Inline Expansion for Functional Languages

Koji Sato (310046)

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

February 10, 2005

Keywords: inline expansion, functional languages, closure conversion.

Inline expansion, which replaces a function call by the function body, is an optimization executed by many compilers. Inline expansion has two important benefits. First, it eliminates function call overhead. This overhead includes the cost of passing arguments, saving and restoring registers, and updating stack information. Second, it creates new program code that can be specialized. Since it creates a separate copy of a function body at a call site, the body can be specialized to the call site.

Since functional languages including ML have free variables in function body, simple inline expansion may change the meaning of the program. Therefore, a free variable in the function has to be able to be referred to correctly in the inlined code. Name capture is a famous problem that a free variable is not referred to correctly in the inlined code. This problem is caused by inserting a new declaration with the same name between the declaration of inlining function and the use of a free variable. It can happen when substituting a variable by its definition, since the definition might refer to variables that are redefined before the variable reference. A solution for this problem is to rename every bound variable so that every bound variable is unique in the whole program. However, all the free variables cannot be necessarily referred to correctly only by the technique.

Higher-order functions can return the function as a result of the function call. For example consider the expression: `let val f = g M in f N end` This is a code in which a certain function is returned by applying function g to the argument M, the function is bound to the variable f, and the argument N is applied to the function f. Even if we know the code of the function that is bound to the variable f, it is impossible to refer to the free variable in the function application `f N` if the free variable exists in the code and the free variable is defined in the function g. Because the free variable can be referred only in function g. Therefore, when executing inline expansion in this case, it may change the meaning of the program because a free variable cannot be referred to correctly. That is, it is not allowed to execute inline expansion directly in this case.

To solve this problem, we propose to execute inline expansion after closure conversion. An important difference of the code before and after closure conversion is as follows. By doing closure conversion, a function is replaced by a closure that is a tuple of the function code and the environment. And, all free variables in the function are replaced with the reference from the environment. Therefore, it is the same meaning to be able to refer to a environment correctly, and to be able to refer to a free variable correctly. Closure is a tuple of the function code and the environment when executing program. Closure is not a special data structure. Closure is considered to be a tuple. Taking out the second element of Closure is same as taking out the environment. Therefore, when the environment of Closure cannot be referred to directly, the environment can be referred to by taking the second element of Closure.

Based on the above idea, we propose the algorithm of inline expansion executed after closure conversion. Because it can be executed regardless of a free variable, This inline expansion can make more candidates of inline expansion. And this paper shows that the type of the code before and after inline expansion is the same. In addition, We have implemented the inline expansion and some optimization on IML compiler, which is an extension of Standard ML. For the programs in some benchmark suite, this inline expansion with some optimization improved execution times by an average of 40%, and reduced code size by an average of about 40%. We individually evaluated inline expansion that refers to the environment in the closure. As a result, it is a little less efficient compared with the method where the environment is referred to directly. Yet, the new inline expansion also improves execution times.