1 Continuation-Passing Style

Consider the statement $\mathsf{if}\ x \leq 0\ \mathsf{then}\ x\ \mathsf{else}\ x+1$. We can think of this as $(\lambda y.\ \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1)\,(x \leq 0)$. To evaluate this, we would first evaluate the argument $x \leq 0$ 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:

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

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 $n \geq 1$, 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

$$(\lambda x.\, e)\, v \quad \to \quad e\{v/x\},$$

and we do not need any evaluation contexts.

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

$$\frac{e\{v/x\} \Downarrow v'}{(\lambda x.\, e)\, v \Downarrow v'}.$$

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):

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

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

3.1 An Example

In the CBV λ-calculus, we have

$$(\lambda x y.\, x)\, 1 \quad \to \quad \lambda y.\, 1$$

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

$$\begin{array}{rcl}
[\![(\lambda x y.\, x)\, 1]\!]\,k & = & [\![\lambda x.\, \lambda y.\, x]\!](\lambda f.\, [\![1]\!](\lambda v.\, f v k)) \\
& = & (\lambda f.\, [\![1]\!](\lambda v.\, f v k))\, (\lambda x.\, [\![\lambda y.\, x]\!]) \\
& \to & [\![1]\!](\lambda v.\, (\lambda x.\, [\![\lambda y.\, x]\!])\, v\, k) \\
& = & (\lambda v.\, (\lambda x.\, [\![\lambda y.\, x]\!])\, v\, k)\, 1 \\
& \to & (\lambda x.\, [\![\lambda y.\, x]\!])\, 1\, k \\
& \to & [\![\lambda y.\, 1]\!]\,k.
\end{array}$$

4 CPS and Strong Typing

Now let us use CPS semantics to augment our previously defined FL language translation so that it supports runtime type checking. This time our translated expressions will be functions of ρ and k denoting an environment and a continuation, respectively. The term $\mathcal{E}[\![e]\!]\rho k$ represents a program that evaluates e in the environment ρ and sends the resulting value to the continuation k.

As before, assume that we have an encoding of variable names x and a representation of environments ρ along with lookup and update functions $\mathsf{lookup}\ \rho\ \text{“}x\text{”}$ and $\mathsf{update}\ \rho\ v\ \text{“}x\text{”}$.

In addition, we want to catch type errors that may occur during evaluation. As before, we use integer tags to keep track of types:

$$\mathsf{Err} \triangleq 0 \qquad \mathsf{Null} \triangleq 1 \qquad \mathsf{Bool} \triangleq 2 \qquad \mathsf{Num} \triangleq 3 \qquad \mathsf{Pair} \triangleq 4 \qquad \mathsf{Fun} \triangleq 5$$

A *tagged value* is a value paired with its type tag; for example, $(\mathsf{Bool}, \mathsf{true})$. Using these tagged values, we can now define a translation that incorporates runtime type checking:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho k &\triangleq k\,(\mathsf{lookup}\ \rho\ \text{“}x\text{”}) \\
\mathcal{E}[\![b]\!]\rho k &\triangleq k\,(\mathsf{Bool}, b) \\
\mathcal{E}[\![n]\!]\rho k &\triangleq k\,(\mathsf{Num}, n) \\
\mathcal{E}[\![()]\!]\rho k &\triangleq k\,(\mathsf{Null}, \mathsf{nil}) \\
\mathcal{E}[\![(e_1, \ldots, e_n)]\!]\rho k &\triangleq \mathcal{E}[\![e_1]\!]\rho(\lambda x_1.\, \mathcal{E}[\![(e_2, \ldots, e_n)]\!]\rho(\lambda x_2.\, k\,(\mathsf{Pair}, (x_1, x_2)))),\ n \geq 1 \\
\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho k &\triangleq \mathcal{E}[\![e_1]\!]\rho(\lambda p.\, \mathcal{E}[\![e_2]\!](\mathsf{update}\ \rho\ p\ \text{“}x\text{”})\,k) \\
\mathcal{E}[\![\lambda x.\, e]\!]\rho k &\triangleq k\,(\mathsf{Fun}, \lambda v k'.\, \mathcal{E}[\![e]\!](\mathsf{update}\ \rho\ v\ \text{“}x\text{”})\,k') \\
\mathcal{E}[\![\mathsf{error}]\!]\rho k &\triangleq k\,(\mathsf{Err}, 0).
\end{aligned}$$

