Comp 411
Principles of Programming Languages
Lecture 27
Lambda Lifting and Closure Elimination

Corky Cartwright

April 2, 2014

Lambda Lifting

If a program does not use closures in interesting ways, we can transform the program to a collection of top level function definitions without introducing heap operations.

Consider a program where functions (lambda-expressions) with free variables (which we will call *global* functions) are never passed as parameters, never stored in data structures and never returned as values. Then free local variables in lambda expressions (function definitions) are always in scope (unless shadowed) at each call site where the function is applied.

If we unshadow all program variables (rename variables to eliminate all shadowing), then we can convert each function definition containing free variables to global form by replacing each free variable by an additional parameter. Of course, we must pass the eliminated free variables as arguments at each call site, but this is straightforward.

Lambda Lifting cont.

The primary complication is the fact that free variables within function definitions may be bound to functions, so we must make sure that each function bound to such a variable is converted to a global function before it is introduced as an argument in a call to another globalized function. We can easily accomplish this by lifting functions in order of nesting level, outermost first. If two or more functions in a letrec are mutually recursive, we must lift them all simultaneously.

Once all function definitions have been globalized, we can move them to the top-level without affecting the meaning of the program.

Note: if closures are used in non-trivial ways (passed as parameters, stored in data structures, returned as results), then we must allocate data structures (closure representations to store the values of the free variables) on the heap and explicitly pass these data structures to eliminate the free variables in such closures and globalize them. In some cases, we can separately allocate each such variable on the heap, but in the general case we must create a closure object including the address of the closure code for each evaluation of a lambda-expression and we must invoke this closure object instead of calling a conventional (C or machine) function.

Hence, in writing high-level code corresponding to a low-level implementation of an interpreter, we either (i) avoid the non-trivial use of closures or (ii) we accept the fact that we must heap allocate closure objects and explicitly invoke these closure objects instead of calling conventional functions. Of course, calling a closure object can be implemented as an indirect function call that passes the address of the closure object as an extra argument to the closure code.

Expressing CPSed Code in Machine Language

CPSed code contains many lambda-expressions. They appear either on the right hand side of let bindings or as arguments in function calls.

If we need to express a CPSed program in machine language, we need a good representation for lambda-expressions. Let's assume that our original program is free of non-trivial closures and that we perform lambda lifting (possibly including the heap allocation of some variables [a minor liberalization]) before CPSing the code.

- How do we represent these lambda-expressions in C/machine language. There are two choices:

- Make each lambda-expression a top level function and use raw function pointers (as in C) to represent functions as values. No lambda expression closes over local variables, so no environment is needed. All bindings are either global or local to a function invoked by a tail-call.

- Closure elimination which we explain on the next slide.

Closure Elimination

- Convert the local variable references in each lambda-expression to references to an arguments array.
- Associate ascending integer indices 0, 1, … with lambda-expressions and embed all of them in a single case (switch) statement. This case statement can be either (I) the body of a huge binary tail-calling procedure that switches on its argument or (ii) part of the main program. (In the main program version, the case statement can be replaced by explicit labels and function invocation by goto's.)
- Applications of lambda-expressions simply call the huge procedure with the index corresponding to the lambda-expression and the arguments array for the call. (In the main program version, each call initializes the arguments array (a global variable) to the appropriate contents and jumps to the appropriate lambda-body.
- Note that this scheme can easily be generalized to handle the general form of closures where closure representations are allocated on the heap. Each closure representation must include the index or address of the corresponding block of code as well as the binding of the free variables (which may be pointers).

Examples and Discussion

The narrative in the course notes shows how to perform closure elimination in our LC interpreter.

Observation: the details of the translation (which vary depending on the specific implementation language and low-level design choices made by the implementor) are not important.

What is important: in any program the number of lambda-expressions embedded in a program is finite. If all local variable references are removed from lambda-expressions and all lambda-expressions are called in tail position, then lambda-bodies simply become blocks of code and lambda-invocations simply become jumps to the appropriate code blocks!

Note that any C program where all function calls are in tail position can be translated to a single main program (assuming only one entry point is needed) where each function become a block of code and tail calls are translated to goto's. The local variables in each function need to be converted to a generic array of parameters. This translation converts option 1 from the last slide to option 2!

In C, function pointers also support general stack-based function invocation but we don't need this capability. We only need tail calls. Of course, this translation will not be space-efficient unless the C compiler performs tail-call optimization. So it may be advantageous to perform closure elimination.