Comp Sci 704
Fall 2024

Lecture 21: Continuations

Ethan Cecchetti

When executing a program, the program is broken into two parts: the current expression being executed, and the *continuation*, which refers to what we do next—how we *continue* the computation. For example, consider the expression $\mathsf{if}\ x > 0\ \mathsf{then}\ x\ \mathsf{else}\ x + 1$. First we will evaluate $x > 0$ to obtain a boolean value b. Only then will we use that boolean to evaluate $\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ x + 1$. If we think of the if statement as a function that takes the result of the condition, the *continuation* would be $\lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ x + 1$.

We have seen representations of continuations before, though we did not call them that. The evaluation contexts we used to simplify the definition of evaluation order steps in small-step operational semantics were fundamentally continuations. For an expression in a context $E[e]$, the expression e represents the current computation, and $E[\cdot]$ is a representation of the continuation.

Some languages, like Scheme and its derivatives (e.g., Racket) have ways to capture and save the current continuation as a function (call/cc and let/cc). Saving that continuation and applying it in another context can have highly non-intuitive behavior, as it replaces the continuation at the point of application with the continuation that was saved. In essence, it destroys the call stack and replaces it with an old, saved one.

The most common use of continuations, however, is a transformation to *continuation passing style*.

1 Continuation Passing Style

Given an expression e, it is possible to transform it into a function that takes a continuation k and applies k to the result of evaluating e. If we apply this transformation recursively, the result is called *continuation passing style* (CPS). There are a number of advantages to CPS.

- CPS expressions have much simpler evaluation semantics. The sequence of reductions is specified by the series of continuations, so the next operation to perform is always uniquely determined and the continuation handles the rest of the computation. Evaluation contexts are therefore unnecessary to specify evaluation order. In fact, the choices we made when defining evaluation contexts are instead made in the translation to CPS.
- In practice, function calls and function returns can be handled in a uniform way. Instead of returning, the called function simply calls the continuation.
- In recursive functions, any computation performed on the value returned by a recursive call is bundled into a continuation that is handed to the recursive call. As a result, every recursive call becomes tail-recursive. For example, the factorial function

  $$\mathsf{fact}\ n \ =\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathsf{fact}\ (n - 1)$$

  becomes

  $$\mathsf{fact}'\ n\ k \ =\ \mathsf{if}\ n = 0\ \mathsf{then}\ k\ 1\ \mathsf{else}\ \mathsf{fact}'\ (n - 1)\ (\lambda x.\ k\ (n * x))$$

  It is possible to show that $\mathsf{fact}'\ n\ k = k\ (\mathsf{fact}\ n)$, and therefore $\mathsf{fact}'\ n\ \mathsf{id} = \mathsf{fact}\ n$. The transformation essentially trades stack space for heap space in the implementation.
- Continuation-passing gives a convenient mechanism for non-local flow of control, such as goto statements and exception handling.

As a result of these advantages, a variety of compiles perform CPS transformations to help with optimization and analysis of programs. We will see how a simple one works now.

2 CPS Semantics

In pure λ-calculus, our grammar was

$$e \quad ::= \quad x \mid \lambda x.\, e \mid e_1\, e_2.$$

Our grammar for CPS λ-calculus will be slightly different to account for the fact that all computations must be bundled into continuations. It is defined as follows.

$$v \quad ::= \quad x \mid \lambda x.\, e \qquad\qquad e \quad ::= \quad v \mid e\, v$$

This is a *highly* constrained syntax. Barring reductions inside of λ-abstractions, the values v are all irreducible. The only reducible expressions are of the form $e\, v$. Moreover, there is only one possible redex: the inner-most e must be $v_0\, v_1$, and both the function and argument are already fully reduced. This means that we do not need any interesting evaluation order rules, we can get away with a simple structural one with no choices and a single reduction rule for our small-step operationsl semantics:

$$\frac{e \longrightarrow e'}{e\, v \longrightarrow e'\, v} \qquad\qquad \frac{}{(\lambda x.\, e)\, v \longrightarrow e[x \mapsto v]}$$

A proof that $e \longrightarrow^* v$ only has one possible shape with this semantics, no matter what e is. It just applies the same operation over and over again. This fact allows for a much simpler interpreter that can work in a straight line rather than having to make multiple recursive calls.

Indeed, the fact that it would be so simple to implement an interpreter is an indication the CPS λ-calculus is, in a deep sense, a lower-level language than CBV λ-calculus. Because it is lower-level (and actually closer to assembly code), CPS is typically used in functional language compilers as an intermediate representation. It also is a good code representation if one is building an interpreter.

