CESK to CEKF

CEKF Machine

This machine is based on the CESK machine: Control, Environment Store and Continuation.

CEKF stands for Control, Environment, Continuation and Failure. It omits "Store" from the CESK machine (turning it into a purely functional CEK machine) but then adds "Fail", a backtracking continuation allowing trivial support for `amb` (see SICP pp. 412-437)

The rest of this document closely follows Matt Might's blog post [Writing an interpreter, CESK-style](), slightly amended to describe a CEKF machine.

Our grammar omits the `set!`, but remember it must still be in A-normal form, and we add `amb` and `back` to `cexp`:

Atomic expressions `aexp` always terminate and never cause an error:

```
lam ::= (λ (var1 ... varN) exp)
```

```
aexp ::= lam
      |  var
      |  #t  |  #f
      |  integer
      |  (prim aexp1 ... aexpN)
```

Complex expressions `cexp` might not terminate and might cause an error:

```
cexp ::= (aexp0 aexp1 ... aexpN)
      |  (if aexp exp exp)
      |  (call/cc aexp)
      |  (letrec ((var1 aexp1) ... (varN aexpN)) exp)
      |  (amb exp exp)
      |  (back)
```

Expressions `exp` are atomic, complex , `let` bound, or the terminating `DONE`.

```
exp ::= aexp
     |  cexp
     |  (let ((var exp)) exp)
     |  DONE
```

Primitives are built-in

```
prim ::=   +  |  -  |  *  |  =
```

CEKF state

We reduce the CESK state machine from four components to three by removing $Store$, then expand it back to four by adding $Fail$:

$$\varsigma \in \Sigma = \mathtt{\mathbf{Exp}} \times Env \times Kont \times Fail$$

Env

Environments directly map variables to values:

$$\rho \in Env = Var \rightharpoonup Value$$

Values

Values are the same as CESK:

$$val \in Value ::= \mathbf{void} \mid z \mid \mathbf{\#t} \mid \mathbf{\#f} \mid \mathbf{clo}(\mathtt{lam}, \rho) \mid \mathbf{cont}(\kappa)$$

z is an integer, $\mathbf{cont}$ is part of $Value$ because we support `call/cc`.

Continuations

$$\kappa \in Kont ::= \mathbf{letk}(\mathtt{var}, \mathtt{body}, \rho, \kappa) \mid \mathbf{halt}$$

e.g. when evaluating `exp` in `(let ((var exp)) body)` , the continuation of that evaluation is as in the `letk` above.

Failure Continuation

$$f \in Fail ::= \mathbf{backtrack}(\mathtt{exp}, \rho, \kappa, f) \mid \mathbf{end}$$

$\mathbf{end}$ is the failure continuation's equivalent to $\mathbf{halt}$. $\mathtt{exp}$ is the (unevaluated) second `exp` of the `amb`, ρ, κ and f are the values current when the `amb` was first evaluated.

`aexp` evaluation

`aexp` are evaluated with an auxiliary function $\mathcal{A}$, simplified because there is no store:

$$\mathcal{A} : \mathtt{aexp} \times Env \rightharpoonup Value$$

Variables get looked up in the environment:

$$\mathcal{A}(\mathtt{var}, \rho) = \rho(\mathtt{var}) \tag{1}$$

Constants evaluate to their value equivalents:

$$\mathcal{A}(\texttt{integer}, \rho) = z \tag{2}$$
$$\mathcal{A}(\texttt{\#t}, \rho) = \#\mathbf{t} \tag{3}$$
$$\mathcal{A}(\texttt{\#f}, \rho) = \#\mathbf{f} \tag{4}$$

Lambdas become closures:

$$\mathcal{A}(\texttt{lam}, \rho) = \mathbf{clo}(\texttt{lam}, \rho) \tag{5}$$

Primitive expressions are evaluated recursively:

$$\mathcal{A}((\texttt{prim aexp}_\texttt{1} \ldots \texttt{aexp}_\texttt{n}), \rho) = \mathcal{O}(\texttt{prim})(\mathcal{A}(\texttt{aexp}_\texttt{1}, \rho) \ldots \mathcal{A}(\texttt{aexp}_\texttt{n}, \rho)) \tag{6}$$

where

$$\mathcal{O}(\texttt{prim}) = (\mathit{Value}^* \rightharpoonup \mathit{Value})$$

maps a primitive to its corresponding operation.

The `step` function

`step` goes from one state to the next.

$$\mathit{step} : \Sigma \rightharpoonup \Sigma$$

`step` for function calls

For ~~procedure~~ function calls, `step` first evaluates the function, then the arguments, then it applies the function:

$$\mathit{step}((\texttt{aexp}_\texttt{0} \ \texttt{aexp}_\texttt{1} \ldots \texttt{aexp}_\texttt{n}), \rho, \kappa, f) = \mathit{applyproc}(\mathit{proc}, \langle \mathit{val}_1, \ldots \mathit{val}_n \rangle, \kappa, f) \tag{7}$$

where

$$\mathit{proc} = \mathcal{A}(\texttt{aexp}_\texttt{0}, \rho) \tag{8}$$
$$\mathit{val}_i = \mathcal{A}(\texttt{aexp}_\texttt{i}, \rho) \tag{9}$$

$\mathit{applyproc}$ (defined later) doesn't need an Env (ρ) because it uses the one in the procedure (remember $\mathcal{A}(\texttt{lam}, \rho) = \mathbf{clo}(\texttt{lam}, \rho)$).

