1 Continuation-Passing Style

Consider the statement $\mathsf{if}\ x \le 0\ \mathsf{then}\ x\ \mathsf{else}\ x+1$. We can think of this as $(\lambda y.\, \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1)\,(x \le 0)$. To evaluate this, we would first evaluate the argument $x \le 0$ to obtain a Boolean value, then apply the function $\lambda y.\, \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1$ to this value. The function $\lambda y.\, \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1$ is called a *continuation*, because it specifies what is to be done with the result of the current computation in order to continue the computation.

Given an expression e, it is possible to transform the expression into a function that takes a continuation k and applies it to the value of e. The transformation is applied recursively. This is called *continuation-passing style* (CPS). There are a number of advantages to this style:

- The resulting expressions have a much simpler evaluation semantics, since the sequence of reductions to be performed is specified by a series of continuations. The next reduction to be performed is always uniquely determined, and the remainder of the computation is handled by a continuation. Thus evaluation contexts are not necessary to specify the evaluation order.
- 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 a recursive function, any computation to be performed on the value returned by a recursive call can be bundled into the continuation. Thus every recursive call becomes tail-recursive. For example, the factorial function

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

  becomes

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

  One can show inductively that $\mathsf{fact}'\, n\, k = k\,(\mathsf{fact}\, n)$, therefore $\mathsf{fact}'\, n\, \lambda x.\, x = \mathsf{fact}\, n$. This transformation effectively 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.

2 CPS Semantics

Our grammar for the λ-calculus was:

$$e \quad ::= \quad x \ \mid\ \lambda x.\, e \ \mid\ e_0\, e_1$$

Our grammar for the CPS λ-calculus will be:

$$v \quad ::= \quad x \ \mid\ \lambda x_0, x_1, \ldots, x_n.\, e \qquad\qquad e \quad ::= \quad v_0\, v_1\, \cdots\, v_n$$

This is a highly constrained syntax. Barring reductions inside the scope of a λ-abstraction operator, the expressions v are all irreducible. The only reducible expression is $v_0\, v_1\, \cdots\, v_n$. If $n \ge 1$, there exactly one redex $v_0\, v_1$, and both the function and the argument are already fully reduced. The small step semantics has a single rule

$$(\lambda x_0, x_1, \ldots, x_n.\, e)\, v_0\, v_1\, \cdots\, v_n \quad \to \quad e\{v_o/x_i\}^{(i \in 1..n)},$$

and we do not need any evaluation contexts.

The big step semantics is also quite simple, with only a single rule:

$$\frac{e\{v_i/x_i\}^{(i\in 1..n)} \Downarrow v'}{(\lambda x_0, x_1, \ldots, x_n.\, e)\, v_0\, v_1\, \cdots\, v_n \Downarrow v'}.$$

The resulting proof tree will not be very tree-like. The rule has one premise, so a proof will be a stack of inferences, each one corresponding to a step in the small-step semantics. This allows for a much simpler interpreter that can work in a straight line rather than having to make multiple recursive calls.

The fact that we can build a simpler interpreter for the language is a strong hint that this language is lower-level than the 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 restricted syntax of CPS, we have not lost any expressive power. Given a λ-calculus expression e, it is possible to define a translation $[\![e]\!]$ that translates it into CPS. This translation is known as *CPS conversion*. It was first described by John Reynolds. The translation takes an arbitrary λ-term e and produces a CPS term $[\![e]\!]$, which is a function that takes a continuation k as an argument. Intuitively, $[\![e]\!]\,k$ applies k to the value of e.

We want our translation to satisfy $e \xrightarrow[\text{CBV}]{*} v$ iff $[\![e]\!]\,k \xrightarrow[\text{CPS}]{*} [\![v]\!]\,k$ for primitive values v and any variable $k \notin FV(e)$, and $e \Uparrow_{\text{CBV}}$ iff $[\![e]\!]\,k \Uparrow_{\text{CPS}}$.

The translation is (adding numbers as primitive values):

$$\begin{aligned}
[\![n]\!]\,k &\triangleq k\,n \\
[\![x]\!]\,k &\triangleq k\,x \\
[\![\lambda x.\, e]\!]\,k &\triangleq k\,(\lambda x.\, [\![e]\!]) \quad = \quad k\,(\lambda x k'.\, [\![e]\!]\,k') \\
[\![e_0\, e_1]\!]\,k &\triangleq [\![e_0]\!](\lambda f.\, [\![e_1]\!](\lambda v.\, f v k)).
\end{aligned}$$

(Recall $[\![e]\!]\,k \triangleq e'$ really means $[\![e]\!] \triangleq \lambda k.\, e'$.)

3.1 An Example

In the CBV λ-calculus, we have

$$(\lambda xy.\, x)\, 1 \quad \rightarrow \quad \lambda y.\, 1$$

Let's evaluate the CPS-translation of the left-hand side using the CPS evaluation rules.

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

4 CPS Semantics for FL!

4.1 Syntax

Since FL! has references, we need to add a store σ to our notation. Thus we now have translations with the form $\mathcal{E}[\![e]\!]\,\rho\,k\,\sigma$, which means, "Evaluate e in the environment ρ with store σ and send the resulting value and the new store to the continuation k." A continuation is now a function of a value and a store; that is, a continuation k should have the form $\lambda v\sigma.\,\cdots$.

The translation for variables is as follows:

$$\mathcal{E}[\![x]\!]\,\rho\,k\,\sigma \triangleq k\,(\mathsf{lookup}\,\rho\,x)\,\sigma$$

Note that if we think about this translation as a function and η-reduce away the σ, we obtain:

$$\mathcal{E}[\![x]\!]\,\rho\,k \quad = \quad \lambda\sigma.\,k\,(\mathsf{lookup}\,\rho\,x)\,\sigma \quad = \quad k\,(\mathsf{lookup}\,\rho\,x).$$

In particular, in the η-reduced version, we have the same translation that we had for FL. In general, any expression in FL! that is not state-aware can be η-reduced to the same translation as FL. Thus in order to translate to FL!, we need to add translation rules only for the functionality that is state-aware.

Assume that we have extended our lookup and update functions to apply to stores. The translations of the state-aware constructs in FL! are defined as follows:

$$\begin{aligned}
\mathcal{E}[\![\mathsf{ref}\,e]\!]\,\rho k &\triangleq \mathcal{E}[\![e]\!]\,\rho\,(\lambda v\sigma'.\,\mathsf{let}\ (\ell,\sigma'') = (\mathsf{malloc}\,\sigma'\,v)\ \mathsf{in}\ k\,(Loc\,\ell)\,\sigma'')\\
\mathcal{E}[\![!e]\!]\,\rho k &\triangleq \mathcal{E}[\![e]\!]\,\rho\,(\lambda\ell\sigma'.\,k\,(\mathsf{lookup}\,\sigma'\,\ell)\,\sigma')\\
\mathcal{E}[\![e_1 := e_2]\!]\,\rho k &\triangleq \mathcal{E}[\![e_1]\!]\,\rho\,(\lambda\ell.\,\mathcal{E}[\![e_2]\!]\,\rho(\lambda v\sigma'.\,k\,\mathsf{null}\,(\mathsf{update}\,\sigma'\,\ell\,v)))
\end{aligned}$$