Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Abstract register machines

Lecture 23 Tuesday, April 19, 2016

1 Why abstract machines?

So far in the class, we have seen a variety of language features. To give a precise meaning to each feature, we have designed a calculus of a small language that has the feature and described the operational semantics of the calculus with a set of inference rules. Thus our calculi have given us a formal way to explain the meaning of any program of each language, including programs that contain the feature of interest.

However, our calculi do not explain how to easily and efficiently translate the operational semantics of each language to a correct interpreter for the language. To that end, in this lecture, we examine how we can translate the operational semantics of a language calculus into an abstract register machine. An abstract register machine deconstructs methodically each step of the operational semantics into a different task represented as a separate machine register. Thus an abstract machine for a language calculus provides a provably correct blueprint for an interpreter for the language. At the same time, the abstract machine reveals sources of inefficiency in the way we perform execution steps. Thus we can further refine the configuration of the machine's registers to derive a more efficient abstract machine, i.e., a more efficient and correct blueprint for an interpreter for the language.

2 The CC machine

Consider the following calculus for a simple language:

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid \delta(e_1, ..., e_m) \\ v &::= n \mid \lambda x.\, e \end{aligned}$$

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid \delta(n_1, ..., n_m, E, e_1, ..., e_k)$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}} \qquad \frac{\delta(n_1, ..., n_m) = n}{\delta(n_1, ..., n_m) \longrightarrow n}$$

The only aspect of this calculus that is different to similar calculi we have seen before is the δ operator. In previous calculi, we had operators such as $+$, $-$, etc. The δ operator stands for any of these arithmetic operators, i.e., it is a syntactic abstraction for n-ary arithmetic operators. Of course programmers that write programs in our simple language do not use δ but a specific operation arithmetic operator. For example in the following small program, the programmer uses $+$ to add two numbers: $((\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y))\ (1+20)$.

Now let's evaluate the above small program and discuss some facts about the evaluation process:

$$\begin{aligned} & ((\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y + y))\ (1 + 20) \\ \longrightarrow\ & (\lambda x.\, (\lambda y.\, (y + y))\ x)\ (1 + 20) \\ \longrightarrow\ & (\lambda x.\, (\lambda y.\, (y + y))\ x)\ 21 \\ \longrightarrow\ & (\lambda y.\, (y + y))\ 21 \\ \longrightarrow\ & 21 + 21 \\ \longrightarrow\ & 42 \end{aligned}$$

Notice that each evaluation step consists in reality of three distinct sub-steps. First, we test if the expression we are about to evaluate is a value. If it is, we have evaluated the expression fully and we stop the evaluation. If not we apply the first rule of the semantics and we split the expression in two parts: an evaluation context and an expression that we can reduce with one of the other rules. This second expression is also know as a redex. Then, we inspect the shape of the redex (is it a function application or a use of an arithmetic operator?) and we apply the corresponding rule. Finally, we plug the result of this last rule application into the evaluation context and we get back the next expression we need to evaluate. Of course, we repeat this process to perform the next evaluation step until we end up with the value result of the program.

This analysis reveals one subtle point about how we can translate our semantics for the language to an interpreter. An evaluation step is more complicated than it seems and thus an interpreter writer should tease out each sub-step into a separate function that can be re-used over and over again.

Furthermore, the analysis reveals a point of inefficiency in the way we perform evaluation steps. In an evaluation trace, we repeatedly split an expression into an evaluation context and a redex. Often, though, this is not necessary. It is common that in one step we use an evaluation context and then in the next one we end up using the same evaluation context or one that is easy to construct from an evaluation context we used previously. For instance in the following example the evaluation context $(1 + [\cdot])$ is the same in the first two steps of the evaluation trace :

$$\begin{aligned} & 1 + (\lambda x.\,(\lambda y.\,41)\ 42)\ 42 \\ \longrightarrow\ & 1 + (\lambda y.\,41)\ 42 \\ \longrightarrow\ & 1 + 41 \end{aligned}$$

