Efficient Closure Utilisation by Higher-Order Inheritance Analysis

Markus Mohnen
Lehrstuhl für Informatik II, Aachen University of Technology
Ahornstraße 55, 52056 Aachen, Germany
email: markusm@zeus.informatik.rwth-aachen.de

Abstract. Higher-order functions and the ability to create new functions at runtime, either by partial application or \(\lambda\)-abstraction of existing functions, are important features of functional languages. Since these runtime-created functions may outlive their creating functions, it is necessary to represent all functional parameters by closures in a memory area which is not affected by the termination of function calls, i.e. in a heap component. Such a closure contains the original function, which may be a closure again, and the bindings of the parameters used (by the partial application). It many cases, however, it is possible to avoid this expensive heap allocation and use the run-time stack or even statically allocated closures. Often a closure created for a partial application is used only locally and will never be part of the result of the creating function. In these cases, the closure may be allocated on the stack. This approach is feasible in eager and lazy languages. If we can additionally ensure that there will never be more than one incarnation of a partial application, a closure can be allocated statically. In order to use this optimisation in lazy languages, we have to perform a strictness analysis first. We develop an abstract interpretation for the detection of inheritance information which allows us to decide whether the heap cells of an argument may be propagated to the result of a function call (i.e. are part of the result). Furthermore we give a criterion for checking whether there will be always only one incarnation of a certain partial application during runtime. In order to increase efficiency, we show how the number of recomputations can be decreased by using only parts of the abstract domains. The worst case time complexity is essentially quadratic in the size of the program. We illustrate the method developed in this paper with several examples. Correctness of the analysis is considered, using a modified denotational semantics as reference point. The main goal of our work is to keep both the run-time and the compile-time overhead as small as possible.

1 Introduction

Higher-order functional languages are characterised by the ability to pass functions as parameters to functions and return functions as result of functions. Moreover, partial applications\(^1\) can be used to create new functions at runtime.

\(^1\) Alternatively, functions can be defined by \(\lambda\)-abstractions, which can be translated to partial applications by a transformation known as lambda lifting (see [Jon85, Hug83]).
These functions are built from existing functions by the transport of parameter bindings. Since such a newly created function can also be returned as result, it is possible that the binding of parameter variables will exist longer than the creating function call. As an extremal example, consider the following program, which uses partial applications of empty and update to represent an array of type int with index type int.

- \( \text{empty : int} \rightarrow \text{int} \)
  \( \text{empty}(i) = 0 \)

- \( \text{update : (int} \rightarrow \text{int}) \times \text{int} \times \text{int} \times \text{int} \rightarrow \text{int} \)
  \( \text{update}(\text{array,newi,newval,i}) = \begin{cases} \text{newval} & \text{if } (i==\text{newi}) \text{ then} \\ \text{array}(i) & \text{else} \end{cases} \)

- \( \text{upto : int} \rightarrow (\text{int} \rightarrow \text{int}) \)
  \( \text{upto}(n) = \begin{cases} \text{empty} & \text{if } (n==0) \text{ then} \\ \text{update}(\text{upto}(n-1),n,n) & \text{else} \end{cases} \)

The call \( \text{upto}(n) \) creates an array, where the \( i \)-th entry is \( i \), if \( 0 < i < n \), and 0 otherwise.

Until an entry is requested, for instance by \( (\text{upto}(n))(j) \), all \( n \) incarnations of update and the appropriate bindings of array, newi, and newval have to be remembered.

Therefore, implementations of functional languages must represent all arguments of functional type by closures which are usually allocated on a heap. A closure is a tuple of a function and a sequence of already given parameters for this function. Implicitly, closures contain information on how many arguments are needed to evaluate the function, how many arguments are present, and the values of those arguments which are present. For instance, the evaluation of \( \text{upto}(4) \) creates five closures cells (see Fig. 1).

Although this implementation technique is necessary for the general case, there are situations where we can do better. It can often be observed that the lifetime of a partial application is restricted to the execution of the function call, which created the partial application. Instead of creating a heap cell for the representation of the partial application each time it is encountered during computation, we can use a cell created on the stack frame associated with the creating function call.

Consider the quicksort function, which can be defined by

- \( \text{quicksort : ListOfInt} \rightarrow \text{ListOfInt} \)
- \( \text{quicksort} (\text{Nil}) := \text{Nil} \)
- \( \text{quicksort}(\text{Cons}(a,L)) := \text{append} (\text{quicksort}(\text{filter}(<a,L)), \text{Cons}(a, \text{quicksort}(\text{filter}(>=a,L)))) \)

The partial applications \(<a\) and \(>=a\) are used as functional arguments for filter. Since filter, in contrast to update in the above example, does not inherit its