Now a function application can check that it is actually applying a function:

$$\mathcal{E}[\![e_0\, e_1]\!]\rho k \triangleq \mathcal{E}[\![e_0]\!]\rho(\lambda p.\, \mathsf{let}\ (t, f) = p\ \mathsf{in}\ \mathsf{if}\ t \neq \mathsf{Fun}\ \mathsf{then}\ k\,(\mathsf{Err}, 0)\ \mathsf{else}\ \mathcal{E}[\![e_1]\!]\rho(\lambda v.\, f v k))$$

We can simplify this by defining a helper function check-fn:

$$\mathsf{check\text{-}fn} \triangleq \lambda k p.\, \mathsf{let}\ (t, f) = p\ \mathsf{in}\ \mathsf{if}\ t \neq \mathsf{Fun}\ \mathsf{then}\ k\,(\mathsf{Err}, 0)\ \mathsf{else}\ k f.$$

The helper function takes in a continuation and a tagged value, checks the type, strips off the tag, and passes the raw (untagged) value to the continuation. Then

$$\mathcal{E}[\![e_0\, e_1]\!]\rho k \triangleq \mathcal{E}[\![e_0]\!]\rho(\mathsf{check\text{-}fn}\,(\lambda f.\, \mathcal{E}[\![e_1]\!]\rho(\lambda v.\, f v k))).$$

Similarly,

$$\begin{aligned}
\mathcal{E}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho k &\triangleq \mathcal{E}[\![e_0]\!]\rho(\mathsf{check\text{-}bool}\,(\lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ \mathcal{E}[\![e_1]\!]\rho k\ \mathsf{else}\ \mathcal{E}[\![e_2]\!]\rho k)) \\
\mathcal{E}[\![\#n\, e]\!]\rho k &\triangleq \mathcal{E}[\![e]\!]\rho(\mathsf{check\text{-}pair}\,(\lambda t.\, k\,(\#n\, t))).
\end{aligned}$$

5 CPS Semantics for FL!

5.1 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 is:

- Variable: $\mathcal{E}[\![x]\!]\,\rho\,k\,\sigma \triangleq k\,(\mathsf{lookup}\ \rho\ \text{“}x\text{”})\,\sigma$.

  If we think about this translation as a function and η-reduce away the σ, we obtain

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

Note that in the η-reduced version, we have the same translation that we 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.

We assume that we have a type tag Loc for locations and check-loc for tagging values as locations and checking those tags. We also assume that we have extended our lookup and update functions to apply to stores.

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

6 Exceptions

An exception mechanism allows non-local transfer of control in exceptional situations. It is typically used to handle abnormal, unexpected, or rarely occurring events. It can simplify code by allowing programmers to factor out these uncommon cases. OCaml also uses them for not-found conditions when searching lists and similar data structures, a questionable design decision; Standard ML uses option for this purpose.

To add an exception handling mechanism to FL, we first extend the syntax:

$$e \;::=\; \ldots \;\mid\; \mathsf{raise}\,s\,e \;\mid\; \mathsf{try}\,e_1\,\mathsf{handle}\,(s\,x)\,e_2$$

Informally, the idea is that handle provides a handler e_2 to be invoked when the exception named s is encountered inside the expression e_1. To raise an exception, the program calls raise $s\,e$, where s is the name of an exception and e is an expression that will be passed to the handler as its argument x.

Most languages use a dynamic scoping mechanism to find the handler for a given exception. When an exception is encountered, the language walks up the runtime call stack until a suitable exception handler is found.

6.1 Exceptions in FL

To add support for exceptions to our CPS translation, we add a *handler environment* h, which maps exception names to continuations. We also extend our lookup and update functions to accommodate handler environments. Applied to a handler environment, lookup returns the continuation bound to a given exception name, and update rebinds an exception name to a new continuation.