As a first step to deal with the above two issues in the description of the semantics of the language, we introduce our first abstract register machine: the CC machine. The machine is an automaton with infinite number of states. Each state of the machine is a pair of two pieces of data: the current expression we evaluate and its evaluation context. We call the first piece of data the control register and the second the context register. The goal of the two registers is to keep redexes and evaluation contexts separate so that we avoid the repeated at each step deconstruction of our program to a redex and an evaluation context. The initial state of the machine has the program we want to evaluate as its control and the empty evaluation context ($[\cdot]$) as its context. Each evaluation step corresponds now to a transition from one state to another. Here are the rules that describe the possible transitions of the CC machine:

$\langle e, E\rangle$	$\longrightarrow_{CC}$	$\langle e', E'\rangle$	
$\langle e_1\ e_2, E\rangle$	$\longrightarrow_{CC}$	$\langle e_1, E[[\cdot]\ e_2]\rangle$	if $e_1 \neq v$
$\langle v_1\ e_2, E\rangle$	$\longrightarrow_{CC}$	$\langle e_2, E[v_1\ [\cdot]]\rangle$	if $e_2 \neq v$
$\langle \delta(n_1, ..., n_m, e, e_1, ..., e_k), E\rangle$	$\longrightarrow_{CC}$	$\langle e, E[\delta(n_1, ..., n_m, [\cdot], e_1, ..., e_k)]\rangle$	if $e \neq n$
$\langle (\lambda x.\, e)\ v, E\rangle$	$\longrightarrow_{CC}$	$\langle e\{v/x\}, E\rangle$	
$\langle \delta(n_1, ..., n_m), E\rangle$	$\longrightarrow_{CC}$	$\langle n, E\rangle$	where $\delta(n_1, ..., n_m) = n$
$\langle v, E[[\cdot]\ e]\rangle$	$\longrightarrow_{CC}$	$\langle v\ e, E\rangle$	
$\langle v_2, E[v_1\ [\cdot]]\rangle$	$\longrightarrow_{CC}$	$\langle v_1\ v_2, E\rangle$	
$\langle n, E[\delta(n_1, ..., n_m, [\cdot], e_1, ..., e_k)]\rangle$	$\longrightarrow_{CC}$	$\langle \delta(n_1, ..., n_m, n, e_1, ..., e_k), E\rangle$	

The first rule fires when the current state's control is an application where the function part has not been evaluated to a value. Its result is a new state where its control is the function part of the application and its context is the context of the previous state composed with the appropriate application evaluation context frame. The second rule is symmetric for the argument of an application and fires when we have fully evaluated the function part of the application but not its argument. The third rule is similar to the second one except it kicks off the evaluation of the first argument of an arithmetic operator that has not been fully evaluated. The fourth and fifth rules perform the β- and δ-reductions the same way as the corresponding inference rules from the beginning of the section. The last three rules help the machine resume evaluation when the control register of the machine holds a value. In all of the last three rules, the machine uses the inner-most frame of the context together with the control value to construct the new control expression.

Let's go back to the first example of this section and derive its evaluation context under the CC machine semantics.

$$\begin{array}{lll}
 & \langle((\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y))\ (1+20) & , [\cdot]\rangle \\
\longrightarrow_{CC} & \langle(\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y) & , [\cdot]\ (1+20)\rangle \\
\longrightarrow_{CC} & \langle\lambda x.\, (\lambda y.\, (y+y))\ x & , [\cdot]\ (1+20)\rangle \\
\longrightarrow_{CC} & \underline{\langle(\lambda x.\, (\lambda y.\, (y+y))\ x)\ (1+20)} & , [\cdot]\rangle \\
\longrightarrow_{CC} & \langle 1+20 & , (\lambda x.\, (\lambda y.\, (y+y))\ x)\ [\cdot]\rangle \\
\longrightarrow_{CC} & \langle 21 & , (\lambda x.\, (\lambda y.\, (y+y))\ x)\ [\cdot]\rangle \\
\longrightarrow_{CC} & \langle(\lambda x.\, (\lambda y.\, y+y)\ x)\ 21 & , [\cdot]\rangle \\
\longrightarrow_{CC} & \langle(\lambda y.\, y+y)\ 21 & , [\cdot]\rangle \\
\longrightarrow_{CC} & \langle 21+21 & , [\cdot]\rangle \\
\longrightarrow_{CC} & \langle 42 & , [\cdot]\rangle
\end{array}$$

