Continuations

The continuation of an expression is "what to do next" after the expression is evaluated. For example, to evaluate the program `(1 + 2) * 30` we follow these steps:

- We evaluate the expression `1 + 2`, which produces `3`
- The continuation of `1 + 2` multiplies the result (`3`) by `30`, which produces `90`.

If you think about writing a recursive interpreter, the continuation corresponds to the program's stack. In fact, "stacks" even occur in our derivations. Here is a partial derivation of `((1 + 4) + 3) + (2 + 7)` where we've calculated that `1 + 4` ⇓ `5` but have not done any other calculation:

$$\dfrac{\dfrac{\texttt{1+4} \Downarrow \texttt{5} \quad \texttt{3} \Downarrow \textit{?}}{\texttt{(1+4)} + \texttt{3} \Downarrow \textit{?}} \quad \texttt{(2+7)} \Downarrow \textit{?}}{\texttt{((1+4)+3)} + \texttt{(2+7)} \Downarrow \textit{?}}$$

Above, the continuation of `1+4` is—roughly speaking—the stack of expressions below it.

1 An Interpreter without Recursion

In the interpreters we have written so far, we have relied on recursion to evaluate expressions. We will now see how to use an *explicit continuation* to write an interpreter that uses iteration (i.e., a `while` loop) instead of recursion. You may need to do this to write an interpreter for a machine that does not have a stack or has very little stack space (e.g., a mobile device). As a reminder, fig. 28.1 shows the syntax and semantics of a language that is recursively defined. We will derive an alternative semantics for this language that uses explicit continuations and calculates the same result as the original semantics.

There are two key ideas:

1. We define a new type kind data structure that represents a *continuation frame* (κ) and a *continuation* ($\vec{\kappa}$) will be a list of continuation frames.
2. We define a relation that maps an expression and its continuation to a new expression and continuation, written $e_1,\vec{\kappa_1} \to e_2,\vec{\kappa_2}$. This relation will define a single step of evaluation and we will apply it repeatedly until the expression is a value and the continuation is empty. To evaluate any program, we always start with the empty continuation ($[]$).

For example, we use the following sequence of steps to evaluate `((1 + 20) + 300) + 4000` (in the empty continuation):

Syntax

e	=	n	numbers
	|	x	identifiers
	|	$e_1 + e_2$	addition
	|	**fun** x . e	functions
	|	$e_1\ e_2$	applications

Semantics

E-Num $n \Downarrow n$

E-Add $\dfrac{e_1 \Downarrow m \quad e_2 \Downarrow n}{e_1 + e_2 \Downarrow m + n}$

E-App $\dfrac{e_1 \Downarrow \textbf{fun}\ x\ .\ e \quad e_2 \Downarrow v \quad e\,[\,x\,/\,v\,] \Downarrow v'}{e_1\ e_2 \Downarrow v'}$

E-Fun **fun** x . $e \Downarrow$ **fun** x . e

Figure 28.1: Original syntax and semantics.

1. Our first step is to evaluate the expression on the left-hand side (1 + (20 + 300)) and remember that the right-hand side adds 4000:

   (1 + (20 + 300)) + 4000, [] → 1 + (20 + 300), [$\mathbf{add}_R$(4000)]

2. To evaluate `1 + (20 + 300)`, we have to evaluate the left-hand side (1) and remember that the right-hand side adds 20 + 300:

   1 + (20 + 300), [$\mathbf{add}_R$(4000)] → 1, [$\mathbf{add}_R$(20 + 300), $\mathbf{add}_R$(4000)]

3. The active expression is now a value (1) that cannot be evaluated further. However, when we examine the topmost continuation frame, we find a right-hand side expression that we have yet to evaluate ($\mathbf{add}_R$(20 + 300)). Therefore, we make 20 + 300 the active expression and remember the left-hand side *value* that we just calculated:

   1, [$\mathbf{add}_R$(20 + 300), $\mathbf{add}_R$(4000)] → 20 + 300, [$\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)]

4. To evaluate 20 + 300, we need to evaluate the left-hand side expression (20 and remember the right-hand side as before:

   20 + 300, [$\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)] → 20, [$\mathbf{add}_R$(300), $\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)]

5. The active expression is again a value (20) that cannot be evaluated further. However, when we examine the topmost continuation frame, we find a right-hand side expression (300) that we have yet to evaluate. Therefore, we make the expression 300 the active expression and remember the left-hand side value:

   20, [$\mathbf{add}_R$(300), $\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)] → 300, [$\mathbf{add}_L$(20), $\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)]

6. The active expression is again a value (300) that cannot be evaluated further. This time, when we examine the topmost continuation frame, we find a left-hand side value. Therefore, we calculate $20 + 300 = 320$ and pop the topmost continuation frame:

   300, [$\mathbf{add}_L$(20), $\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)] → 320,[$\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)]

7. The active expression is again a value (320) and the topmost continuation frame has a left-hand side value. Therefore, we calculate $1 + 320 = 321$ and pop the topmost continuation frame:

   320,[$\mathbf{add}_L$(1), $\mathbf{add}_R$(4000)] → 321, [$\mathbf{add}_R$(4000)]

8. The active expression is now value and the topmost continuation frame has a right-hand side expression (4000). Therefore, we make this expression active and remember that 321 was the left-hand side value:

   321, [$\mathbf{add}_R$(4000)] → 4000, [$\mathbf{add}_L$(321)]

9. The active expression is now a value (4000) and the topmost continuation frame has a left-hand side value. Therefore, we calculate $321 + 4000 = 4321$ and pop the continuation frame:

   4000, [$\mathbf{add}_L$(321)] → 4321, []

Finally, since the active expression is a value and the continuation is empty, we are done and the result is 4321. To summarize, to evaluate addition, we needed two kinds of continuation frames ($\mathbf{add}_L$ and $\mathbf{add}_R$) and three kinds of rules: (1) rules that push the right-hand side expression onto the continuation, i.e., steps 1, 2, 4, and 7; (2) rules that push the left-hand side value onto the continuation, i.e., steps 3, 5, and 8; and (3) rules that actually add two simple numbers, i.e., steps 6, 7, and 9. Figure 28.2 specifies the semantics of this language using explicit continuations. It includes the rules for function applications which follow the same principle we used for arithmetic expressions above. This style of semantics, where the continuation is an explicit data structure is known as a *CK-machine*.

1.1 Properties of CK Machines

Correctness We now have two ways reduce an expression to a value. We can either use our big-step semantics, $e \Downarrow v$, or we can use the CK-Machine, e,[] $\rightarrow^* v$,[]. Both approaches must calculate the same value, and this is a theorem that we can prove.

Expressions

e	=	n	numbers
	|	x	identifiers
	|	$e_1 + e_2$	addition
	|	**fun** x . e	functions
	|	$e_1\ e_2$	applications

Continuation Frames

κ	::=	$\mathbf{add}_R(e_2)$
	|	$\mathbf{add}_L(m)$
	|	$\mathbf{app}_R(e_2)$
	|	$\mathbf{app}_L(x, e)$

Reduction Rules

AddL	$e_1 + e_2, \vec{\kappa} \rightarrow e_1, \mathbf{add}_R(e_2) :: \vec{\kappa}$
AddR	$m, \mathbf{add}_R(e_2) :: \vec{\kappa} \rightarrow e_2, \mathbf{add}_L(m) :: \vec{\kappa}$
AddNum	$m + n, \vec{\kappa} \rightarrow r$ where $r = m + n$
AppL	$e_1\ e_2, \vec{\kappa} \rightarrow e_1, \mathbf{app}_R(e_2) :: \vec{\kappa}$
AppR	**fun** x . e , $\mathbf{app}_R(e_2) :: \vec{\kappa} \rightarrow e_2, \mathbf{app}_L(x, e) :: \vec{\kappa}$
β	$v, \mathbf{app}_L(x, e) :: \vec{\kappa} \rightarrow e[x/v], \vec{\kappa}$

Figure 28.2: Semantics with explicit continuations.

Expressions

e	=	$\cdots$	
	|	**let** $x = e_1$ **in** e_2	let expression
	|	**if** e_1 **then** e_2 **else** e_3	conditional

Continuation Frames

κ	::=	$\cdots$
	|	$\mathbf{let}(x, e_2)$
	|	$\mathbf{if}(e_2, e_3)$

Reduction Rules

LetL	**let** $x = e_1$ **in** $e_2, \vec{\kappa} \rightarrow e_1, \mathbf{let}(x, e_2) :: \vec{\kappa}$
LetR	$v, \mathbf{let}(x, e_2) :: \vec{\kappa} \rightarrow e[x/v], \vec{\kappa}$
IfCond	**if** e_1 **then** e_2 **else** $e_3, \vec{\kappa} \rightarrow e_1, \mathbf{if}(e_2, e_3) :: \vec{\kappa}$
IfTrue	$true, \mathbf{if}(e_2, e_3) :: \vec{\kappa} \rightarrow e_2, \vec{\kappa}$
IfFalse	$false, \mathbf{if}(e_2, e_3) :: \vec{\kappa} \rightarrow e_3, \vec{\kappa}$

Figure 28.3: Let expressions and conditionals. Extends fig. 28.2.

Proper tail calls The CK-machine preserves proper tail calls. i.e., applying a function does not grow the stack. To see an example of this in action, consider the Ω-combinator, which does not terminate:

$$\Omega = (\textbf{fun}\ x\ .\ x\ x)\ (\textbf{fun}\ x\ .\ x\ x)$$

The CK machine evaluates Ω as follows (which you should verify):

	Ω,[]	
=	(**fun** x . x x) (**fun** x . x x), []	
→	(**fun** x . x x), [$\mathbf{app}_R$(**fun** x . x x)]	by AppL
→	(**fun** x . x x), [$\mathbf{app}_L$(x, x x)]	by AppR
→	(**fun** x . x x) (**fun** x . x x), []	by β
=	Ω,[]	

Notice that the size of the continuation never exceeds 1, even though the expression continues evaluating forever.

