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

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). To fully realize this claim we need to do some additional program transformation because the low-level interpreter in Assignment 7 still creates stacks of environment (activation) records to implement (possibly nested) **`let/letrec`** constructions, which can occur in CPSed code. We can completely eliminate all of these **`let/letrec`** constructions by performing some additional program transformations.

- First, we lift all **`let/letrec`** function/procedure bindings to the top-level, using a transformation called *λ-lifting* that program designers often perform during program development. (We assume all variables have been renamed to eliminate shadowing.)
- After performing this transformation, the language implementation (interpreter or compiler) can collapse all of the remaining **`let/letrec`** constructions (which only contain ordinary [non-function] bindings) to a single **`letrec`** at the top of each enclosing function/procedure or a collection top-level bindings (expressed by **`define`** in Scheme/Racket) with default RHSs for the program body.
- Then the top-level **`letrec`** constructions within function/procedure definitions can be absorbed by the enclosing λ-abstraction as additional parameters with default bindings. (The Java compiler performs this absorption process to support all bindings local to a method in the activation record for the method.) Similarly, the generated top-level program definitions along with all pre-existing top-level program definitions can be placed in a "static data area" (historically called the BSS in Unix load modules) to hold all top-level program bindings. This static block of binding cells is fixed for the entire program execution.

Eliminating the Stack from Language Runtimes cont.

After the transformation described in the preceding slides

- All program computation is expressed at the top-level, implying all tail-calls can be implemented by jumps.
- In CPSed code, there is always only one activation record. Each tail-call overwrites the current activation record with the new one—completely eliminating the need for a stack of activation records. But note that we have eliminated the stack at significant cost. The conversion to CPS creates a closure in the heap for every call on a program defined function.

This account also hand-waves one important issue. Do the RHSs of the top level program bindings require general computation?

- If the implemented language specifies a strict (call-by-value) semantics for RHSs, such computations are performed by the language translator (interpreter/compiler) before the program is actually executed, which may cause the language translation process (compilation, interpreter pre-processing) to diverge! This phenomenon can actually happen in DrRacket. In Racket, the RHS of a top-level definition must terminate for compilation to succeed. Of course, if a RHS is a λ-abstraction, it trivially terminates.
- In languages like Haskell where the evaluation of the RHSs of bindings is lazy (demand-driven), the language implementation generally creates suspensions (thunks) in the heap to defer such computations until demanded. But, of course, the language implementation for such a lazy language can perform some static analysis to detect all simple terminating RHSs and perform them at translation time as an optimization.

Eliminating the Stack from Language Runtimes

The clerical details of these transformations are conceptually straightforward but very 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 L similar to Jam into C code. Given the restrictions imposed by L, we can eliminate the need for a heap! But the restrictions are crippling in the context of most real applications. So we show how to eliminate the restrictions on L at the cost of relying on a heap.

If a program does not use closures in interesting ways, namely, it only uses lambda-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 introducing 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 stack of activation records supporting a restricted form of closure that prevents closures (functions) from being first-class values (usable in all of the same contexts as conventional machine-level values).

λ-lifting Without Using Heap Operations

Consider a program where all functions (λ-abstractions) *with free variables* are never passed as parameters (a more stringent restriction than Algol 60 places on function-passing since it supports “downward” closures), never stored in data structures and never returned as values. Our restricted language L mandates this constraint. We will use the term *global functions* to refer to such λ-abstractions. Hence, L mandates that all program functions are *global*. In a lexically scoped programming language (like Jam, Scheme or C), 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 C) 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 function definition *F* may be bound 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) before we process any function bodies where it 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 (taking into account that the RHSs of the definitions in a raw **`let`** are *not* nested within the new scope created by the **`let`**) outermost first. 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), excluding mutual references in a **`letrec`**, are defined at *top-level* or defined within the group of binding 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, stored in data structures, returned as results), then the global function restriction is violated. In the general case, we must allocate data structures (closure representations that store the values of the free variables in the closure body) on the heap and explicitly pass these data structures to encapsulate the free variables in such closures and globalize them. In some special cases, we can separately allocate each such variable on the heap and directly access it in the function body, but in the general case we must create a closure object including the address of the closure code for each evaluation of a λ-abstraction and we must invoke this closure object instead of calling a conventional (C or machine) function.

Hence, in writing an interpreter in a high-level language that we want to map into efficient low-level code, we either (*i*) eschew 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 a general closure dispatch procedure.

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 the function to a top-level C-function, which is a pointer value. Moreover, we can λ-abstractions that are not global in the λ-lifting process if we have a heap to store closure representations and the free variables in 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 where there is no static chain? C is crude in this regard 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 (assuming the λ-lifting transformations described earlier in this lecture). *If our language runtime includes a heap*, *we can support functions that are not global* by heap-allocating the free variables appearing in non-global λ-abstractions. 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 generating machine code, it can support "exact" garbage collector by performing the detailed bookkeeping necessary to determine which memory cells contain pointers.
- 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 representation 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.