Observe now the evaluation trace and in particular the underlined state. This state is the result of applying the sixth rule of the machine. The effect of this rule is that it reconstructs the application we are trying to evaluate to allow us to apply the first rule of the machine and proceed to evaluate the argument part of the application. Thus the underlined term is really an intermediate term due to the way we have designed the CC machine. Such terms show up again and again every time we try to evaluate an application or an arithmetic operation. But are these intermediate terms significant for the evaluation of a program? No, they are just the result of administrative steps, a technicality that introduces inefficiency in the CC machine.

To eradicate the intermediate terms, we introduce a new simplified CC machine, the SCC machine. Its state shape is the same as that of the CC machine, but its rules are slightly different:

$$\begin{array}{rcl}
\langle e, E\rangle & \longrightarrow_{SCC} & \langle e', E'\rangle \\
\hline
\langle e_1\ e_2, E\rangle & \longrightarrow_{SCC} & \langle e_1, E[[\cdot]\ e_2]\rangle \\
\langle \delta(e, e_1, ..., e_2), E\rangle & \longrightarrow_{SCC} & \langle e, E[\delta([\cdot], e_1, ..., e_n)]\rangle \\
\langle v, E[[\cdot]\ e]\rangle & \longrightarrow_{SCC} & \langle e, E[v\ [\cdot]]\rangle \\
\langle n, E[\delta(n_1, ..., n_m, [\cdot])]\rangle & \longrightarrow_{SCC} & \langle n', E\rangle \ \text{ where } \delta(n_1, ..., n_m, n) = n' \\
\langle v, E[\lambda x.\, e\ [\cdot]]\rangle & \longrightarrow_{SCC} & \langle e\{v/x\}, E\rangle \\
\langle n, E[\delta(n_1, ..., n_m, [\cdot], e, e_1, ..., e_k)]\rangle & \longrightarrow_{SCC} & \langle e, E[\delta(n_1, ..., n_m, n, [\cdot], e_1, ..., e_k)]\rangle
\end{array}$$

The first two rules start the evaluation of applications and arithmetic operations by placing their first sub-expression in the control register and the rest in the context register as a frame that extents the current context. The third and fourth rules continue the evaluation after the machine has fully evaluated a sub-expression of an application or an arithmetic operator. They pull out of the context the next sub-expression we have not evaluated yet, place it in the control register and push the evaluated sub-expression appropriately in the context register by modifying the innermost evaluation context frame of the context register. Notice how these rules avoid the redundant intermediate terms of the CC machine by shifting the control directly to the next sub-expression we need to evaluate. The last two rules perform the β- and δ-reductions but with a spin. Once we have evaluated all the sub-expressions of an application or an arithmetic operation, the rule performs the corresponding reduction without re-constructing a redex. Instead it uses the information from the innermost context frame to produce the result, places the result in the control register and pops the frame from the context register.

