CS 7400 November 23-30, 2009
Dr. Wand Readings: EOPL Chapter 5 (more or less)

Lecture 9: Control Contexts and Continuations

Key Concepts:

Contexts
  Composition of
  Outside-in vs. Inside-out definitions of
Context or CK Machine
Continuations
Continuation-Passing Interpreter
CEK Machine
Expression and List Contexts

We now return to small-step semantics. Our implementation of small-step semantics is very inefficient, because at every step it reconstructs the entire expression and then starts looking for the redex again from the very top:

$$e_1 = E_1[l_1] \to E_1[r_1] = e_2 = E_2[l_2] \to E_2[r_2] = e_3 = E_3[l_3] \ldots$$

This total reconstruction is handy for tracing, but seems wasteful: the next redex is usually quite near to the last one.

So we'll next develop an implementation of small-step semantics, called the *CK machine* or *context machine*, that doesn't rebuild the entire expression. Instead, it rebuilds it incrementally, and as soon as it gets above the next redex, it searches down again.

To do this, it's handy to look a little closer at the algebra of reduction contexts. We'll start with just the arithmetic expressions. The contexts were defined by:

$$K ::= [\,] \mid -(K, e) \mid -(n, K)$$

A better way of thinking about this is to think about how the smallstep semantics *builds* these contexts. When we looked at continuations, we saw that continuations were always built *inside-out*. This is an important operation, so let's introduce some notation for it: define $K_1 \circ K_2$ by

$$K_1 \circ K_2 = K_1[K_2]$$

We'll call this *composition* of contexts. Now, $\circ$ is associative, since

$$K_1 \circ (K_2 \circ K_3) = K_1[(K_2 \circ K_3)] = K_1[(K_2[K_3])] = (K_1[K_2])[K_3] = (K_1 \circ K_2) \circ K_3$$

Furthermore $[\,]$ is a left and right identity for $\circ$, since $[\,] \circ E = E = E \circ [\,]$.

Remember, we are always working on some term e in some context K. As we did for continuations, let's consider the possible contexts K. We start out with the empty context $[\,]$. If we have an expression e in the hole of a context K, what could happen as we compute? It could be that e was $-(e_1, e_2)$, in which case we need to search *down* in e_1 to find a possible redex. We observe that

$$K[-(e_1, e_2)] = (K \circ -([\,], e_2))[e_1]$$

thus building the new context $K \circ -([\,], e_2)$. So in this case we should consider the expression e_1 in the new context $K \circ -([\,], e_2)$.

The other possibility is that e is a number n. In that case, we need to search *up* in the context K_to find the next redex. If K is empty, we are done:

$$K[n] = n$$

The other possibility is that K is of the form $K'[-([\,], e_2)]$ for some K'. In that case we observe

$$(K' \circ -([\,], e_2))[n] = (K' \circ -(n, [\,]))[e_2]$$

So we refocus on e_2 in the new context $(K' \circ -(n, [\,]))$. What if the expression is a number (let's call this one m) and the context is of this new form? Then

$$(K' \circ -(n, [\,]))[m] = K'[-(n, m)] \rightarrow K'[p]$$

So we take a step and then refocus on K' and p.

We can summarize this in the following grammar

$$\begin{array}{ll} F ::= -([\,], e) \mid -(n, [\,]) & \text{Frames} \\ K ::= [\,] \mid K[F] & \text{Reduction Contexts} \end{array}$$

Note that there is no recursion in the frames. All the recursion is in the definition of the contexts.

Note also that in the definition of the contexts, we've implicitly used the operation $K_1[K_2]$, which denotes the context obtained by inserting the context K_2 into the hole of K_1.

Since composition is associative, we can redefine the reduction contexts as any sequence of frames combined with composition:

$$E ::= F \circ \ldots \circ F \qquad \text{Reduction Contexts}$$

This also means we can define the reduction contexts either by left- or right- recursion:

$$\begin{array}{ll} K ::= [\,] \mid F \circ K & \text{Outside-in definition} \\ K ::= [\,] \mid K \circ F & \text{Inside-out definition} \end{array}$$

The first definition is the original, outside-in definition of contexts, and the second is the new definition. These two grammars define exactly the same set of contexts. We will use this handy fact continually in what follows.

