

Figure 5: The Motivating Example

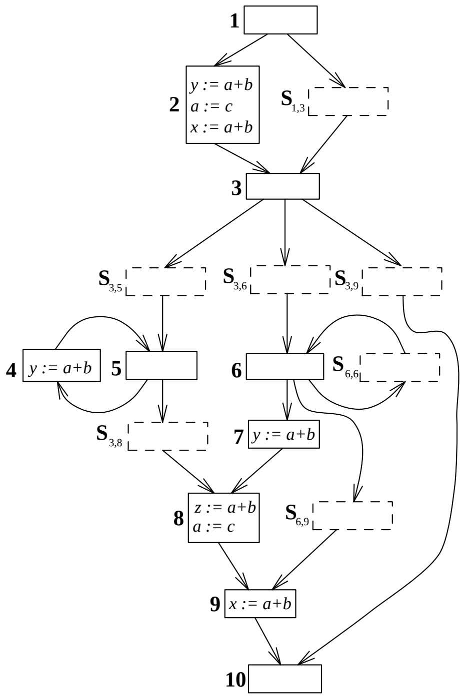

Figure 6: Synthetic Nodes Inserted

Figure 7 shows how entry- and exit-earliestness are determined according to the results of the down-safety and up-safety analysis. Intuitively, the up-safety (down-safety) analysis propagates information in the (opposite) direction of control flow starting with program points containing a code motion candidate of t as long as t is not modified. Here and in the following, entry- and exit-predicates are represented by markings which are pinned at the entry and exit of the basic blocks satisfying the predicate under consideration.

Clearly, every basic block, which is not transparent, but exit-down-safe, is exit-earliest. This holds for the basic blocks **2** and **8**. Additionally, basic block **2** and the synthetic nodes $\mathbf{S_{3,5}}$ and $\mathbf{S_{3,6}}$ are entry-earliest, because they are entry-down-safe due to an entry computation, and none of their immediate predecessors is exit-down-safe or exit-up-safe. Figure 8 shows the result of the busy code motion transformation.[20]

[20]Synthetic nodes without computations can always be suppressed.

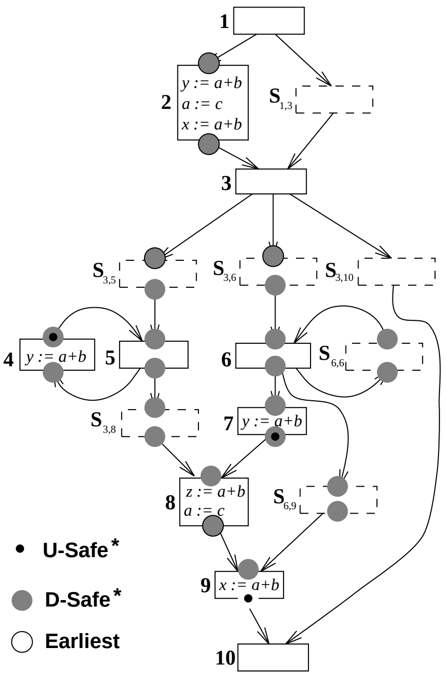

Figure 7: Computation of Earliestness

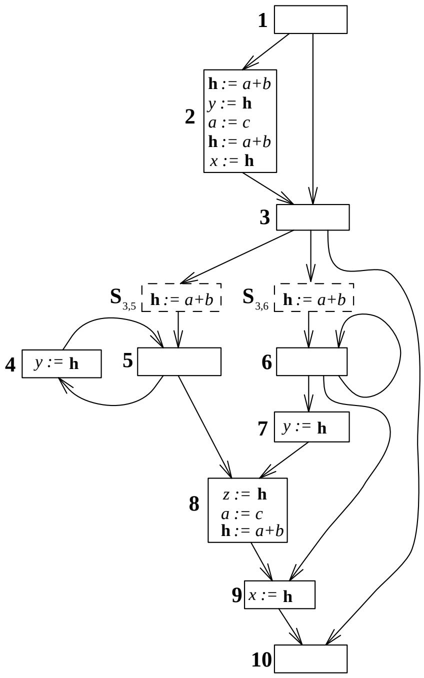

Figure 8: The Busy Code Motion Transformation

Figure 9 illustrates the entry- and exit-latestness predicates, which result from moving the initializations from the earliest program points in the direction of control flow until any further movement would destroy the computational optimality. Thus, every entry-delayable basic block with an entry computation is entry-latest. In Figure 9 these are the basic blocks **2**, **7** and **9**. Moreover, an exit-delayable basic block is exit-latest, if it has an exit computation like the basic block **2**, or if one of its immediate successors is not entry-delayable like the synthetic node $\mathbf{S_{3,5}}$.

As illustrated in Figure 10, the corresponding almost lazy code motion transformation inserts entry and exit initializations $\mathbf{h} := a+b$ in every basic block being entry-latest and exit-latest, respectively, and replaces all code motion candidates of $a+b$ by $\mathbf{h}$.

Note that both initializations in basic blocks **2** and **9** are unnecessary, since they are only used immediately afterwards in the same block. This effect is finally suppressed by means of the isolation analysis.

