**CS 4110 – Programming Languages and Logics**
**Lecture #21: Continuations**

# 1 Continuations

In each of the preceding translations, the control structure of the source language was translated directly into the corresponding control structure in the target language. For example:

$$\mathcal{T}[\![\lambda x.\, e]\!] = \lambda x.\, \mathcal{T}[\![e]\!]$$
$$\mathcal{T}[\![e_1\ e_2]\!] = \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!]$$

This style of translation works well when the source language is similar to the target language. However, when the control structures of the source and target languages differ more substantially, it doesn't work as well.

Continuations are a programming technique that may be used directly by a programmer, or used in program transformations by a compiler. Because they make the control flow of the program explicit, they can be used to overcome discrepancies between source and target languages in definitional translation. They can also be used to define the semantics of control-flow constructs such as exceptions.

Intuitively, a continuation represents "the rest of the program." Consider the program

$$\mathsf{if}\ \mathsf{foo} < 10\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar}$$

and consider the evaluation of the expression $\mathsf{foo} < 10$. When we finish evaluating this subexpression, we will evaluate the $\mathsf{if}$ statement, and then evaluate the appropriate branch. The *continuation* of the subexpression $\mathsf{foo} < 10$ is the rest of the computation that will occur after we evaluate the subexpression. We can write this continuation as a function that takes the result of the subexpression:

$$(\lambda y.\, \mathsf{if}\ y\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar})\ (\mathsf{foo} < 10)$$

The evaluation order and result of this program will be the same as the original expression; the difference is that we extracted the continuation of the subexpression in to a function.

The nice thing about continuations is that it makes the control explicit, and this is especially useful in the case of functional programs, where control is not explicit otherwise. In fact, we can rewrite a program to make continuations more explicit. Let's consider another program, and convert it so that continuations are explicit

$$(\lambda x.\, x)\ ((1 + 2) + 3) + 4$$

We'll start by defining a continuation for the outermost evaluation context, which takes a value, and applies the identity function to it.

$$k_0 = \lambda v.\, (\lambda x.\, x)\ v$$

The evaluation context that is evaluated next-to-last takes a value, adds 4 to it, and then passes the result to $k_0$.

$$k_1 = \lambda a.\, k_0\ (a + 4)$$

Likewise, for the next evaluation contexts.

$$\begin{aligned} k_2 &= \lambda b.\, k_1\ (b + 3) \\ k_3 &= \lambda c.\, k_2\ (c + 2) \end{aligned}$$

The program itself is now equivalent to $k_3\ 1$. Since $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is just syntactic sugar for $(\lambda x.\, e')\ e$, we can actually rewrite the above as

$$\begin{aligned} &\mathsf{let}\ c = 1\ \mathsf{in} \\ &\mathsf{let}\ b = c + 2\ \mathsf{in} \\ &\mathsf{let}\ a = b + 3\ \mathsf{in} \\ &\mathsf{let}\ v = a + 4\ \mathsf{in} \\ &(\lambda x.\, x)\ v \end{aligned}$$

This is fairly close to some machine instructions of the form:

$$\begin{aligned} &\mathsf{set}\ c, 1 \\ &\mathsf{add}\ b, c, 2 \\ &\mathsf{add}\ a, b, 3 \\ &\mathsf{add}\ v, a, 4 \\ &\mathsf{call}\ \mathsf{id}, v \end{aligned}$$

Using continuations, functions can be transformed into "functions that don't return"—i.e., functions that take, besides the usual arguments, an additional argument representing a continuation. When the function finishes, it invokes the continuation on its result, instead of returning the result to its caller. Writing functions in this way is usually referred to as Continuation-Passing Style, or CPS for short. For instance, the CPS version of factorial looks like the following:

$$\mathsf{FACT}_{cps} = \mathsf{Y}\ \lambda f.\, \lambda n, k.\, \mathsf{if}\ n = 0\ \mathsf{then}\ k\ 1\ \mathsf{else}\ f\ (n-1)\ (\lambda v.\, k\ (n * v))$$

Note that the last thing that code in $\mathsf{FACT}_{cps}$ does is call a function (either $k$ or $f$), and does not do anything with the result.

