Loop-invariant hoisting in strict functional language with lazy evaluation

Okuya Kenichi (410027)

School of Information Science,
Japan Advanced Institute of Science and Technology

February 9, 2006

Keywords: loop-invariant, lazy evaluation, compiler, optimization, lambda calculas.

Loop-invariant elimination is one of the optimization executed by many imperative function compilers. It is a conversion which substitute a calculation without change of the value in a loop with a value which binds the caluculation out of a loop. A processing time of a program can be reduced by reducing number of operations in a loop. Purpose in this study is Improvement of speed by implement this optimization in a functional language.

A loop is expressed with a recursive function in a functional language. Branch called a pattern match is used often by the functional language, and branch takes place complexity. As a premise of optimization, the meaning of the program before and behind optimization is not allowed to change. Therefore, in simple conversion, the loop-invariant elimination without change the execution order of a program is unrealizable. Although simple loop-invariant elimination is realized by using middle expression called CPS, however, application of this method is difficult when branch takes place complexity. In order to solve this problem by simple conversion, lazy evaluation is introduced in this study.

A functional language can be divided into a strict and non-strict about an evaluation order. In strict languages, such as Standard ML, After an argument is evaluated first, the value is binded and a function is evaluated.

This is called strict evaluation. In non-strict languages, such as Haskell, evaluation of an argument is delayed and the pointer to an argument is passed to a function, and it is evaluated when needed. This is called lazy evaluation.

When binding a loop-invariant expression out of a loop using this lazy evaluation, at that time, it binds by delaying evaluation of a expression, without evaluating a expression. And I thought that loop-invariant elimination was realizable by simple conversion, controlling useless operation by evaluating, when actually using a value. And I thought that the loop-invariant elimination which controlled useless operation by simple conversion was realizable by evaluating when actually using value.

However, in order to introduce lazy evaluation into a strict functional language, it is necessary to add the evaluation mechanism for judging that the delayed expression is not evaluated yet or whether it is evaluated. Therefore, I thought that some overheads exist in evaluation of the delayed expression.

In this study, the loop-invariant elimination algorithm using the lazy evaluation based on the above theory was built. Using this algorithm in a strict functional language, the loop-invariant elimination by simple conversion is realizable. And loop-invariant elimination optimization based on this algorithm, I imprement of this optimization to SMLsharp compiler, which is an extension of Standard ML and Ohori and others developed. Lazy evaluation was realized by using a Data type, Closure, and Ref type. And it evaluated using a created test program by improvement compiler.

In the result, loop-invariant elimination was able to be applied by using lazy evaluation in case of a loop-invariant expression was in branch in the loop difficult in a conventional method. Although it resulted in becoming late rather compared with the case where optimization is not applied when eliminate light expression, such as primitive operation. However, when to some extent heavy expression, such as recursive funcion, could be elimination as a loop-invariant expression, it has confirmed that optimization worked effectively.

I thought cause of becoming slow is by mechanism of lazy evaluation. In this study, it is Closure generation and evaluation overhead.

As a future subject, First, judgng the weight of processing of a loop-

invariant expression beforehand, even if it takes an overhead into consideration, application loop-invariant elimination only about the case of being effective is mentioned. Although it can consider as the optimization which becomes early certainly if such processing is performed, I thought that it is difficult to judge the weight of processing correctly in the stage of compiling a program.

Second, Reducing an overhead includes a lazy evaluation mechanism in a processor is mentioned. Although the mechanism of lazy evaluation was realized by implimenting as a function in this study, Therefore, whenever it processes, the overhead of functional application was generated. The speed of lazy evaluation is improved by supporting this native by the abstract machine, and I thought possible to consider as more practical optimization.