3 CPS Conversion

Despite the restrictions of CPS syntax, we have not lost any expressive power. We can define a translation $[\![\cdot]\!]$ to take a regular λ-term e and produce a CPS term $[\![e]\!]$ with the same meaning. This is known as *CPS conversion* and was first described by John C. Reynolds (1935–2013).

Recall that a CPS term is a function that takes a continuation k as an argument, so we would like our translation to satisfy

$$e \underset{\text{CBV}}{\longrightarrow^*} v \quad \Longleftrightarrow \quad [\![e]\!]\ k \underset{\text{CPS}}{\longrightarrow^*} [\![v]\!]\ k$$

for any primitive value v and any variable $k \notin \mathrm{FV}(e)$.

If we allow numbers as primitive values and add simple arithmetic operators, which we denote $\otimes$, then the transformation is as follows. Recall that $[\![e]\!]\ k \triangleq e'$ is short-hand for $[\![e]\!] \triangleq \lambda k.\, e'$.

$$\begin{aligned}
[\![n]\!]\ k &\triangleq k\, n \\
[\![x]\!]\ k &\triangleq k\, x \\
[\![\lambda x.\, e]\!]\ k &\triangleq k\ (\lambda x k'.\, [\![e]\!]\ k') =_\eta k\ (\lambda x.\, [\![e]\!]) \\
[\![e_1 \otimes e_2]\!]\ k &\triangleq [\![e_1]\!]\ (\lambda x_1.\, [\![e_2]\!]\ (\lambda x_2.\, k\ (x_1 \otimes x_2))) \\
[\![e_1\, e_2]\!]\ k &\triangleq [\![e_1]\!]\ (\lambda f.\, [\![e_2]\!]\ (\lambda x.\, f\ x\ k))
\end{aligned}$$

In this translation, we transform a λ-abstraction $\lambda x.\, e$ that takes one input, a value x, to a λ-abstraction $\lambda x k'.\, [\![e]\!]\ k'$ that takes two inputs: the same value x and a continuation k'. Note that k and k' are not the same. The continuation k' is supplied to $[\![e]\!]$ at the point of the function call, while the continuation k is applied to the translated λ-abstraction itself where the function is defined.

3.1 An Example

In CBV λ-calculus, we have

$$(\lambda xy.\, x)\ 1 \longrightarrow \lambda y.\, 1$$

The CPS translations of those two are:

$$\begin{aligned}
[\![(\lambda xy.\, x)\ 1]\!]\ k &= [\![\lambda x.\, \lambda y.\, x]\!]\ (\lambda f.\, [\![1]\!]\ (\lambda v.\, f\ v\ k)) \\
&= (\lambda f.\, [\![1]\!]\ (\lambda v.\, f\ v\ k))\ (\lambda x k'.\, [\![\lambda y.\, x]\!]\ k') \\
&= (\lambda f.\, (\lambda v.\, f\ v\ k)\ 1)\ (\lambda x k'.\, k'\ (\lambda y.\, [\![x]\!])) \\
&= (\lambda f.\, (\lambda v.\, f\ v\ k)\ 1)\ (\lambda x k'.\, k'\ (\lambda y k''.\, k''\ x))
\end{aligned}
\qquad
\begin{aligned}
[\![\lambda y.\, 1]\!]\ k &= k\ (\lambda y k'.\, [\![1]\!]\ k') \\
&= k\ (\lambda y k'.\, k'\ 1)
\end{aligned}$$

The translation of the value is itself k applied to a value, so there is nothing to do. We can, however, evaluate the right side, producing the following sequence.

$$\begin{aligned}
(\lambda f.\, (\lambda v.\, f\ v\ k)\ 1)\ (\lambda x k'.\, k'\ (\lambda y k''.\, k''\ x)) &\longrightarrow (\lambda v.\, (\lambda x k'.\, k'\ (\lambda y k''.\, k''\ x))\ v\ k)\ 1 \\
&\longrightarrow (\lambda x k'.\, k'\ (\lambda y k''.\, k''\ x))\ 1\ k \\
&\longrightarrow (\lambda k'.\, k'\ (\lambda y k''.\, k''\ 1))\ k \\
&\longrightarrow k\ (\lambda y k''.\, k''\ 1) \\
&=_\alpha [\![\lambda y.\, 1]\!]\ k
\end{aligned}$$

This is precisely the result we were hoping for. Also note that, in every step of that evaluation, the leftmost term was already a λ-abstraction, and each term in an argument position was already a value. There was never any choice of which steps to take, it was always just applying a continuation to the value produced by the previous operation.