Tony Faradjian

We now introduce continuations into our $F!$ language. Whereas before the order of evaluation used to be implicit, it will now be *explicit*. Recall our earlier compilation functions. Each of them had the form

$$\mathcal{C}[\![e]\!] \;:\; \mathit{Context} \rightarrow \mathit{Result}$$

Here they are in more detail:

$$\begin{array}{rcl}
F_0 & : & \mathit{Context} = \mathit{Env} \\
 & & \mathit{Result} = (\mathit{Value} + \mathit{Error})_\perp \\
F_{0(cbn)} & : & \mathit{Env} = \mathit{Var} \rightarrow \mathit{Result} \\
F_{0(cbv)} & : & \mathit{Env} = \mathit{Var} \rightarrow \mathit{Value} \\
F! & : & \mathit{Context} = \mathit{Env} \times \mathit{Store} \\
 & & \mathit{Result} = ((\mathit{Value} + \mathit{Error}) \times \mathit{Store})_\perp
\end{array}$$

where "cbv" means "call by value" and "cbn" means "call by name". The old semantics are called *direct* semantics, because they translate control flow in the language directly into the equivalent metalanguage constructs. The new semantics will expose control explicitly in a semantics based on continuations. This kind of semantics is known as a *standard semantics*.

Think of the continuation as "the rest of the computation". Just as ρ is the naming context, and σ is the store context, we'll now have a *control context* gotten be extending by adding k, the continuation (we'll curry our functions from now on):

$$\begin{array}{rcl}
\mathcal{C}[\![e]\!] & : & \mathit{Env} \rightarrow \mathit{Cont} \rightarrow \mathit{Store} \rightarrow \mathit{Result} \\
 & & \mathit{Context} = \mathit{Env} \rightarrow \mathit{Cont} \rightarrow \mathit{Store} \\
 & & \mathit{Result} = ((\mathit{Value} + \mathit{Error}) \times \mathit{Store})_\perp \\
 & & \mathit{Cont} = (\mathit{Value} + \mathit{Error}) \rightarrow \mathit{Store} \rightarrow \mathit{Result} \\
 & & \mathit{Store} = \mathit{Loc} \rightarrow \mathit{Value} \\
 & & \mathit{Env} = \mathit{Var} \rightarrow \mathit{Value} \\
 & & \mathit{Function} = \mathit{Value} \rightarrow \mathit{Cont} \rightarrow \mathit{Store} \rightarrow \mathit{Result}
\end{array}$$

The meaning of a program p is:

$$\mathcal{T}[\![p]\!] = \mathcal{C}[\![p]\!]\rho_0\sigma_0 k_0$$

where ρ_0 is the empty environment, σ_0 is the empty store, and k_0 is the top-level continuation, also known as the "output continuation" since it simply outputs whatever its given, like an identity function:

$$k_0 = \lambda v\sigma \,.\, \langle v, \sigma \rangle$$

Below we use the family of functions *check–X* which check for type errors:

$$\begin{array}{rcl}
\mathit{check\text{–}X}(k) & : & \mathit{Cont} \rightarrow X \rightarrow \mathit{Store} \\
\mathit{check\text{–}value}(k) & = & \lambda v\sigma \,.\, \text{case } v \text{ of } \mathit{value}(v') \Rightarrow kv\sigma \\
 & & \qquad\qquad\qquad\quad \mid \mathit{error}(e) \Rightarrow k_0 v\sigma
\end{array}$$

Without further ado, here are the semantics (note that η-reductions have been used extensively for σ, and, in fact, they could be used even more for ρ if the arguments to $\mathcal{C}[\![e]\!]$ were reordered to put the continuation first).

$$
\begin{aligned}
\mathcal{C}[\![n]\!]\rho k &= \lambda\sigma\ .\ kn\sigma = kn \\[1em]
\mathcal{C}[\![x]\!]\rho k &= k(\rho x) \\[1em]
\mathcal{C}[\![(\mathit{if}\ \ e_0\ e_1\ e_2)]\!]\rho k &= \mathcal{C}[\![e_0]\!]\rho(\mathit{check\text{--}boolean}(\lambda b\,.\,\text{case } b \text{ of} \\
&\qquad\qquad \text{true} \Rightarrow \mathcal{C}[\![e_1]\!]\rho k \\
&\qquad\qquad \text{false} \Rightarrow \mathcal{C}[\![e_2]\!]\rho k)) \\[1em]
\mathcal{C}[\![(\mathit{cons}\ e_0\ e_1)]\!]\rho k &= \mathcal{C}[\![e_0]\!]\rho(\mathit{check\text{--}value} \\
&\qquad\qquad (\lambda v.\mathcal{C}[\![e_1]\!]\rho(\mathit{check\text{--}value}(\lambda v_1.k\langle v_0, v_1\rangle))))) \\[1em]
\mathcal{C}[\![(\mathit{call}\ e_0\ e_1)]\!]\rho k &= \mathcal{C}[\![e_0]\!]\rho(\mathit{check\text{--}function}(\lambda f.\mathcal{C}[\![e_1]\!])\rho.(\mathit{check\text{--}value}(\lambda v.fvk))) \\[1em]
\mathcal{C}[\![(\mathit{fn}\ x\ e)]\!]\rho k &= k(\lambda v k' \mathcal{C}[\![e]\!]\rho[x \mapsto v]k') \\[1em]
\mathcal{C}[\![(\mathit{ref}\ \ e)]\!]\rho k &= \mathcal{C}[\![e]\!]\rho(\mathit{check\text{--}value}(\lambda v\sigma'.\text{let } l = \text{alloc}(\sigma') \text{ in } kl\sigma'[l \mapsto v])) \\[1em]
\mathcal{C}[\![(\hat{}\ e)]\!]\rho k &= \mathcal{C}[\![e]\!]\rho(\mathit{check\text{--}location}(\lambda l\sigma'.k(\sigma' l)\sigma')) \\[1em]
\mathcal{C}[\![(e_1 := e_2)]\!]\rho k &= \mathcal{C}[\![e_1]\!]\rho(\mathit{check\text{--}location}(\lambda l.\mathcal{C}[\![e_2]\!]\rho(\mathit{check\text{--}value}(\lambda v\sigma'.kv(\sigma'[l \mapsto v])))))
\end{aligned}
$$

Using our new semantics, we can define nifty *label* and *break* expressions, as follows.

$$
\begin{aligned}
\mathcal{C}[\![(\mathit{label}\ x\ e)]\!]\rho k &= \mathcal{C}[\![e]\!]\rho[x \mapsto k] \\
\mathcal{C}[\![(\mathit{break}\ x\ e)]\!]\rho k &= \mathcal{C}[\![x]\!]\rho(\mathit{check\text{--}cont}(\lambda k'.\mathcal{C}[\![e]\!]\rho(\mathit{check\text{--}value}(\lambda v.k'v))))
\end{aligned}
$$

These expressions are similar to the `setjmp` and `longjmp` calls of the standard C library. The *label* expression evaluates e while binding the current continuation to the variable x. The *break* expression jumps out to the end of the enclosing *label* expression whose label is x. This mechanism is reminiscent of the Java labeled `break` statement, which jumps to the appropriately labeled statement (Here the label precedes the statement rather than following it, as in Java.) This mechanism is more powerful, though, because the label is actually placed into the variable named x and can be manipulated as a first-class value. In order to make this possible, we need to allow continuations as a member of $Value$.