Here is the evaluation trace of our running example under the SCC machine semantics:

$$
\begin{array}{lll}
 & \langle(((\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y))\ (1+20) & , [\cdot]\rangle \\
\longrightarrow_{SCC} & \langle((\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y) & , [\cdot]\ (1+20)\rangle \\
\longrightarrow_{SCC} & \langle\lambda f.\, \lambda x.\, f\ x & , ([\cdot]\ \lambda y.\, (y+y))\ (1+20)\rangle \\
\longrightarrow_{SCC} & \langle\lambda y.\, (y+y) & , (\lambda f.\, \lambda x.\, f\ x\ [\cdot])\ (1+20)\rangle \\
\longrightarrow_{SCC} & \langle\lambda x.\, (\lambda y.\, (y+y))\ x & , [\cdot]\ (1+20)\rangle \\
\longrightarrow_{SCC} & \langle 1+20 & , (\lambda x.\, (\lambda y.\, (y+y))\ x)\ [\cdot]\rangle \\
\longrightarrow_{SCC} & \langle 1 & , (\lambda x.\, (\lambda y.\, (y+y))\ x)\ ([\cdot]+20)\rangle \\
\longrightarrow_{SCC} & \langle 20 & , (\lambda x.\, (\lambda y.\, (y+y))\ x)\ (1+[\cdot])\rangle \\
\longrightarrow_{SCC} & \langle 21 & , (\lambda x.\, (\lambda y.\, (y+y))\ x)\ [\cdot]\rangle \\
\longrightarrow_{SCC} & \langle(\lambda y.\, y+y)\ 21 & , [\cdot]\rangle \\
\longrightarrow_{SCC} & \langle\lambda y.\, y+y & , [\cdot]\ 21\rangle \\
\longrightarrow_{SCC} & \langle 21 & , (\lambda y.\, y+y)\ [\cdot]\rangle \\
\longrightarrow_{SCC} & \langle 21+21 & , [\cdot]\rangle \\
\longrightarrow_{SCC} & \langle 21 & , 21+[\cdot]\rangle \\
\longrightarrow_{SCC} & \langle 21 & , [\cdot]+21\rangle \\
\longrightarrow_{SCC} & \langle 42 & , [\cdot]\rangle
\end{array}
$$

As a final remark on SCC machines, note that in addition to removing unnecessary terms, they also reduce the portion of the expression in the control register that we need to inspect to decide which rule to apply next. Compare, for instance, the first rule of the CC and the SCC machine. The former requires a check that the function part of the application is not a value, the second has no such restriction. In fact in an SCC machine and in contrast to an CC, each rule only inspects the shape of the control register (is it an application or is it a use of an arithmetic operator or is it a value?). Of course this leads to evaluation traces with more steps than under CC semantics. However, these steps replace the extra checks on the contents of the control register and transform them into explicit steps in the SCC machine. Thus the SCC brings us even closer to the precise description of an interpreter for our language.

3 The CK machine

There are some unsatisfactory aspects in the way we evaluate programs in an SCC machine. Recall that we introduced machines to avoid having to split our programs again and again into a redex and an evaluation context. Does the SCC really achieve this goal? No. At each step we traverse the contents of the context register, find its innermost frame and use it to decide what step to take next. So we have simply shifted the problem of finding the next redex to the problem of finding the innermost frame of the current context.

But the SCC machine shows us the way to eliminate the issue. The solution is to change the representation of the context to make the innermost frame readily available. To that end we replace the context register with a register that holds a stack of evaluation context frames. This structure is historically called a *continuation* and by convention we use the Greek letter κ to range over continuations. The following grammar defines formally the structure of continuations:

$$\kappa ::= \mathbf{mt} \mid \langle \mathbf{fun}, \lambda x.\, e, \kappa\rangle \mid \langle \mathbf{arg}, e, \kappa\rangle \mid \langle \delta, \langle n, ..., n\rangle, \langle e, ..., e\rangle, \kappa\rangle$$

Each continuation corresponds to an evaluation context of our reduction semantics from the beginning of the lecture notes. The empty continuation $\mathbf{mt}$ corresponds to the empty evaluation context $[\cdot]$. The $\langle \mathbf{fun}, \lambda x.\, e, \kappa\rangle$ continuation corresponds to an evaluation context where the innermost frame is $\lambda x.\, e\ [\cdot]$ and the $\langle \mathbf{arg}, e, \kappa\rangle$ continuation corresponds to an evaluation context where the innermost frame is $[\cdot]\ e$. The $\langle \delta, \langle n_m, ..., n_1\rangle, \langle e_k, ..., e_1\rangle, \kappa\rangle$ corresponds to an evaluation context where the innermost frame is
$\delta(n_1, ..., n_m, [\cdot], e_1, ..., e_k)$.

