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

Corky Cartwright

Swarat Chaudhuri

October 3, 2011

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

Corky Cartwright

Swarat Chaudhuri

October 3, 2011

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.
- 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, restores 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 argument values to procedures 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* on the control stack. The activation record contains:
  - the new variable bindings 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 where the *static link* serves as the *link* field. The first record in the sequence contains the local bindings (static distance 0), the second record gives the bindings at static distance 1, 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.

Environment Extension in 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

- What is a closure? A pair containing code and an environment.
- 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.
  - A closure is a pair consisting of the address of the routine (procedure) to be executed and the corresponding environment (pointer).
  - 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.

Runtimes for Modern Stack-based Languages

- Nearly all practical languages are stack-based. Some ML implementations are not 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.
- Historically, practical languages have always provided such a data area (perhaps in ugly, low-level form), e.g., Fortran COMMON blocks.
- All data values created by "new" operations are allocated in the heap. Data values that are directly stored in local variables are not, *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. Why does Java require free variables in closures to be final? So that they can safely be copied into the closure (inner class) instance!

Runtimes for Modern Stack-based Languages

- Heap storage can be managed manually or automatically.
- Manual management (C/C++) can be extremely painful and error-prone. Even manual management relies on automatic allocation ("new" operations). Is "new" trivial? Once the virgin heap space is exhausted, no!
- Fully automatic management ("garbage collection") requires run-time bookkeeping (which is cheap in a well-designed system). The manager (garbage collector) must be able to determine (or conservatively estimate) which objects in the heap are still accessible from the stack and the global data area. When free space is exhausted, GC is triggered.