Figure 11 displays the result of the backward analysis for entry- and exit-isolation. Intuitively, this analysis propagates the isolation information, which is obviously valid at the exit of the program as well as immediately before a reinitialization at an earliest program point, towards the start node until an initialization would cover a code motion candidate of t later on. For example the entry-latest basic block **9** is also entry-isolated, whereas the entry-latest basic block **7** is not. The lazy code motion transformation presented in Figure 12 differs from the almost lazy code motion transformation of Figure 10 in that isolated initializations are suppressed and the corresponding code motion candidates remain in the program. This concerns all initializations in entry or exit parts that are marked both isolated and latest.

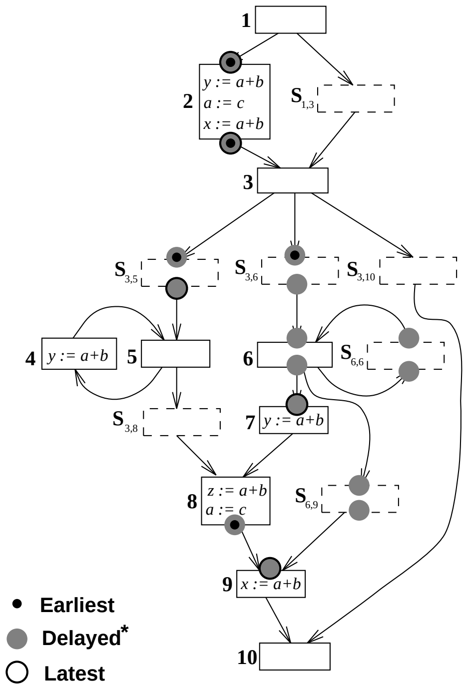

Figure 9: Computation of Latestness

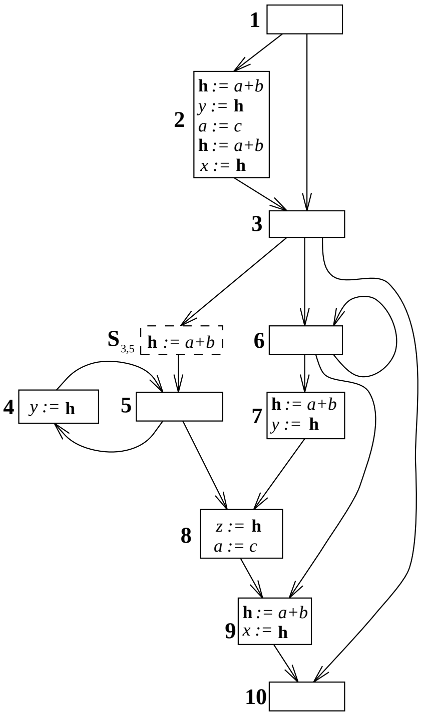

Figure 10: The Almost Lazy Code Motion Transformation

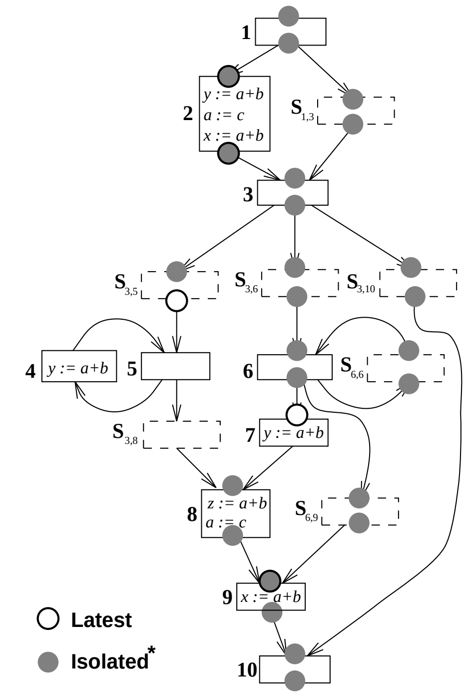

Figure 11: The Isolation Analysis

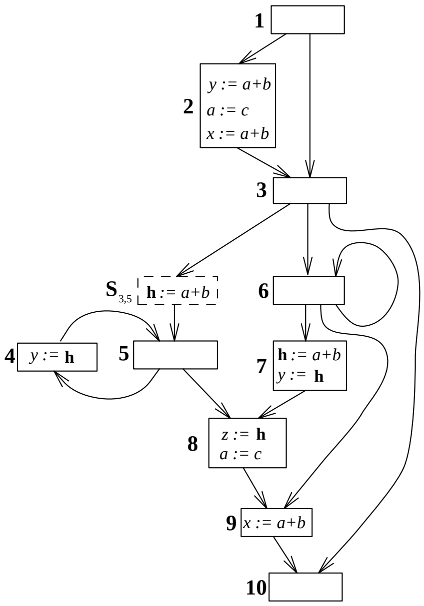

Figure 12: The Lazy Code Motion Transformation

4.3 Combining Theory and Practice

In this section the gap between the theoretical and the practical presentations of the busy and lazy code motion transformations is closed by proving the equivalence of their corresponding predicates. Clearly, this is sufficient to establish the optimality results of the theoretical Section 3 for the implementation oriented algorithms specified in Tables 6 and 7. The equivalence of the local predicates is trivial:

LEMMA 4.2 *Let* $\mathbf{n} \in \mathbf{N}$. *Then we have:*

1. $\mathit{Transp}\,(n_N) \;\Leftrightarrow\; \mathsf{TRANSP}_{\mathbf{n}}$
2. $\mathit{Comp}\,(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}COMP}_{\mathbf{n}}$
   $\mathit{Comp}\,(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}COMP}_{\mathbf{n}}$

The proof for the global predicates is more involved:

THEOREM 4.3 (IMPLEMENTATION THEOREM)
Let $\mathbf{n} \in \mathbf{N}$. *Then we have:*

1. $\mathit{D\text{-}Safe}(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}D\text{-}SAFE}^\star_{\mathbf{n}}$
   $\mathit{D\text{-}Safe}(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}D\text{-}SAFE}^\star_{\mathbf{n}}$
2. $\mathit{U\text{-}Safe}(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}U\text{-}SAFE}^\star_{\mathbf{n}}$
   $\mathit{U\text{-}Safe}(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}U\text{-}SAFE}^\star_{\mathbf{n}}$
3. $\mathit{Earliest}\,(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}EARLIEST}_{\mathbf{n}}$
   $\mathit{Earliest}\,(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}EARLIEST}_{\mathbf{n}}$
4. $\mathit{Delayed}\,(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}DELAYED}^\star_{\mathbf{n}}$
   $\mathit{Delayed}\,(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}DELAYED}^\star_{\mathbf{n}}$
5. $\mathit{Latest}\,(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}LATEST}_{\mathbf{n}}$
   $\mathit{Latest}\,(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}LATEST}_{\mathbf{n}}$
6. $\mathit{Isolated}_{BCM}\,(n_N) \;\Leftrightarrow\; \mathsf{N\text{-}ISOLATED}^\star_{\mathbf{n}}$
   $\mathit{Isolated}_{BCM}\,(n_X) \;\Leftrightarrow\; \mathsf{X\text{-}ISOLATED}^\star_{\mathbf{n}}$

PROOF. (3) and (5) are simple consequences of (1), (2), (4) and (6), where in the case of (3) also Lemma 3.11(2) is used. Central for the "$\Rightarrow$"-direction of the remaining proofs is to derive an invariant for the predicates of the theory part from their definition.[21] Conversely, the "$\Leftarrow$"-direction can be proved by means of a suitable induction showing that the fixed point characterization of the practical part implies the path condition stated in the theoretical part. Since the proofs for all remaining parts are analogous, we only evolve the first part here.

"$\Rightarrow$": Here, the definition of down-safety directly yields as the appropriate invariant:

$$\mathit{D\text{-}Safe}(n) \;\Leftrightarrow\; \mathit{Comp}\,(n) \vee \begin{cases} \mathit{false} & \mathit{if}\ n = e \\ \mathit{Transp}\,(n) \wedge \bigwedge\limits_{m \in succ(n)} \mathit{D\text{-}Safe}(m) & \mathit{otherwise} \end{cases} \tag{I}$$

Additionally, *D-Safe* induces two global predicates on basic blocks $\mathsf{N\text{-}FIX}$ and $\mathsf{X\text{-}FIX}$ defined by $\mathsf{N\text{-}FIX}_{\mathbf{n}} =_{df} \mathit{D\text{-}Safe}(n_N)$ and $\mathsf{X\text{-}FIX}_{\mathbf{n}} =_{df} \mathit{D\text{-}Safe}(n_X)$ for any $\mathbf{n} \in \mathbf{N}$.

$\mathsf{N\text{-}D\text{-}SAFE}^\star$ and $\mathsf{X\text{-}D\text{-}SAFE}^\star$ are defined as the greatest fixed point solution of the equation system of part a) of Table 6. In order to complete the proof of the first implication it suffices to show that $\mathsf{N\text{-}FIX}$ and $\mathsf{X\text{-}FIX}$ define a fixed point solution of this equation system. We have:

[21] This is straightforward, except for the case of (6). Here we have:
$\mathit{Isolated}_{BCM}\,(n) \;\Leftrightarrow\; \bigwedge\limits_{m \in succ(n)} \mathit{Earliest}\,(m) \vee \neg \mathit{Comp}\,(m) \wedge \mathit{Isolated}_{BCM}\,(m)$.

$$\begin{array}{rcl} & & \textsf{N-FIX}_{\mathbf{n}} \\ & \Leftrightarrow & \textit{D-Safe}(n_N) \\ \text{(I)} & \Leftrightarrow & \textit{Comp}\,(n_N) \vee \textit{Transp}\,(n_N) \wedge \textit{D-Safe}(n_X) \\ \text{(Lemma 4.2)} & \Leftrightarrow & \textsf{N-COMP}_{\mathbf{n}} \vee \textsf{TRANSP}_{\mathbf{n}} \wedge \textit{D-Safe}(n_X) \\ & \Leftrightarrow & \textsf{N-COMP}_{\mathbf{n}} \vee \textsf{TRANSP}_{\mathbf{n}} \wedge \textsf{X-FIX}_{\mathbf{n}} \end{array}$$

Obviously we have $\textsf{X-FIX}_{\mathbf{e}} \Leftrightarrow \textsf{X-COMP}_{\mathbf{e}}$. Thus, let $\mathbf{n} \in \mathbf{N} \backslash \{\mathbf{e}\}$, where we have:

$$\begin{array}{rcl} & & \textsf{X-FIX}_{\mathbf{n}} \\ & \Leftrightarrow & \textit{D-Safe}(n_X) \\ \text{(I)} & \Leftrightarrow & \textit{Comp}\,(n_X) \vee \textit{Transp}\,(n_X) \wedge \bigwedge_{m \in succ(n_X)} \textit{D-Safe}(m) \\ & \Leftrightarrow & \textit{Comp}\,(n_X) \vee \bigwedge_{\mathbf{m} \in succ(\mathbf{n})} \textit{D-Safe}(m_N) \\ \text{(Lemma 4.2)} & \Leftrightarrow & \textsf{X-COMP}_{\mathbf{n}} \vee \bigwedge_{\mathbf{m} \in succ(\mathbf{n})} \textsf{N-FIX}_{\mathbf{m}} \end{array}$$