Continuation-passing style is an important concept in the compilation of functional languages and is used as an intermediate compiler representation (it has been used in compilers for Scheme, ML, etc). The main advantage is that CPS makes the control flow explicit and makes it easier to translate functional code to machine code where control is explicit (in the form of sequences of machine instructions and jumps). For instance, a CPS call can be easily translated into a jump to the invoked method, since the invoked function does not return the control.

### 1.1 CPS translation

We can translate $\lambda$-calculus programs into continuation-passing style. We define a translation function $\mathcal{CPS}[\![\cdot]\!]$, which takes a CBV $\lambda$-calculus expression, and translates the expression to a CBV $\lambda$-calculus expression in continuation-passing style.

Let's consider a translation from $\lambda$-calculus with pairs and integers. The syntax of the source language is as follows.

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$

The translation $\mathcal{CPS}[\![e]\!]$ will produce a function that whose argument is the continuation to which to pass the result. That is, for all expressions $e$, the translation is of the form $\mathcal{CPS}[\![e]\!] = \lambda k.\ \ldots$, where $k$ is a continuation. We will both assume and guarantee that for any expression $e$, the translation $\mathcal{CPS}[\![e]\!] = \lambda k.\ \ldots$ will apply $k$ to the result of evaluating $e$.

For convenience, instead of writing $\mathcal{CPS}[\![e]\!] = \lambda k.\ \ldots$ we write $\mathcal{CPS}[\![e]\!]\ k = \ldots$.

$$\begin{aligned}
\mathcal{CPS}[\![n]\!]\ k &= k\ n \\
\mathcal{CPS}[\![e_1 + e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda n.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda m.\, k\ (n+m))) && n \text{ is not a free variable of } e_2
\end{aligned}$$

$$\begin{aligned}
\mathcal{CPS}[\![(e_1, e_2)]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda v.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda w.\, k\ (v, w))) && v \text{ is not a free variable of } e_2 \\
\mathcal{CPS}[\![\#1\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v.\, k\ (\#1\ v)) \\
\mathcal{CPS}[\![\#2\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v.\, k\ (\#2\ v))
\end{aligned}$$

$$\begin{aligned}
\mathcal{CPS}[\![x]\!]\ k &= k\ x \\
\mathcal{CPS}[\![\lambda x.\, e]\!]\ k &= k\ (\lambda x.\, \lambda k'.\, \mathcal{CPS}[\![e]\!]\ k') && k' \text{ is not a free variable of } e \\
\mathcal{CPS}[\![e_1\ e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda f.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda v.\, f\ v\ k)) && f \text{ is not a free variable of } e_2
\end{aligned}$$

We translate a function $\lambda x.\, e$ to a function that takes an additional argument $k'$, which is the continuation after the function application. That is, $k'$ is the continuation to which we hand the result of evaluating the function body. In function application, we see that in addition to the actual argument, we also give the continuation as the additional argument.

Let's see an example translation and execution...

$$\begin{aligned}
\mathcal{CPS}[\![(\lambda a.\, a + 6)\ 7]\!]\ \mathsf{ID} &= \mathcal{CPS}[\![(\lambda a.\, a + 6)]\!]\ \ (\lambda f.\, \mathcal{CPS}[\![7]\!]\ (\lambda v.\, f\ v\ \mathsf{ID})) \\
&= (\lambda f.\, \mathcal{CPS}[\![7]\!]\ (\lambda v.\, f\ v\ \mathsf{ID}))\ (\lambda a, k'.\, \mathcal{CPS}[\![a + 6]\!] k') \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a + 6]\!] k') \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!]\ (\lambda n.\, \mathcal{CPS}[\![6]\!]\ (\lambda m.\, k'\ (m+n)))) \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!]\ (\lambda n.\, (\lambda m.\, k'\ (m+n))\ 6)) \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m+n))\ 6)\ a) \\
&\longrightarrow (\lambda v.\, (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m+n))\ 6)\ a)\ v\ \mathsf{ID})\ 7 \\
&\longrightarrow (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m+n))\ 6)\ a)\ 7\ \mathsf{ID} \\
&\longrightarrow (\lambda n.\, (\lambda m.\, \mathsf{ID}\ (m+n))\ 6)\ 7 \\
&\longrightarrow (\lambda m.\, \mathsf{ID}\ (m+7))\ 6 \\
&\longrightarrow \mathsf{ID}\ (6+7) \\
&\longrightarrow \mathsf{ID}\ 13 \\
&\longrightarrow 13
\end{aligned}$$