There are three points worth making here. First, continuations turn the context inside out to make access to the innermost frame instant. Second, the use of a linked data structure for continuations gives as a representation of continuations that we can directly implement as a list. Third the $\langle \delta, \langle n_m, ..., n_1\rangle, \langle e_k, ..., e_1\rangle, \kappa\rangle$ continuation separates the arguments of the operator that we have evaluated from those that we have not yet and furthermore these arguments are stored in the continuation in reversed order for efficiency.

Notice that by replacing the context with a continuation, we change the shape of the state of our machine. Now we have a control register as before and a continuation register in each state. Thus the name of the new machine is CK.

Here are the rules of the CK machine:

$$\begin{array}{rcll}
\langle e, \kappa\rangle & \longrightarrow_{CK} & \langle e', \kappa'\rangle & \\
\hline
\langle e_1\ e_2, \kappa\rangle & \longrightarrow_{CK} & \langle e_1, \langle \mathbf{arg}, e_2, \kappa\rangle\rangle & \\
\langle \delta(e, e_1, ..., e_k), \kappa\rangle & \longrightarrow_{CK} & \langle e, \langle \delta, \langle\ \rangle, \langle e_k, ..., e_1\rangle, \kappa\rangle\rangle & \\
\langle \lambda x.\, e_1, \langle \mathbf{arg}, e_2, \kappa\rangle\rangle & \longrightarrow_{CK} & \langle e_2, \langle \mathbf{fun}, \lambda x.\, e_1, \kappa\rangle\rangle & \\
\langle v, \langle \mathbf{fun}, \lambda x.\, e, \kappa\rangle\rangle & \longrightarrow_{CK} & \langle e\{v/x\}, \kappa\rangle & \\
\langle n, \langle \delta, \langle n_m, ..., n_1\rangle, \langle e, e_k, ..., e_1\rangle, \kappa\rangle\rangle & \longrightarrow_{CK} & \langle e, \langle \delta, \langle n, n_m, ..., n_1\rangle, \langle e_k, ..., e_1\rangle, \kappa\rangle\rangle & \\
\langle n, \langle \delta, \langle n_m, ..., n_1\rangle, \langle\ \rangle, \kappa\rangle\rangle & \longrightarrow_{CK} & \langle n', \kappa\rangle & \text{where } \delta(n_1, ..., n_m, n) = n'
\end{array}$$

The difference between the SCC machine and the CK machine is that we use a different data structure to represent an evaluation context; the rules of the two machines are homomorphic.

Observe how our example evaluates under the CK semantics. Every step has a corresponding homomorphic step to the corresponding evaluation trace under the SCC semantics.