"$\Leftarrow$": This implication is an immediate consequence of the following proposition that can be shown for every node $n \in N$ and path $p \in \mathbf{P}[n, e]$ by means of an induction on $\lambda_p - i$:

$$\forall\, 1 \leq i \leq \lambda_p.\ \textsf{N-D-SAFE}^{\star}_{\mathbf{p_i}} \vee \textsf{X-D-SAFE}^{\star}_{\mathbf{p_i}} \Rightarrow \exists\, j \geq i.\ \textit{Comp}\,(p_j) \wedge \textit{Transp}\,(\,p[i, j[\,)$$

□

5 Conclusions

We have presented a practise-oriented adaptation of the *computationally* and *lifetime optimal* code motion algorithm of [22], which is composed of unidirectional standard analyses, and works on flowgraphs whose nodes are basic blocks rather than single statements. As the original algorithm it yields optimal results and is as efficient as the well-known algorithms for unidirectional bit-vector analysis.

The modularity of our algorithm supports further extensions: in [23] we present an extension of our lazy code motion algorithm which, in a similar fashion as in [7, 18, 19], uniformly combines code motion and strength reduction, and following the lines of [24, 25] a generalization to programs with procedures, global and local variables and formal value parameters is straightforward. We are also investigating an adaptation of the *as early as necessary* but *as late as possible* placement strategy to the semantically based code motion algorithms of [32, 33]. Moreover, the two extreme placing strategies of generating computationally optimal programs, which are realized by our algorithms for busy and lazy code motion, offer a well-founded platform for developing a variant that also minimizes the overall register pressure of a program.

Acknowledgements

The authors would like to thank the anonymous referees for their conscientious and fast reviews and their constructive criticism, which improved the presentation of the paper.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[2] Aho, A. V., and Ullman, J. D. Node listings for reducible flow graphs. In *Proc. 7^{th} ACM Symposium on the Theory of Computing* (Albuquerque, NM, 1975), pp. 177 – 185.

[3] Chow, F. *A portable machine independent optimizer – Design and measurements.* PhD thesis, Stanford University, Dept. of Electrical Engineering, Stanford, CA, 1983. Published as Tech. Rep. 83-254, Computer Systems Lab., Stanford University.

[4] Cocke, J., and Schwartz, J. T. Programming languages and their compilers. Courant Institute of Mathematical Sciences, NY, 1970.

[5] Dhamdhere, D. M. Characterization of program loops in code optimization. *Journal of Computer Languages 8*, 2 (1983), 69 – 76.

[6] Dhamdhere, D. M. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices 23*, 10 (1988), 172 – 180.

[7] Dhamdhere, D. M. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *International Journal of Computer Mathematics 27* (1989), 1 – 14 (+ 31 – 32).

[8] Dhamdhere, D. M. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems 13*, 2 (1991), 291 – 294. Technical Correspondence.

[9] Dhamdhere, D. M., and Khedker, U. P. Complexity of bidirectional data flow analysis. In *Conf. Record of the 20^{th} ACM Symposium on the Principles of Programming Languages* (Charleston, SC, Jan. 1993), pp. 397–409.

[10] Dhamdhere, D. M., and Patil, H. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems 15*, 2 (Apr. 1993), 312 – 336.

[11] Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92* (San Francisco, CA, June 1992), vol. *27*,7 of *ACM SIGPLAN Notices*, pp. 212 – 223.

[12] Drechsler, K.-H., and Stadel, M. P. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems 10*, 4 (1988), 635 – 640. Technical Correspondence.

[13] Drechsler, K.-H., and Stadel, M. P. A variation of Knoop, Rüthing and Steffen's lazy code motion. *ACM SIGPLAN Notices 28*, 5 (1993), 29 – 38.

[14] Graham, S. L., and Wegman, M. N. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM 23*, 1 (1976), 172 – 202.

[15] Hecht, M. S. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[16] Hecht, M. S., and Ullman, J. D. Analysis of a simple algorithm for global flow problems. In *Conf. Record of the 1st ACM Symposium on the Principles of Programming Languages* (Boston, MA, 1973), pp. 207 – 217.

[17] Hecht, M. S., and Ullman, J. D. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing 4*, 4 (1977), 519 – 532.

[18] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization – part I. *International Journal of Computer Mathematics 11* (1982), 21 – 41.

[19] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization – part II. *International Journal of Computer Mathematics 11* (1982), 111 – 126.

[20] Kam, J. B., and Ullman, J. D. Global data flow analysis and iterative algorithms. *Journal of the ACM 23*, 1 (1976), 158 – 171.

[21] Kennedy, K. Node listings applied to data flow analysis. In *Conf. Record of the 2nd ACM Symposium on the Principles of Programming Languages* (Palo Alto, CA, 1975), pp. 10 – 21.

[22] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92* (San Francisco, CA, June 1992), vol. *27*,7 of *ACM SIGPLAN Notices*, pp. 224 – 234.

[23] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71–91.

[24] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proc. 4th Conference on Compiler Construction (CC)* (Paderborn, Germany, 1992), Lecture Notes in Computer Science 641, Springer-Verlag, pp. 125 – 140.

