Comp 411
Principles of Programming Languages
Lecture 31 (Supplemental)
Lambda Lifting and Closure Elimination

Corky Cartwright

April 28, 2021

Eliminating the Stack from Language Runtimes Part I

In Assignment 7, we observe that CPSed programs converted to SDAST form can be executed without the benefit of an environment stack (or environment list in the heap). This claim is correct, but it requires a more complex formulation of the activation records that we presented for the Algol 60 runtime. In the more complex formulation, an activation record consists of a prelude consisting of blocks of lexically enclosing `let/letrec` bindings optionally followed by the bindings for an applied λ-abstraction corresponding to a program-defined function. The body of this function is then reduced, potentially adding a sequel of binding blocks to the activation record corresponding to `let/letrec` blocks encountered in reducing the function body.

Given this recasting of the definition of an activation record, the current state of the evaluation of a CPSed program is expressed using a single activation record (a prelude of enclosing `let/letrec` binding blocks optionally followed by a the bindings for an applied program-defined function (called the function parameter bindings) optionally followed by a sequel of `let/letrec` blocks enclosing the reduced function body). If the reduced body of this program defined function is a tail-call invoking another program-defined function (which must be visible in the binding blocks in the activation record), then the tail of binding blocks in the current activation record back to (but not including) the block containing the binding of the called function is discarded (necessarily discarding the parameter bindings for the calling function) forming a new prelude and a new parameter binding block for the called function is added to this prelude.

Note that a compiler has complete knowledge of the layout of the activation record for every variable access. Hence, it can load the value of any variable in the activation record in a single indexed load instruction.

At all times, the bindings in the blocks in the activation record are the only variables that are visible to the code being reduced (evaluated). There is no stack of pending computations since the program has been CPSed.

Eliminating the Stack from Language Runtimes Part II

Another way to describe essentially the same process without changing the form of activation records from the Algol 60 runtime (other than eliminating dynamic links) is to transform a CPSed program so that

- All variable names are unique;
- Every program-defined function is moved to the top-level where all variables occurring free in a program defined function are converted to additional parameters, implying that the caller must have access to the values that would have been bound in a closure. This process is called *λ-lifting*.
- Every `let/letrec` block is eliminated by absorbing it into the enclosing λ-abstraction or program top-level.
- All top-level bindings are placed in a special "static" data area separate from both the stack and the heap. These static bindings include all of the binding that are lifted to top-level by *λ-lifting*.
- The program state is defined by contents of the static data area and the current activation record which is roughly equivalent to the parameter bindings and sequel of `let/letrec` blocks following the parameter binding in the expanded format activation record described previously. The static data area roughly corresponds to the prelude of the current activation record in the expanded format but the latter changes depending on the current function being evaluated while the former does not.
- The details of the *λ-lifting* process are tedious and delicate just like the transformation rules for CPS conversion. But the idea should be clear since it is a normal part of the program design process in writing code in a fully lexically scoped language.
- There are two important advantages of this approach. First, it directly applies to programs that have not been CPSed. Second, it meshes well with using C as an intermediate language, although exact garbage collection in this context looks painful.

The Mechanics of λ-Lifting

The λ-lifting transformation (with the accompanying `let/letrec` elimination process) is conceptually straightforward but messy to write down precisely, so the remainder of this lecture will focus on a the specific example: translating a low-level interpreter written in Scheme/Racket for a restrictive language mostly functional language J consisting a subset of Jam into C code. Given the restrictions imposed by J, we can eliminate the need for a heap! But the restrictions imposed by J are stylistically awkward in the context of most real applications. So we show how to eliminate the restrictions on J at the cost of modest reliance on a heap.

If a program does not use closures in interesting ways, namely, it only uses λ-abstractions as rators or as the RHSs of variable definitions, we can transform the program to a collection of top-level function definitions as in C without using heap operations *essentially because we never need to fetch the value of a variable that is free in a function/procedure body*.

This observation should not be surprising since it underlies the Algol 60 runtime which has no heap. Of course, Algol supports the use of lexically visible free variables in function/procedure bodies by maintaining a static chain of activation records in the central stack. In Algol 60, functions can only passed downward into contexts for which the closing activation record for the function (fetehed from the static chain when it is passed) is guaranteed to exist.

λ-lifting in the Absence of Heap Operations

Consider a Jam program where all functions (λ-abstractions) *with free variables* are never passed as parameters (a more stringent restriction than in Algol 60), never stored in data structures and never returned as values. We will use the name J to refer to the subset of Jam meeting this restriction. We will use the term *global functions* to refer to such λ-abstractions. Hence, J only supports the definition of program functions that are *global*. Note that many common uses of the well-known `map` library function pass function parameters that not *global*. In Jam (and hence J), the free variables in global functions are always in scope (unless shadowed) at each call site where the function is applied since each call site falls within the same scope as the function definition.