Now we can add support for exceptions to our translation:

$$\begin{aligned}
\mathcal{E}[\![\mathsf{raise}\,s\,e]\!]\,\rho\,k\,h &\triangleq \mathcal{E}[\![e]\!]\,\rho\,(\mathsf{lookup}\ h\ \text{“}s\text{”})\,h\\
\mathcal{E}[\![\mathsf{try}\,e_1\,\mathsf{handle}\,(s\,x)\,e_2]\!]\,\rho\,k\,h &\triangleq \mathcal{E}[\![e_1]\!]\,\rho\,k\,(\mathsf{update}\ h\ (\lambda v.\,\mathcal{E}[\![e_2]\!]\,(\mathsf{update}\ \rho\ v\ \text{“}x\text{”})\,k\,h)\ \text{“}s\text{”})\\
\mathcal{E}[\![\lambda x.\,e]\!]\,\rho\,k\,h &\triangleq k\,(\mathsf{tag\text{-}fun}\,(\lambda v k' h'.\,\mathcal{E}[\![e]\!]\,(\mathsf{update}\ \rho\ v\ \text{“}x\text{”})\,k'\,h'))\\
&= k\,(\mathsf{tag\text{-}fun}\,(\lambda v.\,\mathcal{E}[\![e]\!]\,(\mathsf{update}\ \rho\ v\ \text{“}x\text{”})))\\
\mathcal{E}[\![e_0\,e_1]\!]\,\rho\,k\,h &\triangleq \mathcal{E}[\![e_0]\!]\,\rho\,(\mathsf{check\text{-}fun}\,(\lambda f.\,\mathcal{E}[\![e_1]\!]\,\rho\,(\lambda v.\,f v k h)\,h))\,h
\end{aligned}$$

where tag-fun tags a function value with its runtime type.

There are some subtle design decisions captured by this translation. For example, note that in try... handle, x is in scope in e_2, but s is not. Thus if e_2 attempts to raise exception s in $\mathsf{try}\, e_1\, \mathsf{handle}\, (s\, x)\, e_2$, in this translation e_2 will not be invoked again. That is, e_2 cannot be invoked recursively.

6.2 Exceptions with Resumption

The exception mechanism above has the property that raising an exception terminates execution of the evaluation context. Most modern programming languages have exceptions with this *termination semantics*. A different approach to exceptions is to allow execution to continue at the point where the exception was raised, after the exception handler gets a chance to repair the damage. This approach is known as exceptions with *resumption semantics*. In practice it seems to be difficult to use these mechanisms usefully. The Cedar/Mesa system supported both kinds of exceptions and found that resumption-style exceptions were almost never used, and often resulted in bugs when they were.

Operating system interrupts are one place where resumption semantics can be seen. When a process receives an interrupt, the interrupt handler is run, and then execution continues at the point in the program where the interrupt happened.

We can give a translation that captures the semantics of resumption-style exceptions. We add two constructs to FL:

$$e ::= \mathsf{interrupt}\, s\, e \;\mid\; \mathsf{try}\, e_1\, \mathsf{handle}\, (s\, x)\, e_2$$

The translation makes the exception-handling environment h a mapping from exception names to *functions* rather than to continuations:

$$\begin{aligned} [\![\mathsf{interrupt}\, s\, e]\!]\, \rho\, k\, h &= [\![e]\!]\, \rho\, (\lambda v.\, (\mathsf{lookup}\ h\ \text{“}s\text{”})\, v\, k)\, h \\ [\![\mathsf{try}\, e_1\, \mathsf{handle}\, (s\, x)\, e_2]\!]\, \rho\, k\, h &= [\![e_1]\!]\, \rho\, k\, (\mathsf{update}\ h\ (\lambda v k'.\, [\![e_2]\!]\, \rho\, k'\, h)\ \text{“}s\text{”}) \end{aligned}$$

This translation shows that with resumption semantics, the exception handler is really a dynamically bound function that is invoked at the point where the exception happens.