$$\begin{array}{lll}
 & \langle((\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y))\ (1+20) & , \mathbf{mt}\rangle \\
\longrightarrow_{CK} & \langle(\lambda f.\, \lambda x.\, f\ x)\ \lambda y.\, (y+y) & , \langle \mathbf{arg}, 1+20, \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle \lambda f.\, \lambda x.\, f\ x & , \langle \mathbf{arg}, \lambda y.\, (y+y), \langle \mathbf{arg}, 1+20, \mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CK} & \langle \lambda y.\, (y+y) & , \langle \mathbf{fun}, \lambda f.\, \lambda x.\, f\ x, \langle \mathbf{arg}, 1+20, \mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CK} & \langle \lambda x.\, (\lambda y.\, (y+y))\ x & , \langle \mathbf{arg}, 1+20, \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle 1+20 & , \langle \mathbf{fun}, \lambda x.\, (\lambda y.\, (y+y))\ x), \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle 1 & , \langle +, \langle\ \rangle, \langle 20\rangle, \langle \mathbf{fun}, \lambda x.\, (\lambda y.\, (y+y))\ x, \mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CK} & \langle 20 & , \langle +, \langle 1\rangle, \langle\ \rangle, \langle \mathbf{fun}, \lambda x.\, (\lambda y.\, (y+y))\ x, \mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CK} & \langle 21 & , \langle \mathbf{fun}, \lambda x.\, (\lambda y.\, (y+y))\ x, \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle(\lambda y.\, y+y)\ 21 & , \mathbf{mt}\rangle \\
\longrightarrow_{CK} & \langle \lambda y.\, y+y & , \langle \mathbf{arg}, 21, \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle 21 & , \langle \mathbf{fun}, \lambda y.\, y+y), \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle 21+21 & , \mathbf{mt}\rangle \\
\longrightarrow_{CK} & \langle 21 & , \langle +, \langle\ \rangle, \langle 21\rangle, \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle 21 & , \langle +, \langle 21\rangle, \langle\ \rangle, \mathbf{mt}\rangle\rangle \\
\longrightarrow_{CK} & \langle 42 & , \mathbf{mt}\rangle
\end{array}$$

4 The CEK machine

With the CK machine we have achieved the goal we stated at the beginning of the lecture notes; we can efficiently decide what rule to apply at every step without having to traverse the contents of the control nor the continuation register.

However, we have neglected a case where we still traverse the contents of the control register. This is the substitution part of the β-reduction (and its equivalent rules in the SCC and CK machines). Substitution of a value v for a free variable x in a term e requires traversing e, finding all free occurrences of x and replacing them with v. A pretty expensive task... Furthermore, the result of substitution is an expression even larger than the one we started with and that possibly leads to an even more expensive traversal during the next substitution.

An approach to make substitutions more efficient is to delay them as much as we can, i.e., until we really need to obtain the value associated with a free variable. In other words, we keep the substitution available to the machine until a variable becomes the control expression of the machine in which case we lookup the variable in the substitution. Thus, in this setting, the contents of the control register of our machine is no longer just an expression. Instead it is a pair of an expression, possibly with free variables, and a substitution that associates at least the free variables of the expression to values. The resulting structure is known as a *closure* while the conventional name of a substitution is an *environment*. Of course, environments cannot map variables to plain values because if they did and we looked up the value of a variable in the environment we would introduce back to the control register a plain expression. Thus environments map variables to closures. A similar change to the definition of continuations is also necessary. We call the new machine CEK because it combines a control and continuation register with an environment. The following grammar formally describes the definitions for closures (c and $\underline{v}$), continuations and CEK machine states:

$$\begin{aligned}
c &\in \{\langle e, \mathcal{E}\rangle \mid \mathcal{FV}(e) \subseteq \mathrm{dom}(\mathcal{E})\}\\
\underline{v} &\in \{\langle v, \mathcal{E}\rangle \mid \mathcal{FV}(v) \subseteq \mathrm{dom}(\mathcal{E})\}\\
\underline{\kappa} &::= \mathbf{mt} \mid \langle \mathbf{fun}, \underline{v}, \underline{\kappa}\rangle \mid \langle \mathbf{arg}, c, \underline{\kappa}\rangle \mid \langle \delta, \langle \underline{v}, ..., \underline{v}\rangle, \langle c, ..., c\rangle, \underline{\kappa}\rangle\\
S_{CEK} &::= \langle c, \underline{\kappa}\rangle
\end{aligned}$$

Before delving into the details of the CEK machine, we need to formally define environments. An environment is a finite mapping from variable to closures. This implies that we can see an environment either as a set or as a function:

$$\mathcal{E} = \{x_1{=}c_1, ..., x_n{=}c_n\}$$
$$\text{a finite function: for all } i \neq j \text{ and } 1 \leq i, j \leq n,\ x_i \neq x_j$$

In both cases, we can define for an environment $\mathcal{E}$ its domain $\mathrm{dom}(\mathcal{E})$ and its update function $\mathcal{E}[x \mapsto \underline{v}]$. The first has the obvious meaning while the second produces a new environment $\mathcal{E}'$ that maps x to $\underline{v}$ and any other variable in the domain of $\mathcal{E}$ to the same value as $\mathcal{E}$. Finally, observe that the definitions of environments, closures and continuations are mutually recursive.

