Comp 411
Principles of Programming Languages
Lecture 18
Run-time Environment Representations II

Corky Cartwright

March 2, 2020

Review

- In Algol-like languages, the collection of environments that exist at any point during a computation is embedded in the machine *control stack* supporting (recursive) procedure calls. When the frames of the control stack are used in this way, they are called *activation records*.
- In each activation record, a pointer called the *static link* points to the environment parent of the record. Similarly, a pointer called the *dynamic link* points to the preceding stack frame (activation record) to which control will return when the current computation [conducted using the current activation record] completes. The static link is used for looking up non-local bindings (of free variables in the current `let` or `lambda`-abstraction) in the environment.
- The dynamic link is used to return control from the current "procedure" to its caller (whose local variables may not be accessible from the current frame).

Example I

Consider the following Scheme program to reverse a list:

```
(define rev (lambda (l)
  (letrec
    [(revhelp ; :=
       (lambda (tl acc)
         (if (empty? tl) acc
             (revhelp (rest tl) (cons (first tl) acc)))))]
    (revhelp l empty))))
```

The Pidgin Algol equivalent (extended to include functional lists as built-in type:

```
List rev(l: List) = {
  { List revhelp(tl: List, acc: List) = {
      if empty?(tl) then acc else revhelp(rest(tl), cons(first(tl), acc)) };
    revhelp(l, empty)
  }}
```

What happens when `(rev '(0 1))` is called?

- The top level call on **`rev`** allocates activation record (AR) #1 with null static and dynamic links and a slot for **`l`** (el) initialized to **`'(0 1)`**.
- The body of **`rev`** (executing in AR #1) allocates AR #2 for the **`letrec`** with static and dynamic links pointing to preceding activation record and a slot for **`revhelp`** initialized to the closure for its definition.
- The body of **`revhelp`** allocates AR #3 record for the recursive call on **`revhelp`** with static link taken from closure binding of **`revhelp`** (in AR #2) and dynamic link pointing to preceding activation record.

- Since `l` is not empty, body of `revhelp` allocates AR #4 for the recursive call on `revhelp` with static link taken from closure binding of `revhelp`, dynamic link …, and slots for `tl` and `acc` initialized to `'(1)` and `'(0)`, respectively.
- Since `l` is not empty, body of `revhelp` allocates AR #5 record for recursive call on `revhelp` with static link taken from closure binding of `revhelp`, dynamic link …, and slots for `tl` and `acc` initialized to `'()` and `'(1 0)`, respectively.
- Since `l` is empty, body of `revhelp` in context of AR #5 returns the value `'(1 0)`, popping AR #5 off the stack.
- The pending evaluation in AR #4 returns the value `'(1 0)`, popping AR #4.
- The pending evaluation in AR #3 returns the value `'(1 0)`, popping AR #3.
- The pending evaluation in AR #2 returns the value `'(1 0)`, popping AR #2.
- The pending evaluation in AR #1 returns the value `'(1 0)`, popping AR #1.

Notes:

1. The last four steps are trivial because they are returns from tail calls.
2. The dynamic link is *always* set to point to the preceding AR.
3. Algol 60 was designed so that the ARs could be stack allocated (and deallocated). Function values are not "first-class".
4. Guy Steele's heap allocation "hack" (to be covered later) extends the Algol stack allocation runtime to support to languages supporting first-class functions/procedures.
5. In Java, inner classes enable the nesting of scopes as in Algol; the static chain is formed by embedding hidden parent instance pointers in the inner class objects. In addition all non-local variables accessed in an inner class must be final so that they can be copied into the inner class instances.

Example II

Consider the following Scheme program to reverse a list:

```
(define lookup (lambda (sym env)
  (letrec
    [(lookup-help
       (lambda (env)
         (cond [(empty? env) null]
               [(eq? sym (pair-var (first env))
                (pair-val (first env))]
               [else (lookup-help (rest env) tl)]))]
    (lookup-help env))))
```

Let's trace the evaluation of `(lookup 'a (cons (make-pair 'a 5) null))`

- The top-level call on `lookup` allocates AR #1 with null static link and slots for `sym` and `env` initialized to `'a` and `'(['a 5])`.
- The body of `lookup` (executing in AR #1) allocates AR #2 for the block with the static link pointing to AR #1 and a slot for `lookup-help` initialized to the closure for its definition.
- The body of the block executing in AR #2 allocates AR #3 for the call on `lookup-help` with the static link extracted from the closure bound to `lookup-help` and a slot for `env` initialized to `'(['a 5])` (the value of `env` in the environment determined by the static link of AR #2).
- The body of `lookup-help` executing in AR #3 looks at `env` and finds a match for `sym` (found in the static chain in AR #1) in the first pair, namely `['a 5]` and

Exceptions

Exceptions were not included in Algol 60 or most of its successors (Pascal, Algol W, C). But the Algol 60 run-time stack can easily handle the Java `try/catch` construct.

How does exception handling work? Activation records must include a `catch` table for the active `catch` (assuming one exists) listing the caught exception classes (types) and their handlers (the bodies of the `catch` clauses). (A `catch` is active if control is within the corresponding `try` block.) When an exception is thrown the executing code (interpreter or compiled code) searches back through the dynamic chain—popping exited frames off the stack—to find the first matching catch clause.