This is just like what we did when we build continuations by functional composition, except that now we are dealing with their syntactic representations, rather their functional representations.

This idea of representing a focussed tree (a tree focussed on a subtree within it) is sometimes called a *zipper*, and is not limited to reduction contexts, etc. [Example]

[I am now switching back to E for contexts/continuations]

The context machine will have two registers, e and E, which range over expressions and contexts, respectively. The pair (e, E) is a representation of $E[e]$. The machine will push bits of expression from e to E and back again, as follows:

Arithmetic Expressions: The Context Machine

$$
\begin{array}{ll}
\langle -(e_1, e_2),\ E\rangle & \rightarrow \langle e_1,\ E \circ -([\,], e_2)\rangle \\
\langle n,\ E \circ -([\,], e_2)\rangle & \rightarrow \langle e_2,\ E \circ -(n, [\,])\rangle \\
\langle n,\ E \circ -(m, [\,])\rangle & \rightarrow \langle m - n,\ E\rangle \\
\langle n,\ [\,]\rangle & \rightarrow n
\end{array}
$$

Note that each transition $\langle e_1,\ E_1\rangle \rightarrow \langle e_2,\ E_2\rangle$ has the property that either $E_1[e_1] = E_2[e_2]$ or $E_1[e_1] \rightarrow E_2[e_2]$. (The third rule is the one that simulates the reduction).

So this machine is going to try to simulate a reduction. It starts with an arithmetic expression. It searches down using the first rule until it finds a number. When it finds a number it searches up in the tree by popping frames off the context:

- If the number is in the left subtree of a diff, start reducing the right subtree.
- If the number is in the right subtree of a diff, we must have already found the value of the left subtree, so we reduce and start continue searching.
- If the number is at the root (in the empty context), then it is the final answer.

Now let's extend this idea to our core language, which was defined by:

Core Language: Expression and Values

$$
\begin{array}{lll}
n & & \text{Integers} \\
x & & \text{Variables} \\
e ::= n \mid -(e_1, e_2) & & \\
\quad \mid x \mid (e_1\ e_2) \mid (\lambda x.e) & & \\
v ::= n \mid (\lambda x.e) & &
\end{array}
$$

$$
\begin{array}{lll}
E ::= [\,] & & \text{Reduction Contexts} \\
\quad \mid -(E, e) \mid -(v, E) & & \textsc{Cxt Arith} 1,2 \\
\quad \mid (E\ e) & & \textsc{Cxt Rator} \\
\quad \mid (v\ E) & & \textsc{Cxt Rand}
\end{array}
$$

RED DIFF
$-(n, m) \rightarrow p \quad$ where $p = n - m$

RED BETA
$((\lambda x.e)\ v) \rightarrow e[v/x]$

The inside-out grammar for contexts is

Core Language: Reduction Contexts using Inside-Out Grammar

$$
\begin{array}{l}
E ::= [\,] \\
\quad \mid E \circ -([\,], e) \\
\quad \mid E \circ -(v, [\,]) \\
\quad \mid E \circ ([\,]\ e) \\
\quad \mid E \circ (v\ [\,])
\end{array}
$$

and the context machine is:

Core Language: The Context Machine

$$
\begin{array}{ll}
\langle -(e_1, e_2),\ E\rangle & \rightarrow \langle e_1,\ E \circ -([\,], e_2)\rangle \\
\langle (e_1\ e_2),\ E\rangle & \rightarrow \langle e_1,\ E \circ ([\,]\ e_2)\rangle \\
 & \\
\langle v,\ [\,]\rangle & \rightarrow v \\
\langle v,\ E \circ -([\,], e_2)\rangle & \rightarrow \langle e_2,\ E \circ -(v, [\,])\rangle \\
\langle n,\ E \circ -(m, [\,])\rangle & \rightarrow \langle m - n,\ E\rangle \\
\langle v,\ E \circ ([\,]\ e_2)\rangle & \rightarrow \langle e_2,\ E \circ (v\ [\,])\rangle \\
\langle v,\ E \circ ((\lambda x.e)\ [\,])\rangle & \rightarrow \langle e[v/x],\ E\rangle
\end{array}
$$

Here we've separated the rules into two sets: one set that dispatches on the form of a non-value expression, and another that deals with a value, and instead dispatches on the form of the context.