Now we can move to the rules for the CEK machine:

$$\begin{array}{rcl}
\langle c, \underline{\kappa}\rangle & \longrightarrow_{CEK} & \langle c', \underline{\kappa}'\rangle\\
\hline
\langle\langle e_1\ e_2, \mathcal{E}\rangle, \underline{\kappa}\rangle & \longrightarrow_{CEK} & \langle\langle e_1, \mathcal{E}\rangle, \langle \mathbf{arg}, \langle e_2, \mathcal{E}\rangle, \underline{\kappa}\rangle\rangle\\
\langle\langle \delta(e, e_1, ..., e_k), \mathcal{E}\rangle, \underline{\kappa}\rangle & \longrightarrow_{CEK} & \langle\langle e, \mathcal{E}\rangle, \langle \delta, \langle\ \rangle, \langle\langle e_k, \mathcal{E}\rangle, ...\langle e_1, \mathcal{E}\rangle\rangle, \underline{\kappa}\rangle\rangle\\
\langle\langle \lambda x.\, e_1, \mathcal{E}_1\rangle, \langle \mathbf{arg}, \langle e_2, \mathcal{E}_2\rangle, \underline{\kappa}\rangle\rangle & \longrightarrow_{CEK} & \langle\langle e_2, \mathcal{E}_2\rangle, \langle \mathbf{fun}, \langle \lambda x.\, e_1, \mathcal{E}_1\rangle, \underline{\kappa}\rangle\rangle\\
\langle \underline{v}, \langle \mathbf{fun}, \langle \lambda x.\, e, \mathcal{E}\rangle, \underline{\kappa}\rangle\rangle & \longrightarrow_{CEK} & \langle\langle e, \mathcal{E}[x \mapsto \underline{v}]\rangle, \underline{\kappa}\rangle\\
\langle \underline{v}, \langle \delta, \langle \underline{v}_m, ..., \underline{v}_1\rangle, \langle c, c_k, ..., c_1\rangle, \underline{\kappa}\rangle\rangle & \longrightarrow_{CEK} & \langle c, \langle \delta, \langle \underline{v}, \underline{v}_m, ..., \underline{v}_1\rangle, \langle c_k, ..., c_1\rangle, \underline{\kappa}\rangle\rangle\\
\langle\langle n, \mathcal{E}\rangle, \langle \delta, \langle\langle n_m, \mathcal{E}_m\rangle, ..., \langle n_1, \mathcal{E}_1\rangle\rangle, \langle\ \rangle, \underline{\kappa}\rangle\rangle & \longrightarrow_{CEK} & \langle\langle n', \emptyset\rangle, \underline{\kappa}\rangle \ \text{ where } \delta(n_1, ..., n_m, n) = n'\\
\langle\langle x, \mathcal{E}\rangle, \underline{\kappa}\rangle & \longrightarrow_{CEK} & \langle \underline{v}, \underline{\kappa}\rangle \qquad \text{where } \mathcal{E}(x) = \underline{v}
\end{array}$$

These rules are quite similar to those of the CK machine except that expressions may contain free variables, the β-reduction rule updates the environment instead of performing a substitution and, when the control is a variable we look up its value in the current environment.

