Comp 411
Principles of Programming Languages
Lecture 17
Efficient Run-time Environment Representations

Corky Cartwright

March 3-5, 2021

Intuitive Overview

- Compiled code (machine code corresponding to high-level language code) is typically much faster than interpreted code because the executed operations can be specialized (customized) to the actual code being executed. In addition, interpreter are typically written in high-level languages that add overhead and constrain how the data values are represented and the interpreted operations are implemented.
- For fast execution of source programs, they almost always need to be compiled.
- Compiled code executes in the context of a *run-time*, a protocol for how each compiled source program unit couples with the compiled code for other program units including a representation for the execution environment. (Compiled code relies on an environment just like meta-interpreters do, but its representation is optimized for efficiency at machine level.
- The first comprehensive compiler run-time was the run-time created for Algol 60. The committee who designed this run-time was brilliant. Most contemporary compiler run-times are extensions/elaborations of this design. Is this surprising? Yes. Is it unprecedented? No. Carl Friedrich Gauss invented the FFT algorithm in 1805, wrote it up as a cute trick for solving an otherwise intractable computational problem (which he did by hand) in his research notes, but never published it. For Gauss, it was a perhaps a "trifle". Fourier did not do his work until about 1820 although Francois Budan published much of the same work in 1807-1811.

Jeff Erickson's online book on Algorithms is a great source of misattribution.

Stack-Based Environment Representations

- In Algol-like languages, the environments that exist at any point during a computation can be collectively represented using a *stack* that is an elaboration of the *control stack* supporting procedure calls in most modern processors.
- Algol-like languages are almost always *compiled* to machine code rather than interpreted like Jam. Nevertheless, the compiled code must perform the same operations on program data structures as interpreted code does. A compiler typically performs far more program analysis than an interpreter enabling it to pre-compute quantities that are determined at run-time by an interpreter.
- Almost all modern machines provide a control stack to store the return addresses of procedure calls. In addition, other context information (such as saved register values) is typically saved with the return address in a *frame* on the control stack. To return, the called procedure pops the current (top) frame off the stack restoring the saved context information and jumps to the specified return address. Popping the stack frame for a called procedure restores the stack to the form it had before the call (but the bindings of some variables stored in the stack may have changed).
- Many machines also pass procedure/method argument values in the stack. Another possible convention is to pass arguments (up to some bound) in registers.
- The result returned by a procedure is typically returned in a register because the stack frame associated with the call is deallocated on return.

Lexical Scope in Stack Environments

In a stack-based implementation of a lexically-scoped language, a new environment is constructed (`extend-env` in our LC interpreter) to evaluate the body of a `let` or `lambda`-application by allocating a new frame called an *activation record* (or *stack frame*) on the control stack. The activation record contains:

- the new variable binding values introduced by the `let` or `lambda`,
- a pointer called the *static link* pointing back to the rest of the environment(a linked list of activation records),
- a pointer called the *dynamic link* to the preceding activation record,
- the return address (address of the next instruction in the code block that invoked the `let` or `lambda`-application), and
- any register/context values that need to be saved for restoration on return from the let or lambda.

In this representation, an environment consists of a linked list of activation records called the *static chain* where the *static link* serves as the *link* field of the list. The first record in the gives the bindings at static distance 0 and so forth. The length of this list is simply the lexical nesting level of the body of the `let` or `lambda`-application being evaluated. The compiler knows the exact structure of the *static chain* and the offset of each binding value in each activation record.

Extensions of Stack Environments

For let invocations (regardless of whether let is recursive)

```
(let ([x1 e1] ... [xn en]) E)
```

and raw lambda applications

```
((lambda (x1 ... xn) E) e1 ... en)
```

the static link and dynamic link in the new activation record both point to the same place, namely the preceding activation record on the stack (the activation record for the enclosing let form or lambda application).

For a function application

```
(f e1 ... en)
```

where `f` is the name of a declared function (in scope), the static link in the new activation points to the activation record in the static chain corresponding the static distance between the application site and the definition of `f`. Hence, this activation record contains the bindings of the variables defined in the same lexical unit as `f`. For a simple recursive function call (*e.g.*, the recursive call in the usual definition of factorial), this static link is identical to the static link in the calling activation record (the preceding activation record on the stack).

Closure Representation in Stack Environments

- Recall the definition of a *closure*: a pair containing an executable unit of code and an environment (or relevant part of it).
- How can we represent such a pair given the environment is a linked list of activation records?
  - Environments are represented by pointers to activation records. The represented environment is the linked list of activation records (determined by the *static link* fields) specified by the pointer. Each such record contains a sequence of binding values.
  - A closure is a pair consisting of the address of the routine (procedure) to be executed and the corresponding environment (pointer). But this implementation has limits.
  - When a closure is invoked the embedded environment frame is typically not the top (most recently created) stack frame. This environment pointer is copied into the static link of the new frame allocated for the closure invocation.

Alternate Light-weight Closure Representation

- Observation: the code in a closure can be executed without the entire closure environment. Only the bindings of the referenced variables matter!
- Alternate representation of closures: code plus a pruned environment, typically a few bindings, which can be embedded in the closure object! Of course the code must be tailored to the fact the binding values are located in the closure object rather than a standard environment. The compiler can generate code where each free variable simply corresponds to an offset in the closure object/record.

Run times for Modern Stack-based Languages

- Nearly all practical languages are stack-based following the Algol 60 run-time. Some ML implementations are not (all activation records in the heap) but it is a stretch to claim that they are practical. Nearly all modern stack-based languages also include a heap which is simply a data area where the lifetimes of data values do not necessarily obey a stack discipline. Note that the the lifetime of a closure over the stack cannot exceed the lifetime of the activation record at the head of its environment, which is huge limitation. As a result, practical languages have nearly always provided such a separate static data area (perhaps in ugly, low-level form), *e.g.*, Fortran COMMON blocks. All data values created by "new" operations (*malloc* in C, constructors in Scheme) are allocated in the heap.
- Data values that are directly stored in local variables are not heap-allocated, *unless they appear free in a closure*. Placing such variables in the heap is a ***critically important idea introduced by Guy Steele in the Rabbit compiler for Scheme***. This "light-weight" representation of closures must store the addresses of all environment variables that may be accessed in the closure object/record because the conventional environment pointer does not work after the corresponding stack frame (activation record) has been deallocated. Using this closure representation, ***there is no restriction on the use of closures as data values***. Functions are truly first-class. Guy Steele's **important** *hack* (in the good sense of the word) is yet another example of David Wheeler's maxim: "All problems in computer science can be solved by another level of indirection."

Run times for Modern Stack-based Languages II

- Why does Java require free variables in closures to be `final`? So that they can safely be copied into a light-weight closure representation (inner class instance)! If they were mutable, copying them would break the semantics of assignment!
- The original implementation of inner classes heap-allocated (not necessarily `final`) variables (as arrays of size 1) that appear free in some closure as array of size 1 and stored pointers to the binding cells (the array objects) in the inner class instance instead of the actual binding values. As a result, inner class instances could close over mutable cells in addition to `final` variables (values). But it is easy to work around the restriction to `final` variables: simply use `final` arrays, because the cells (elements) of an array are mutable! (Aside: what does it mean for a variable of array (or any reference [object]) type to be `final`?)