More generally, consider the reduction rules that involve function application (i.e., AppL, AppR, and β). The AppL rule pushes a $\mathbf{app}_R$ continuation frame. The AppR rule pops the $\mathbf{app}_R$ frame and pushes an $\mathbf{app}_L$ frame. Finally, the β rule pops the $\mathbf{app}_L$ frame and does not push anything onto the continuation. i.e., the continuation ($\vec{\kappa}$) on the right-hand side of the β rule is the same continuation on the left-hand side of the AppL rule.

1.2 Additional Language Features

It is straightforward to extend the CK machine with additional features. For example, fig. 28.3 extends the language with conditionals and let-expressions. The principal is the same: to evaluate the named expression in a let-expression, we need to remember the name and the let-body, which we do with the **let**(x,e_2) frame. To evaluate the test expression in a conditional, we need to remember the expressions for either branch, which we do with the **if**(e_2,e_3) frame.

1.3 First Class Continuations

Since the CK-Machine represents continuations explicitly, we can now add language features that manipulate the continuation in more interesting ways. In fact, we can *turn continuations into a value* in the language.[1] Figure 28.4 does this by introducing a new operator called *control* (written $\mathcal{C}$). The expression $\mathcal{C}$(**fun** x . e) binds x to a value that represents the current continuation and evaluates e in the empty continuation. Within e, we can apply x like a function, which discards the current continuation and restores x.

[1] This is related to `setjmp` and `longjmp` in C.

Expressions

e	$=$	$\cdots$	
	$\mid$	$\mathcal{C}(e)$	continuation capture
	$\mid$	$\vec{\kappa}$	continuation value

Continuation Frames

κ	$::=$	$\cdots$	
	$\mid$	$cont$	
	$\mid$	$appCont(\vec{\kappa}')$	

Reduction Rules

Cont1	$\mathcal{C}(e), \vec{\kappa} \to e, c :: \vec{\kappa}$
Cont2	$\textsf{fun}\ x\ .\ e\ ,\ c :: \vec{\kappa} \to e\ [\ x\ /\ \vec{\kappa}\ ],\ []$
AppCont1	$\vec{\kappa}', \textbf{app}_R(e_2), \vec{\kappa} \to e_2, \textsf{appCont}(\vec{\kappa}') :: \vec{\kappa}$
AppCont2	$v,\ appCont(\vec{\kappa}') :: \vec{\kappa} \to v, \vec{\kappa}'$

Figure 28.4: First-class continuations. Extends fig. 28.2.

For example, the following expression produces 0:

$$
\begin{array}{rll}
 & 10 + \mathcal{C}(\textsf{fun}\ k\ .\ 0),\ [] & \\
\to & 10,\ [\textbf{add}_R(\mathcal{C}(\textsf{fun}\ k\ .\ 0))] & \textsc{AddL} \\
\to & \mathcal{C}(\textsf{fun}\ k\ .\ 0),\ [\textbf{add}_L(10)] & \textsc{AddR} \\
\to & \textsf{fun}\ k\ .\ 0,\ [cont, \textbf{add}_L(10)] & \textsc{Cont1} \\
\to & 0[k/[\textbf{add}_L(10)]],\ [] & \textsc{Cont2} \\
= & 0,\ [] &
\end{array}
$$

In contrast, the next expression produces 11:

$$
\begin{array}{rll}
 & 10 + \mathcal{C}(\textsf{fun}\ k\ .\ (k\ 1) + 2),\ [] & \\
\to & 10,\ [\textbf{add}_R(\mathcal{C}(\textsf{fun}\ k\ .\ (k\ 1) + 2))] & \textsc{AddL} \\
\to & \mathcal{C}(\textsf{fun}\ k\ .\ (k\ 1) + 2),\ [\textbf{add}_L(10)] & \textsc{AddR} \\
\to & \textsf{fun}\ k\ .\ (k\ 1) + 2,\ [cont, \textbf{add}_L(10)] & \textsc{Cont1} \\
\to & (k\ 1) + 2\ [k/[\textbf{add}_L(10)]],\ [] & \textsc{Cont2} \\
= & ([\textbf{add}_L(10)]\ 1) + 2,\ [] & \\
\to & ([\textbf{add}_L(10)]\ 1),\ [\textbf{add}_R(2)] & \textsc{AddL} \\
\to & [\textbf{add}_L(10)],\ [\textbf{app}_R(1),\ \textbf{add}_R(2)] & \textsc{AppL} \\
\to & 1,\ [appCont([\textbf{add}_L(10)]),\ \textbf{add}_R(2)] & \textsc{AppCont1} \\
\to & 1,\ [\textbf{add}_L(10)] & \textsc{AppCont2} \\
\to & 11,\ [] & \textsc{AddNum}
\end{array}
$$

$10 + \mathcal{C}(\textsf{fun}\ k\ .\ (k\ 1) + 2)$

Above, $\mathcal{C}$ evaluates the body in an empty continuation and the body applies $k\ 1$. This application discards the current continuation (which adds 2) and restores the saved continuation (which adds 10), thus produces 11.