[25] Knoop, J., and Steffen, B. Optimal interprocedural partial redundancy elimination. Extended abstract. In *Addenda to Proc. 4th Conference on Compiler Construction (CC)* (Paderborn, Germany, 1992), Published as Tech. Rep. No. 103, Department of Computer Science, University of Paderborn, pp. 36 – 39.

[26] Morel, E. Data flow analysis and global optimization. In *Methods and tools for compiler construction*, B. Lorho, Ed. Cambridge University Press, 1984.

[27] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2 (1979), 96 – 103.

[28] Morel, E., and Renvoise, C. Interprocedural elimination of partial redundancies. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice Hall, Englewood Cliffs, NJ, 1981, ch. 6, pp. 160 – 188.

[29] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Conf. Record of the 15th ACM Symposium on the Principles of Programming Languages* (San Diego, CA, 1988), pp. 12 – 27.

[30] Sorkin, A. Some comments on a solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems 11*, 4 (1989), 666 – 668. Technical Correspondence.

[31] Steffen, B. Data flow analysis as model checking. In *Proc. TACS* (Sendai, Japan, 1991), Lecture Notes in Computer Science 526, Springer-Verlag, pp. 346 – 364.

[32] Steffen, B., Knoop, J., and Rüthing, O. The value flow graph: A program representation for optimal program transformations. In *Proc. 3rd European Symposium on Programming (ESOP)* (Copenhagen, Denmark, 1990), Lecture Notes in Computer Science 432, Springer-Verlag, pp. 389 – 405.

[33] Steffen, B., Knoop, J., and Rüthing, O. Efficient code motion and an adaption to strength reduction. In *Proc. 4th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)* (Brighton, UK, 1991), Lecture Notes in Computer Science 494, Springer-Verlag, pp. 394 – 415.

[34] Tarjan, R. E. Applications of path compression on balanced trees. *Journal of the ACM 26*, 4 (1979), 690 – 715.

[35] Tarjan, R. E. Fast algorithms for solving path problems. *Journal of the ACM 28*, 3 (1981), 594 – 614.

[36] Tarjan, R. E. A unified approach to path problems. *Journal of the ACM 28*, 3 (1981), 577 – 593.

[37] Ullman, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2*, 3 (1973), 191 – 213.

A APPENDIX: PROOFS

A.1 Proof of the Correctness Lemma 3.5

$$
\begin{array}{rcl}
 & & \mathit{Correct}_{CM}(n) \\
(\text{Def. } 3.1(2)) & \Rightarrow & \forall\, p \in \mathbf{P}[\mathbf{s}, n]\ \exists\, i \leq \lambda_p.\ \mathit{Insert}_{CM}(p_i) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,) \\
(CM \in \mathcal{CM}_{Adm}) & \Rightarrow & \forall\, p \in \mathbf{P}[\mathbf{s}, n]\ \exists\, i \leq \lambda_p.\ \mathit{Safe}(p_i) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,) \\
(\text{Lemma } 3.3) & \Rightarrow & \forall\, p \in \mathbf{P}[\mathbf{s}, n]\ \exists\, i \leq \lambda_p. \\
 & & \qquad (\mathit{D\text{-}Safe}(p_i) \vee \mathit{U\text{-}Safe}(p_i)) \wedge \mathit{Transp}^{\forall}(\, p[i, \lambda_p[\,) \\
(\text{Def. } 3.2(1)) & \Rightarrow & \forall\, p \in \mathbf{P}[\mathbf{s}, n]\ \exists\, i \leq \lambda_p.
\end{array}
$$

$$(D\text{-}Safe(p_i) \vee Comp\,(p_i)) \wedge Transp^{\forall}(\,p[i, \lambda_p[\,)$$

$$\begin{array}{rcl} \text{(Def. 3.2)} & \Rightarrow & D\text{-}Safe(n) \vee U\text{-}Safe(n) \\ \text{(Lemma 3.3)} & \Rightarrow & Safe\,(n) \end{array}$$

□

A.2 Proof of the First-Use-Lifetime Range Lemma 3.9

Assume that $p[i_1, j_1]$ and $p[i_2, j_2]$ neither coincide nor are disjoint. In the case where one of the lifetime ranges, say $p[i_1, j_1]$, is properly contained inside the other, the definition of first-use-lifetime ranges delivers $p[i_1, j_1] = p[i_2, j_2]$, which contradicts our assumption. Thus we can assume without loss of generality that $p[i_1, j_1]$ and $p[i_2, j_2]$ intersect in a way such that $i_1 < i_2 \leq j_1 \leq j_2$. In this case, the definition of $p[i_2, j_2]$ as a lifetime range implies that $Insert_{CM}(n_{i_2})$ holds, which excludes $p[i_1, j_1]$ from being a lifetime range. □

A.3 Proof of the Earliestness Lemma 3.11

Ad 1) This is a consequence of the following proposition which can be shown for each node $n \in N$ and path $p \in \mathbf{P}[s, n]$ by means of an induction on i.

$$\forall\, i \leq \lambda_p.\ Safe\,(p_i) \Rightarrow \exists\, j \leq i.\ Earliest\,(p_j) \wedge Transp^{\forall}(\,p[j, i[\,)$$