Another way of writing these rules is to observe that each kind of context appears exactly twice: once on the right, where it is created, and once on the left, where it is used. If we group the rules in this way, we can see the analogy with big-step semantics

Core Language: The Context Machine (Alternate Grouping)

$$
\begin{array}{ll}
\langle v,\ [\,]\rangle & \rightarrow v \\
 & \\
\langle -(e_1, e_2),\ E\rangle & \rightarrow \langle e_1,\ E \circ -([\,], e_2)\rangle \\
\langle v,\ E \circ -([\,], e_2)\rangle & \rightarrow \langle e_2,\ E \circ -(v, [\,])\rangle \\
\langle n,\ E \circ -(m, [\,])\rangle & \rightarrow \langle m - n,\ E\rangle \\
 & \\
\langle (e_1\ e_2),\ E\rangle & \rightarrow \langle e_1,\ E \circ ([\,]\ e_2)\rangle \\
\langle v,\ E \circ ([\,]\ e_2)\rangle & \rightarrow \langle e_2,\ E \circ (v\ [\,])\rangle \\
\langle v,\ E \circ ((\lambda x.e)\ [\,])\rangle & \rightarrow \langle e[v/x],\ E\rangle
\end{array}
$$

This is sometimes called the *CK machine* (C = Code, K = [K]ontext).

9.1 Implementation

Our implementation will mimic our small-step implementation, except that it will involve two registers, K and e. Everything is inside `core-lang/ck.scm`.

First we represent frames. Continuations are lists of frames, inside-most first:

```
;;;;;;;;;;;;;;;; frames ;;;;;;;;;;;;;;;;

(define-datatype frame frame?
  (diff1-frame                      ; represents -([],e)
    (arg2 expression?))
  (diff2-frame                      ; represents -(v,[])
    (val1 expression?))
  (app-rator-frame                  ; represents ([] e2)
    (rand expression?))
  (app-rand-frame                   ; represents (v [])
    (rator expression?))
  )

;; ;;;;;;;;;;;;;;;; continuations ;;;;;;;;;;;;;;;;

(define cont? (list-of frame?))

(define (empty-cont) '())

;; empty list denotes []
;; (cons frame cont) denotes cont o frame, so continuations are
;; represented inside-out
```

`step-once` takes two arguments: a continuation and an expression, and returns either two values, a continuation and a expression (packaged as an `answer`) or `#f`if no step is possible:

```
;; step-once : cont * expression -> (maybe (cont * expression))
;; take one step, or return #f if impossible
;; note this is NOT recursive

(define (step-once k exp)
  (if *trace* (eopl:pretty-print (list exp k)))
  (cases expression exp

    (const-exp (n) (step-once-val k exp))
    (diff-exp (e1 e2)
      (answer
        (cons (diff1-frame e2) k)  ; new frame
        e1))                       ; focus on e1

    (var-exp (id) (eopl:error 'step-once
                    "open term to step-once ~s"
                    exp))

    (proc-exp (id body) (step-once-val k exp))

    (app-exp (e1 e2)
      (answer
        (cons (app-rator-frame e2) k) ; new frame
        e1))                          ; focus on e1
    ))
```