Return

When the expression under evaluation is an `aexp`, that means we need to return it to the continuation:

$$\mathit{step}(\texttt{aexp}, \rho, \kappa, f) = \mathit{applykont}(\kappa, \mathcal{A}(\texttt{aexp}, \rho), f) \tag{10}$$

where

$$\mathit{applykont} : \mathit{Kont} \times \mathit{Value} \times \mathit{Fail} \rightharpoonup \Sigma$$

is defined below.

Conditionals

$$step((\texttt{if aexp e}_{\texttt{true}}\ \texttt{e}_{\texttt{false}}), \rho, \kappa, f) = \begin{cases} (\texttt{e}_{\texttt{false}}, \rho, \kappa, f) & \mathcal{A}(\texttt{aexp}, \rho) = \#f \\ (\texttt{e}_{\texttt{true}}, \rho, \kappa, f) & \text{otherwise} \end{cases} \tag{11}$$

We might want to come back and revise this once we have stricter types.

Let

Evaluating `let` forces the creation of a continuation

$$step((\texttt{let (var exp) body}), \rho, \kappa, f) = (\texttt{exp}, \rho, \kappa', f) \tag{12}$$

where

$$\kappa' = \mathbf{letk}(\texttt{var}, \texttt{body}, \rho, \kappa) \tag{13}$$

Recursion

In CESK, `letrec` is done by extending the environment to point at fresh store locations, then evaluating the expressions in the extended environment, then assigning the values in the store.

Even then this only works if the computed values are closures, they can't actually *use* the values before they are assigned.

I'm thinking that in CEK, for `letrec` only, we allow assignment into the Env (treating it like a store) because we're not bound by functional constraints if we're eventually implementing in C. We couldn't directly convert this to Haskell though.

$$step((\texttt{letrec ((var}_1\ \texttt{aexp}_1) \ldots (\texttt{var}_\texttt{n}\ \texttt{aexp}_\texttt{n}))\ \texttt{body}), \rho, \kappa, f) = (\texttt{body}, \rho', \kappa, f) \tag{14}$$

where:

$$\rho' = \rho[\texttt{var}_\texttt{i} \Rightarrow \mathbf{void}] \tag{15}$$

but subsequently mutated with

$$\rho'[\texttt{var}_\texttt{i}] \Leftarrow \mathcal{A}(\texttt{aexp}_\texttt{i}, \rho') \tag{16}$$

First class continuations

`call/cc` takes a function as argument and invokes it with the current continuation (dressed up to look like a function) as its only argument:

$$step((\texttt{call/cc aexp}), \rho, \kappa, f) = applyproc(\mathcal{A}(\texttt{aexp}, \rho), \mathbf{cont}(\kappa), \kappa, f) \tag{17}$$

Amb

`amb` arranges the next state such that its first argument will be evaluated, and additionally installs a new Fail continuation that, if backtracked to, will resume computation from the same state, except evaluating the second argument.

$$\mathit{step}((\texttt{amb}\ \texttt{exp}_1\ \texttt{exp}_2), \rho, \kappa, f) = (\texttt{exp}_1, \rho, \kappa, \mathbf{backtrack}(\texttt{exp}_2, \rho, \kappa, f)) \tag{18}$$

Back

`back` invokes the failure continuation, restoring the state captured by `amb`.

$$\mathit{step}((\texttt{back}), \rho, \kappa, \mathbf{backtrack}(\texttt{exp}, \rho', \kappa', f)) = (\texttt{exp}, \rho', \kappa', f) \tag{19}$$

$$\mathit{step}((\texttt{back}), \rho, \kappa, \mathbf{end}) = (\texttt{DONE}, \rho, \mathbf{halt}, \mathbf{end}) \tag{20}$$

The `DONE` Exp signals termination.

Applying procedures

$$\mathit{applyproc} : \mathit{Value} \times \mathit{Value}^* \times \mathit{Kont} \times \mathit{Fail} \rightharpoonup \Sigma$$

$$\mathit{applyproc}(\mathbf{clo}((\lambda(\texttt{var}_1 \ldots \texttt{var}_\texttt{n})\ \texttt{body})\ \langle \mathit{val}_1 \ldots \mathit{val}_n \rangle, \kappa, f) = (\texttt{body}, \rho', \kappa, f) \tag{21}$$

where

$$\rho' = \rho[\texttt{var}_\texttt{i} \Rightarrow \mathit{val}_i] \tag{22}$$

Applying continuations

$$\mathit{applykont} : \mathit{Kont} \times \mathit{Value} \times \mathit{Fail} \rightharpoonup \Sigma$$

$$\mathit{applykont}(\mathbf{letk}(\texttt{v}, \texttt{body}, \rho, \kappa), \mathit{val}, f) = (\texttt{body}, \rho[\texttt{v} \Rightarrow \mathit{val}], \kappa, f) \tag{23}$$

$$\mathit{applykont}(\mathbf{halt}, \mathit{val}, f) = (\texttt{DONE}, \rho, \mathbf{halt}, \mathbf{end}) \tag{24}$$

Q. Should $\mathit{applycont}$ get f from its arguments or should we put it in the $\mathbf{letk}$?
A. probably best to get it from its arguments, then we will backtrack through `call/cc`.

Again what happens on return to the $\mathbf{halt}$ continuation is not well defined, so I've added a `DONE` expression to signal termination.