Ad 2) Since "$\Leftarrow$" is obvious, we concentrate on the other implication. For the start node s this implication is trivially true. Otherwise, we are first going to show:

$$Earliest\,(n) \Rightarrow D\text{-}Safe(n) \qquad \text{(I)}$$

We have:

$$\begin{array}{rcl} & & Earliest\,(n) \\ \text{(Def. 3.10)} & \Rightarrow & Safe\,(n) \wedge \bigvee_{m \in pred(n)} \neg\, Transp\,(m) \vee \neg Safe\,(m) \\ \text{(Lemma 3.3)} & \Rightarrow & Safe\,(n) \wedge \bigvee_{m \in pred(n)} \neg\, Transp\,(m) \vee \neg\, U\text{-}Safe(m) \wedge \neg D\text{-}Safe(m) \\ \text{(Def. 3.2(2))} & \Rightarrow & Safe\,(n) \wedge \bigvee_{m \in pred(n)} \neg\, Transp\,(m) \vee \neg\, U\text{-}Safe(m) \wedge \neg\, Comp\,(m) \\ \text{(Def. 3.2(1))} & \Rightarrow & Safe\,(n) \wedge \neg\, U\text{-}Safe(n) \\ \text{(Lemma 3.3)} & \Rightarrow & D\text{-}Safe(n) \end{array}$$

Thus it remains to show:

$$Earliest\,(n) \Rightarrow \bigwedge_{m \in pred(n)} \neg\, Transp\,(m) \vee \neg Safe\,(m)$$

which can obviously be done by proving:

$$Earliest\,(n) \Rightarrow\ |\,pred(n)\,|\ \leq 1.$$

In order to prove the contrapositive of this implication, let us fix an immediate consequence of the definition of the t-refined flowgraph: every node $n \in N$ with multiple predecessors is an entry part with all its predecessors being exit parts. Thus we have:

$$|\,pred(n)\,| > 1 \Rightarrow \bigwedge_{m \in pred(n)} Transp\,(m) \tag{II}$$

and therefore

$$\begin{array}{rcl} & & |\,pred(n)\,| > 1 \\ \text{(Lemma 2.1 \& II)} & \Rightarrow & succ(pred(n)) = \{n\} \wedge \bigwedge_{m \in pred(n)} Transp\,(m) \\ \text{(Def. 3.2(2))} & \Rightarrow & (D\text{-}Safe(n) \Rightarrow \bigwedge_{m \in pred(n)} Transp\,(m) \wedge D\text{-}Safe(m)) \\ \text{(Def. 3.10)} & \Rightarrow & (D\text{-}Safe(n) \Rightarrow \neg Earliest\,(n)) \\ \text{(I)} & \Rightarrow & \neg Earliest\,(n) \end{array}$$

Ad 3) Since this is trivial for s, let $n \in N \backslash \{s\}$. Moreover, *Earliest* implies *Safe*, which allows us to complete the proof for "$\Rightarrow$" by showing

$$Earliest\,(n) \Rightarrow (Correct_{CM}\,(n) \Rightarrow Insert_{CM}\,(n))$$

for an arbitrary code motion transformation $CM \in \mathcal{CM}_{Adm}$. Thus, let us assume that $Earliest\,(n)$ holds. Then we have as desired:

$$\begin{array}{rcl} & & Correct_{CM}\,(n) \\ \text{(Def. 3.1(2))} & \Rightarrow & Insert_{CM}\,(n) \vee \bigwedge_{m \in pred(n)} Transp\,(m) \wedge Correct_{CM}\,(m) \\ \text{(Lemma 3.5)} & \Rightarrow & Insert_{CM}\,(n) \vee \bigwedge_{m \in pred(n)} Transp\,(m) \wedge Safe\,(m) \\ \text{(Def. 3.10)} & \Rightarrow & Insert_{CM}\,(n) \vee \neg Earliest\,(n) \\ (Earliest\,(n)) & \Rightarrow & Insert_{CM}\,(n) \end{array}$$

For the proof of "$\Leftarrow$" consider the admissible code motion transformation CM that

- inserts initialization statements at *any* node in N being earliest and
- replaces *none* of the code motion candidates except those at nodes with an initialization statement.

Although CM is useless as a concrete code motion transformation, the first part of this lemma directly delivers the following property:

$$\forall\, n \in N.\ Safe\,(n) \Rightarrow Correct_{CM}\,(n) \tag{III}$$

which allows to complete the proof:

$$
\begin{array}{rcl}
 & & (\mathit{Safe}(n) \wedge (\mathit{Correct}_{CM}(n) \Rightarrow \mathit{Insert}_{CM}(n))) \\
 & \Rightarrow & \mathit{Safe}(n) \wedge \neg \mathit{Correct}_{CM}(n) \vee \mathit{Safe}(n) \wedge \mathit{Insert}_{CM}(n) \\
(\mathrm{III}\ ) & \Rightarrow & \mathit{Safe}(n) \wedge \neg \mathit{Safe}(n) \vee \mathit{Safe}(n) \wedge \mathit{Insert}_{CM}(n) \\
 & \Rightarrow & \mathit{Insert}_{CM}(n) \\
(\text{Def. } CM) & \Rightarrow & \mathit{Earliest}(n)
\end{array}
$$