When its expression argument is a value, `step-once` calls `step-once-val` to search up in the continuation and perform the reduction. (Think like `apply-cont`, but only one step).

```
;; step-once searches up til it finds the redex, and then reduces it.

(define (step-once-val k v)
  (if (null? k) #f                    ; empty cont, so can't step
    (let ((k (cdr k))
          (top-frame (car k)))
      (cases frame top-frame
        (diff1-frame (e2)             ; ok, let's focus on e2
          (answer
            (cons (diff2-frame v) k)
            e2))
        (diff2-frame (val1)           ; see if we can reduce
          (let ((val2 v))
            (cond
              ((reduce-diff-redex val1 val2)
               => (lambda (reductum) (answer k reductum)))
              (else #f))))
        (app-rator-frame (rand-exp)   ; focus on the rand-exp
          (let ((rator-val v))
            (answer
              (cons (app-rand-frame rator-val) k)
              rand-exp)))
        (app-rand-frame (rator-val)   ; like diff2-frame
          (let ((rand-val v))
            (cond
              ((reduce-beta-redex rator-val rand-val)
               => (lambda (reductum) (answer k reductum)))
              (else #f))))
        ))))
```

As we did for smallstep, we wrap `step-once` in a procedure, here called `step-many`, which calls `step-once` until it can't step any more. `reduce-to-value` calls `step-many` and checks to see whether the reduction finished (with K empty), or whether it got stuck along the way.

```
  ;; step-many : cont * exp ->  cont * exp
  ;; step as many times as possible.

  (define (step-many k exp)
    (cond
      ((step-once k exp)
       => (lambda (ans) (step-many
                         (answer->cont ans)
                         (answer->exp  ans))))
      (else (answer k exp))))

  ;; reduce-to-value just invokes step-many
  (define (reduce-to-value exp k)
    (let ((ans (step-many k exp)))
      (if (null? (answer->cont ans))
        (answer->exp ans)
        (eopl:error 'reduce-to-value
          "stuck:~%
term         = ~s~%
continuation = ~s~%"
          (answer->exp ans)
          (answer->cont ans)))))
```

And that's all there is; the rest is engineering.

9.2 letrec

`letrec` in smallstep semantics is tricky. We don't have the luxury of creating special environments, since all we have is substitution.

Let's consider a letrec that declares only 1 procedure, let's call it f_1:

$$\texttt{letrec } f_1(x_1) = e_1 \texttt{ in } e_2$$

We want to run e_2 in an environment where f_1 is bound to a procedure, and in the body of that procedure, f_1 is bound to "the right thing.". We can do this by substituting:

$$\texttt{letrec } f_1(x_1) = e_1 \texttt{ in } e \rightarrow e[f_1 = \texttt{proc } (x_1) \texttt{ letrec } f_1(x_1) = e_1 \texttt{ in } e_1]$$

If the letrec declared two procedures f_1 and f_2, their bodies e_1 and e_2 would each have to know about each other. So the corresponding substitution is

$$\begin{array}{l}\texttt{letrec } f_1(x_1) = e_1,\, f_2(x_2) = e_2 \texttt{ in } e \\ \rightarrow \\ e[f_1 = \texttt{proc } (x_1) \texttt{ letrec } f_1(x_1) = e_1,\, f_2(x_2) = e_2 \texttt{ in } e_1, \\ \quad f_2 = \texttt{proc } (x_2) \texttt{ letrec } f_1(x_1) = e_1,\, f_2(x_2) = e_2 \texttt{ in } e_2]\end{array}$$

In general, if

$$\Delta = \{f_i(x_i) = e_i\}_i$$

then

$$\texttt{letrec } \Delta \texttt{ in } e \;\rightarrow\; e[f_i = \texttt{proc } (x_i) \texttt{ letrec } \Delta \texttt{ in } e_i]_i$$

where $\{\ \}_i$ or $[\ ]_i$ means "do this for all i, like $\sum_i$ or $\bigcup_i$.

9.3 Doing it with a Store

Let's scale this up to handle IMPLICIT-REFS. Instead of reducing a configuration that looks like $K[e]$, we'll be reducing a configuration that looks like $\langle K[e],\ s\rangle$: a pair of an expression and a store.