If we rename all program variables to eliminate shadowing, then we can convert each global function definition containing free variables to *top-level form* (expressible as a top level function definition in Jamb) by replacing each free variable by an additional parameter. Of course, we must pass the replaced free variables as arguments at each call site, but this is straightforward. The next slide explains the only technical complication.

Technical Complication

The technical complication is the fact that a free variable *f* within a global function definition *F* may be bound in some use to another global function *G*. Hence, we must make sure that each such function *G* is converted to top-level form (eliminating its free variables and possibly adding more parameters) as we process function bodies where *G* is passed as an argument in a call (as a top-level function) to the globalized version of *F*. We can do this by lifting functions in order of nesting level outermost first (forcing *G* to be processed before or at the same time as *F*). Within a `letrec`, all function definitions (which may be mutually recursive) are lifted simultaneously. As a result, a function is only lifted when all of the functions to which it refers (free variables and added parameter bindings) are already defined at *top-level* or defined within the group of bindings being lifted.

Once all function definitions have been converted to top-level form, we can execute a such a program without performing any heap operations.

Supporting First-Class Functions

When closures are used in non-trivial ways (passed as parameters when they include free variable references, stored in data structures, returned as results), then the global function restriction is violated. In the general case, we must allocate closure representations (which store both the function definition and the values of the free variables in the function body) in the heap and explicitly pass these data structures to encapsulate the bindings of the free variables in such closures and globalize (convert all references to functions to their global equivalents) the function body and the variable binding expressions. In the general case we must create a closure representation including the address of the globalized closure code for each evaluation of a λ-abstraction and we must apply this closure representation instead of calling a conventional top-level function.

Hence, in efficiently implementing a language like Jam supporting functions as data, we must either (*i*) restrict the use of functions as data values (as in J) or (*ii*) accept the fact that we must heap allocate closure representations and explicitly invoke these closure objects instead of calling conventional functions, which in general requires creating local bindings that the closure can access (which we could do by absorbing these binding as extra function parameters).

Expressing Functional Code in C/Machine Language

Functional code contains many λ-abstractions. They appear either on the right hand side of let bindings, as rators in applications, as arguments in function calls, or as the bodies of functions/procedures

If we need to express a functional program in C/machine language, we need a simple representation for λ-abstractions. In the simple case when the original program is free of non-trivial closures, we can obviously perform λ-lifting as previously described, reducing all functions to top-level C-functions, which are conveniently represented as pointers. Moreover, we can support λ-abstractions that are not global in the λ-lifting process if we have a heap to store closure representations of the evaluations of non-global functions. After performing λ-lifting, we can collapse nested let/letrec constructions to the top of the function/procedure/main-program in which they are enclosed. At this point, all bindings can be absorbed into either the top-level static data area or the activation record of the enclosing function. Finally, we may or may not CPS the code. By CPSing the code we can eliminate the environment stack and expose intermediate results otherwise buried in local variables/temporaries in the stack.

How do we represent the λ-abstractions in λ-lifted code in C/machine language without a static chain? C has no procedure nesting and no static chain because it is very close to the machine. Two choices are shown on the next slide.

Representing λ-Abstractions Without Environment Static Chains and (perhaps) Heap-Allocated Closures

- We can use C-style function pointers to represent function values if we make each λ-abstraction a top-level function (no free local variables). Since these trivial λ-abstractions cannot contain free variables, no environment is needed to represent the corresponding closures; they are simply function pointers. All program bindings are either global (at known addresses in the static data area) or local to a function. If the code has been CPSed, then every function is invoked by a tail-call, eliminating the need for a control stack (using either elaborated activation records or λ-lifting and). *If our language runtime includes a heap, we can support functions that are not global* by heap-allocating the representations on non-global λ-abstractions that include the bindings of for free variables. Since these variables are simply heap locations, we still do not need a static chain. Supporting a memory-safe heap (no dangling pointers) requires heap storage management in the form of reference-counting or conservative/exact garbage collection. If our language implementation is a compiler that generates machine code, it can support "exact" garbage collector by performing the detailed bookkeeping necessary to determine which memory cells contain pointers.
- Alternatively, we can perform closure elimination, a hack which we explain on the next slide. This option is more expensive (even with tail-call optimization) but does not require explicit function pointers (the preferred approach IMO).

Closure Elimination

- Convert the local variable references in each λ-abstraction to references to an arguments array.
- Associate ascending integer indices 0, 1, … with λ-abstractions 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 λ-abstractions simply call the huge procedure with the index corresponding to the λ-abstraction and the arguments array for the call. If the code has been CPSed, the arguments array is stored in the current (and only) activation record that is re-used in each function call. A function call simply initializes the arguments array to the appropriate contents and jumps to the appropriate λ-body.
- Note that this scheme works for general closures where closure representations are allocated on the heap. Each closure representation (encapsulating the bindings of the free variables in the λ-body) 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 into the heap).
- This approach assumes the heap incorporates some form of garbage-collection.

Note that this representation is a C-language hack. If the implementation is expressed in machine code (the norm for compiled code), ordinary pointers are simpler and more efficient.