# Overview

## Expression Motion

*Expression motion* is a technique for suppressing the computation of partially redundant expressions where possible, i. e. expressions that are unnecessarily reevaluated on some program paths at run-time. This is achieved by replacing the original computations of a program by temporary variables (registers) that are initialised *correctly* at suitable program points. A major advantage of expression motion is the fact that it uniformly covers *loop invariant expression motion* and the *elimination of redundant computations*.

In their seminal paper [MR79] Morel and Renvoise were the first who proposed an algorithm for expression motion being based upon data flow analysis techniques. Their algorithm triggered a number of variations and improvements mainly focusing on two drawbacks of Morel's and Renvoise's algorithm [Cho83, Dha83, Dha88, Dha89b, Dha91, DS88, JD82a, JD82b, Mor84, MR81, Sor89]). First, their algorithm was given in terms of bidirectional data flow analyses which are in general conceptually and computationally more complex than unidirectional ones, and second, expressions are unnecessarily moved, a fact which may increase register pressure. In [DRZ92] Dhamdhere, Rosen, and Zadeck showed that the original transformation of Morel and Renvoise can be solved as easily as a unidirectional problem. However, they did not address the problem of unnecessary code motion. This problem was first tackled in [Cho83, Dha88, Dha91] and more recently in [DP93]. 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 flow graph under consideration.

In [KRS92] we developed an algorithm for *lazy expression motion* which evolved from a total redesign of Morel's and Renvoise's algorithm starting from a specification point of view. In fact, our algorithm was the first that succeeded in solving both deficiencies of Morel's and Renvoise's algorithm optimally: the algorithm is completely based on simple, purely unidirectional data flow analyses[9] and suppresses any unnecessary code motion.[10] In fact, for a given expression this placing strategy *minimises* the lifetime range of the temporary that is associated with the expression: any other computationally optimal expression motion has to cover this lifetime range as well.

## Expression Motion of Multiple Expressions

Typically, algorithms for expression motion are presented with respect to a fixed but arbitrary expression pattern. This is due to the fact that an

[9] The idea for this was first proposed in [Ste91].

[10] This algorithm was later interprocedurally generalised to programs with procedures, local and global variables and formal value parameters in [KS92, Kno93, KRS96b].

O. Ruething: Interacting Code Motion Transformations, LNCS 1539, pp. 19-20, 1998.

extension to multiple expression patterns is straightforward for sets of expressions with a flat structure, i. e. sets of expressions that do not contain both expressions and their subexpressions. In this case a simultaneous algorithm is essentially determined as the independent combination of all individual transformations. Such a situation is for instance given when considering programs whose expressions are completely decomposed into three-address format. Even though such a decomposition severely weakens the power of expression motion, it is fairly standard since Morel's and Renvoise's seminal paper [MR79]. In fact, all relevant algorithms are based on this separation paradigm of Morel/Renvoise-like expression motion. Whereas giving up this paradigm does not severely influence considerations with respect to computational optimality, it affects lifetime considerations. This observation did not yet enter the reasoning on the register pressure problem in expression motion papers. In fact, this does not surprise, as the problem requires one to cope with subtle trade-offs between the lifetimes of different temporary variables. Nonetheless, the problem can be tackled and solved efficiently: in Chapter 4 we present the first algorithm for lifetime optimal expression motion that adequately copes with large expressions and their subexpressions simultaneously.

### Expression Motion and Critical Edges

It is well-known that critical edges are the reason for various problems in expression motion. The major deficiencies are that critical edges may cause poor transformations due to the lack of suitable placement points and higher solution costs of the associated data flow analyses. In Chapter 5 we investigate the reasons for known and for new difficulties caused by the presence of critical edges.

### Conventions

As in [KRS92] we consider flow graphs whose nodes are elementary statements rather than basic blocks. Following the lines of [KRS94a] we can easily develop algorithms where the global data flow analyses operate on basic blocks whose instructions are only inspected once in a preprocess.[11] Moreover, we primarily investigate the global aspects of lifetimes, i. e. we abstract from lifetimes of temporaries that do not survive the boundaries of a node in the flow graph. However, local aspects of lifetimes can be completely captured within a postprocess of our algorithm that requires one additional analysis (cf. [KRS92, KRS94a]). The whole part refers to a fixed flow graph $G$ which, however, in Chapter 3 and Chapter 4 is assumed to be out of $\mathfrak{FG}$ and in Chapter 5 to be out of $\mathfrak{FG}_{\mathtt{Crit}}$.

[11] Note that such an adaption is only superior from a pragmatic point of view but does not reduce the asymptotic computational complexity.