I'm tired of writing out Tex, so we'll do it in ASCII for now. ASCII is actually not bad for this, with some practice.

```
Smallstep rules for the implicit-refs machine

=================================================================

Expressions:

e ::= n | -(e1,e2)
      zero?(e1) | if e1 then e2 else e3
      x | let x = e1 in e2 | proc(x)e | (e1 e2)
      letrec f(x)=e* in e
      begin e; e* end
      set x = e
      //---additional expression forms introduced by substitution---//
      %loc(n)       // location values  loc ranges over
              expressions of this form
      %true              // boolean values
      %false             //
      %set1(e1,e2)       // substituted a %loc(n) for a variable in a
                         // set!

Configurations:

C ::= <e, s>   where e is an expression and s is a store

=================================================================
```

```
Values:

v ::= n
      proc(x)e           // must be closed
      %true, %false
      %loc(n)

=================================================================

Frames and Reduction Contexts:

F ::= -([],e)           // diff1-frame
      -(v,[])           // diff2-frame
      zero?([])         // zero?-frame
      if [] then e1 else e2   // if-test-frame
      let id=[] in e2   // let-rhs-frame
      ([] e2)           // app-rator-frame
      (v [])            // app-rand-frame
      begin [] e* end   // begin-frame
      set l = []        // assign-rhs-frame

K = F*

=================================================================

Reductions on expressions:

-(n1,n2)-> p           // where p = n1-n2

if %true then e1 else e2 -> e1

if %false then e1 else e2 -> e2

letrec D in e -> e[f_i = proc(x_i)letrec D in e_i]_i
  where D = {f_i=proc(x_i)e_i}_i

=================================================================
```

```
Reductions on configurations:

       l -> r
--------------------      Ordinary reductions lift to configurations
<K[l],s> -> <K[r],s>

Store operations:

<K[loc(n)],s> -> <K[v],s>  where v = s(n)

<K[(proc(x)e v)], s> -> <K[e[loc(n)/x]], s[v/n]>  where n not in dom(s).

<K[let x=v in e], s> -> <K[e[loc(n)/x]], s[v/n]>  where n not in dom(s).

================================================================
```

Now we can move on to the CK rules. This is technically called a *CKS machine*.

```
Search-down rules:

search-down <K, n, s>              -> focus-up <K, n, s>
search-down <K, proc(x)e, s>       -> focus-up <K, proc(x)e, s>
search-down <K, %true, s>          -> focus-up <K, %true, s>
search-down <K, %false, s>         -> focus-up <K, %false, s>

search-down <K, loc(n), s>         -> focus-up <K, s(n), s>
search-down <K, letrec D in e, s>  -> search-down <K, reduce-letrec(e,D), s>

search-down <K, -(e1,e2),s>        -> search-down <K o -([],e2), e1, s>
search-down <K, zero?(e1), s>      -> search-down <K o zero?([]), e1, s>
search-down <K, if(e1,e2,e3), s>   -> search-down <K o if([],e2,e3), e1, s>
search-down <K, let x=e1 in e2, s> -> search-down <K o let x=[] in e2, e1, s>
search-down <K, (e1 e2), s>        -> search-down <K o ([] e2), e1, s>

search-down <K, begin e1 e* end, s> -> search-down <K o begin [] e*, e1, s>

search-down <K, set loc(n)=e1, s>   -> search-down <K o set loc(n)=[], e1, s>

x and set x=e are not closed, so either is an error.


================================================================
```

```
Focus-up rules:

focus-up <[], v, s>  -> v

focus-up <K o -([],e2), v1, s>       -> search-down <K o -(v1,[]), e2, s>
focus-up <K o -(v1,[]), v2, s>       -> focus-up <K, p, s>  where p = v1-v2

focus-up <K o ([] e2), v1, s>        -> search-down <K o (v1 []), e2, s>
focus-up <K o (proc(x)e []), v2, s>  -> search-down <K, e2[n/x], s[v2/n]>
                                         where n is new in s.

focus-up <K o zero?([]), v1, s>      -> focus-up    <K, %true, s>
                                         or <K, %false, s>
focus-up <K o if([],e2,e3), v1, s>   -> search-down <K, e2, s>
                                         or <K, e3, s>

focus-up <K o let x=[] in e2, v1, s> -> search-down <K, e2[n/x], s[v1/n]>
                                         where n is new in s.

focus-up <K o begin [] empty, v1, s>     -> focus-up <K, v1, s>
focus-up <K o begin [] (e2 . e*), v1, s> -> search-down
                                             <K o begin [] e*, e2, s>

focus-up <K o set loc(n)=[], v, s>    -> focus-up <K, 27, s[v/n]>

================================================================
```