Here is how our running example evaluates under the CEK semantics:

$$
\begin{array}{llll}
 & \langle\langle((\lambda f.\,\lambda x.\,f\ x)\ \lambda y.\,(y+y))\ (1+20), & \emptyset\rangle & ,\mathbf{mt}\rangle \\
\longrightarrow_{CEK} & \langle\langle(\lambda f.\,\lambda x.\,f\ x)\ \lambda y.\,(y+y) & ,\emptyset\rangle & ,\langle\mathbf{arg},\langle 1+20,\emptyset\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle\lambda f.\,\lambda x.\,f\ x & ,\emptyset\rangle & ,\langle\mathbf{arg},\langle\lambda y.\,(y+y),\emptyset\rangle,\langle\mathbf{arg},\langle 1+20,\emptyset\rangle,\mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle\lambda y.\,(y+y) & ,\emptyset\rangle & ,\langle\mathbf{fun},\langle\lambda f.\,\lambda x.\,f\ x,\emptyset\rangle,\langle\mathbf{arg},\langle 1+20,\emptyset\rangle,\mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle\lambda x.\,f\ x & ,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle\}\rangle & ,\langle\mathbf{arg},\langle 1+20,\emptyset\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 1+20 & ,\emptyset\rangle & ,\langle\mathbf{fun},\langle\lambda x.\,f\ x,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 1 & ,\emptyset\rangle & ,\langle +,\langle\,\rangle,\langle\langle 20,\emptyset\rangle\rangle,\langle\mathbf{fun},\langle\lambda x.\,f\ x,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 20 & ,\emptyset\rangle & ,\langle +,\langle\langle 1,\emptyset\rangle\rangle,\langle\,\rangle,\langle\mathbf{fun},\langle\lambda x.\,f\ x,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 21 & ,\emptyset\rangle & ,\langle\mathbf{fun},\langle\lambda x.\,f\ x,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle f\ x & ,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle,x{=}\langle 21,\emptyset\rangle\}\rangle & ,\mathbf{mt}\rangle \\
\longrightarrow_{CEK} & \langle\langle f & ,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle,x{=}\langle 21,\emptyset\rangle\}\rangle & ,\langle\,\mathrm{arg},\langle x,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle,x{=}\langle 21,\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle\lambda y.\,(y+y) & ,\emptyset\rangle & ,\langle\,\mathrm{arg},\langle x,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle,x{=}\langle 21,\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle x & ,\{f{=}\langle\lambda y.\,(y+y),\emptyset\rangle,x{=}\langle 21,\emptyset\rangle\}\rangle & ,\langle\mathbf{fun},\langle\lambda y.\,(y+y),\emptyset\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 21 & ,\emptyset\rangle & ,\langle\mathbf{fun},\langle\lambda y.\,(y+y),\emptyset\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle y+y & ,\{y{=}\langle 21,\emptyset\rangle\}\rangle & ,\mathbf{mt}\rangle \\
\longrightarrow_{CEK} & \langle\langle y & ,\{y{=}\langle 21,\emptyset\rangle\}\rangle & ,\langle +,\langle\,\rangle,\langle y,\{y{=}\langle 21,\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 21 & ,\emptyset\rangle & ,\langle +,\langle\,\rangle,\langle y,\{y{=}\langle 21,\emptyset\rangle\}\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle y & ,\{y{=}\langle 21,\emptyset\rangle\}\rangle & ,\langle +,\langle\langle 21,\emptyset\rangle\rangle,\langle\,\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 21 & ,\emptyset\rangle & ,\langle +,\langle\langle 21,\emptyset\rangle\rangle,\langle\,\rangle,\mathbf{mt}\rangle\rangle \\
\longrightarrow_{CEK} & \langle\langle 42 & ,\emptyset\rangle & ,\mathbf{mt}\rangle
\end{array}
$$

As a final remark to the lecture notes, the methodical derivation of each machine makes us confident of its correctness, i.e., that each machine semantics describes the same semantics as the reduction rules from the beginning of the section. In fact, we can prove this property for each machine semantics and thus establish that the machines are a sound basis for constructing an interpreter for our simple language.