□

A.4 Proof of the Busy Code Motion Lemma 3.12

Ad 1) Let $p \in \mathbf{P}[s,e]$ and $1 \leq i \leq \lambda_p$. Since the implication backwards is a trivial consequence of the definition of lifetime ranges we are left with the other implication. Suppose that $\mathit{Insert}_{BCM}(p_i)$ holds. According to the down-safety of p_i that is granted by Lemma 3.11(2), there is an index $j \geq i$ such that

$$\mathit{Comp}(p_j) \wedge \neg \mathit{Comp}^{\exists}(\, p[i,j[\,) \wedge \mathit{Transp}^{\forall}(\, p[i,j[\,)$$

holds. Together with Definition 3.2(2) and the down-safety of p_i this implies:

$$\mathit{D\text{-}Safe}^{\forall}(\, p[i,j]\,) \wedge \mathit{Transp}^{\forall}(\, p[i,j[\,)$$

and therefore because of Lemma 3.11(2):

$$\neg \mathit{Earliest}^{\exists}(\, p]i,j]\,)$$

meaning $p[i,j] \in \mathit{FU\text{-}LtRg}(BCM)$.

Ad 2) Let $CM \in \mathcal{CM}_{Adm}$ and $p \in \mathit{LtRg}(BCM)$. Without loss of generality we can assume the code motion candidate in p_{λ_p} to be replaced, i.e., $\mathit{Repl}_{CM}(p_{\lambda_p})$ holds. Hence, due to the admissibility of CM we have:

$$\mathit{Correct}_{CM}(p_{\lambda_p}) \tag{I}$$

Moreover, according to Lemma 3.11(2) the earliestness of p_1 yields:

$$\forall\, n \in \mathit{pred}(p_1).\ \neg \mathit{Transp}(n) \vee \neg \mathit{Safe}(n)$$

and therefore together with Lemma 3.5:

$$\forall\, n \in \mathit{pred}(p_1).\ \neg \mathit{Transp}(n) \vee \neg \mathit{Correct}_{CM}(n)$$

which immediately delivers by means of the definiton of correctness (Def. 3.1(2)):

$$\forall\, i \leq \lambda_p.\ \mathit{Correct}_{CM}(p_i) \Rightarrow \exists\, j \leq i.\ \mathit{Insert}_{CM}(p_j)$$

Exploiting (I) this delivers as desired:

$$\mathit{Insert}^{\exists}_{CM}(p)$$

Ad 3) Let $CM \in \mathcal{CM}_{CmpOpt}$, $p \in \mathbf{P}[s,e]$ and i be an index such that $Comp_{CM}(p_i)$ is valid. Then the assumption

$$\forall\, j \le i \le l.\ p[j,l] \notin \mathit{FU\text{-}LtRg}(BCM)$$

would contradict the computational optimality of CM:

$$\begin{array}{rcl} & & |\{i \mid Comp_{CM}(p_i)\}| \\ \text{(Lemma 3.9 \& 3.12(2))} & \ge & |\{i \mid \exists\, j.\ p[i,j] \in \mathit{FU\text{-}LtRg}(BCM)\}| + 1 \\ \text{(Lemma 3.9 \& 3.12(1))} & = & |\{i \mid Insert_{BCM}(p_i)\}| + 1 \\ & > & |\{i \mid Insert_{BCM}(p_i)\}| \\ & = & |\{i \mid Comp_{BCM}(p_i)\}| \end{array}$$

□

A.5 Proof of Lemma 3.14

Let $CM \in \mathcal{CM}_{CmpOpt}$ and $p \in \mathbf{P}[s,e]$ such that $p[i,j] \in LtRg(CM)$. Then the definition of $p[i,j]$ as a lifetime range implies

$$\neg Insert^{\exists}_{CM}(\,p]i,j]\,)$$

and according to Lemma 3.12(2) we have:

$$\neg Insert^{\exists}_{BCM}(\,p]i,j]\,)$$

Moreover, the third part of Lemma 3.12 yields an index $l \le i$ such that $p[l,i]$ is a subpath of a first-use-lifetime range in $\mathit{FU\text{-}LtRg}(BCM)$, i.e., in particular we have:

$$Insert_{BCM}(n_l) \wedge \neg Insert^{\exists}_{BCM}(\,p]l,i]\,)$$

Summarizing, for $p[l,j]$ we obtain:

1. $p[l,j] \in LtRg(BCM)$ and
2. $p[i,j] \sqsubseteq p[l,j]$

□

A.6 Proof of the Delayability Lemma 3.16

The first part is a consequence of the following sequence of implications:

$$\begin{array}{rcl} & & Delayed(n) \\ \text{(Def. 3.15)} & \Rightarrow & \forall\, p \in \mathbf{P}[\mathbf{s},n]\ \exists\, i \le \lambda_p.\ Earliest(p_i) \wedge \neg Comp^{\exists}(\,p[i,\lambda_p[\,) \\ \text{(Lemma 3.11(2))} & \Rightarrow & \exists\, p \in \mathbf{P}[\mathbf{s},n]\ \exists\, i \le \lambda_p.\ D\text{-}Safe(p_i) \wedge \neg Comp^{\exists}(\,p[i,\lambda_p[\,) \\ \text{(Def. 3.2(2))} & \Rightarrow & D\text{-}Safe(n) \end{array}$$