It's often helpful to group these the other way, grouping them by the operation they implement, so that each time `search-down` grows a continuation with a new frame, it's grouped with the corresponding `focus-up` line that consumes that frame.

```
focus-up <[], v, s>  -> v

search-down <K, n, s>              -> focus-up <K, n, s>
search-down <K, proc(x)e, s>       -> focus-up <K, proc(x)e, s>
search-down <K, %true, s>          -> focus-up <K, %true, s>
search-down <K, %false, s>         -> focus-up <K, %false, s>
search-down <K, loc(n), s>         -> focus-up <K, s(n), s>
search-down <K, letrec D in e, s>  -> search-down <K, reduce-letrec(e,D), s>

search-down <K, -(e1,e2),s>        -> search-down <K o -([],e2), e1, s>
focus-up <K o -([],e2), v1, s>     -> search-down <K o -(v1,[]), e2, s>
focus-up <K o -(v1,[]), v2, s>     -> focus-up    <K, p, s>
                                       where p = v1-v2

search-down <K, zero?(e1), s>      -> search-down <K o zero?([]), e1, s>
focus-up <K o zero?([]), v1, s>    -> focus-up    <K, %true, s>
                                       or <K, %false, s>

search-down <K, if(e1,e2,e3), s>   -> search-down <K o if([],e2,e3), e1, s>
focus-up <K o if([],e2,e3), v1, s> -> search-down <K, e2, s>
                                       or <K, e3, s>

search-down <K, let x=e1 in e2, s>   -> search-down
                                          <K o let x=[] in e2, e1, s>
focus-up <K o let x=[] in e2, v1, s> -> search-down <K, e2[n/x], s[v1/n]>
                                          where n is new in s.

search-down <K, (e1 e2), s>         -> search-down <K o ([] e2), e1, s>
focus-up <K o ([] e2), v1, s>       -> search-down <K o (v1 []), e2, s>
focus-up <K o (proc(x)e []), v2, s> -> search-down <K, e2[n/x], s[v2/n]>
                                          where n is new in s.
```

```
search-down <K, begin e1 e* end, s>      -> search-down
                                            <K o begin [] e*, e1, s>
focus-up <K o begin [] empty, v1, s>     -> focus-up <K, v1, s>
focus-up <K o begin [] (e2 . e*), v1, s> -> search-down
                                            <K o begin [] e*, e2, s>

search-down <K, set loc(n)=e1, s>  -> search-down
                                        <K o set loc(n)=[], e1, s>
focus-up <K o set loc(n)=[], v, s> -> focus-up <K, 27, s[v/n]>

================================================================
x and set x=e are not closed, so either is an error.
```

Observe that in this presentation, there's a clear analogy to the big-step rules.

9.4 Implementation

This is in `interps/implicit-refs-via-cks`. Our rules describe a state machine with two states and three registers. We could code this up in any number of ways. One obvious choice is to write two mutually-tail-recursive procedures, `search-down` and `focus-up`.

We've chosen to break the loop at `search-down`, as we did in our earlier example. Our procedure `step-once` takes a $\langle K,\ e,\ s\rangle$ `search-down` configuration and produces the next `search-down` configuration (or `#f`is there is none).

This technique of breaking up a tail-recursive loop is called a `trampoline`. It is handy to know if you have to write a tail-recursion in a language that does not implement tail recursion properly.

I've introduced some interior abstractions to make the code simpler to read (I hope!).