For the second part, let $p \in \mathbf{P}[s,e]$ and i be an index such that $\mathit{Delayed}(p_i)$ is satisfied. Then according to the definition of delayability there exists an index $j \leq i$ with

$$\mathit{Earliest}(p_j) \wedge \neg \mathit{Comp}^{\exists}(\, p[j,i[\,)$$

Thus, Lemma 3.12(1) guarantees the required existence of an index $l \geq i$ with

$$p[j,l] \;\in \mathit{FU\text{-}LtRg}(\mathit{BCM}).$$

Finally, the third part is proved by:

$$\begin{array}{rcl} & & \mathit{Comp}_{CM}(n) \\ \text{(Lemma 3.12(3))} & \Rightarrow & (\forall\, p \in \mathbf{P}[s,e]\ \forall\, i \leq \lambda_p.\ (p_i = n) \Rightarrow \\ & & \qquad \exists\, j \leq i \leq l.\ p[j,l] \;\in \mathit{FU\text{-}LtRg}(\mathit{BCM})) \\ \text{(Def. } \mathit{FU\text{-}LtRg}(\mathit{BCM})\text{)} & \Rightarrow & \forall\, p \in \mathbf{P}[s,n]\ \exists\, j \leq \lambda_p.\ \mathit{Earliest}(p_j) \wedge \neg \mathit{Comp}^{\exists}(\, p[j,\lambda_p[\,) \\ \text{(Def. 3.15)} & \Rightarrow & \mathit{Delayed}(n) \end{array}$$

□

A.7 Proof of the Latestness Lemma 3.17

The first part directly follows from the slightly more general proposition below, which can be proved by means of a simple induction on $\lambda_p - i$:

$$\forall\, i \leq \lambda_p.\ \mathit{Delayed}(p_i) \Rightarrow \exists\, j \geq i.\ \mathit{Latest}(p_j)$$

For the proof of the second part, let $p \in \mathit{LtRg}(\mathit{BCM})$ and i be an index satisfying $\mathit{Latest}(p_i)$. If $i = \lambda_p$ the lemma is trivial. Thus we are left with the case that $i < \lambda_p$. Due to the definition of a lifetime range we have in particular:

$$\neg \mathit{Earliest}^{\exists}(\, p]i,\lambda_p]\,) \tag{I}$$

To show that $\mathit{Latest}(p_i)$ implies $\neg \mathit{Delayed}(p_{i+1})$ we have to examine two cases according to the definition of latestness: Provided the case that $\mathit{Latest}(p_i)$ is due to $\mathit{Comp}(p_i)$ the definition of delayability together with $\neg \mathit{Earliest}(p_{i+1})$ implies $\neg \mathit{Delayed}(p_{i+1})$ as desired; if $\mathit{Latest}(p_i)$ is due to $\neg \mathit{Delayed}(m)$ for some successor m of p_i, we have nothing to show if $m = p_{i+1}$, otherwise the Control Flow Lemma 2.1(2) and the definition of delayability (Def. 3.15) delivers $\mathit{Comp}(p_i)$ like in the first case. Thus, we have

$$\neg \mathit{Delayed}(p_{i+1}).$$

In the presence of property (I) the definition of delayability finally implies:

$$\neg \mathit{Delayed}^{\exists}(\, p]i,j]\,)$$

□

A.8 Proof of the Isolation Lemma 3.20

Since the first part is a trivial consequence of the definitions of isolation and lifetime ranges, we are left with the proof of the second part. Thus let $CM \in \mathcal{CM}_{CmpOpt}$, and $n \in N$ such that $Latest(n)$ holds. Then it remains to show:

$$Isolated_{CM}(n) \;\Leftrightarrow\; Isolated_{BCM}(n)$$

For the proof of "⇒" we show the contrapositive. Here $\neg Isolated_{BCM}(n)$ delivers:

$$\exists\, p \in \mathbf{P}[n,e]\ \exists\, i > 1.\ Repl_{BCM}(p_i) \wedge \neg Insert^{\exists}_{BCM}(\,p]1,i]\,)$$

which, according to the admissibility of BCM, implies:

$$\exists\, p \in \mathbf{P}[n,e]\ \exists\, i > 1\ \exists\, q \in LtRg(BCM).\ p[1,i] \sqsubseteq q$$

Since $Latest(n)$ holds, we are able to apply the Latestness Lemma 3.17(2), yielding:

$$\exists\, p \in \mathbf{P}[n,e]\ \exists\, i > 1.\ \neg Delayed^{\exists}(\,p]1,i]\,)$$

Thus, the third part of the Delayability Lemma 3.16 delivers:

$$\exists\, p \in \mathbf{P}[n,e]\ \exists\, i > 1.\ Repl_{CM}(p_i) \wedge \neg Insert^{\exists}_{CM}(\,p]1,i]\,)$$

which proves $\neg Isolated_{CM}(n)$ as desired.

For the converse implication "⇐", we have according to the definition of isolation:

$$\forall\, p \in \mathbf{P}[n,e]\ \forall\, i > 1.\ Repl_{BCM}(p_i) \Rightarrow Insert^{\exists}_{BCM}(\,p]1,i]\,)$$

Obviously, this can be rewritten as:

$$\forall\, p \in \mathbf{P}[n,e]\ \forall\, i > 1.\ Repl_{BCM}(p_i) \Rightarrow \exists\, 1 < j \leq i.\ p[j,i] \in LtRg(BCM)$$

Using the Busy Code Motion Lemma 3.12(2) and the fact that $Repl_{CM}$ implies $Repl_{BCM}$, we obtain:

$$\forall\, p \in \mathbf{P}[n,e]\ \forall\, i > 1.\ Repl_{CM}(n_i) \Rightarrow Insert^{\exists}_{CM}(\,p]1,i]\,)$$

which shows $Isolated_{CM}(n)$ by definition. □