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 if $x > 0$ then x else $x + 1$. First we will evaluate $x > 0$ to obtain a boolean value b. Only then will we use that boolean to evaluate if b then x else $x + 1$. If we think of the if statement as a function that takes the result of the condition, the *continuation* would be $\lambda b.\ \text{if } b \text{ then } x \text{ else } x + 1$.

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

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

  becomes

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

  It is possible to show that $\text{fact}' \; n \; k = k \; (\text{fact } n)$, and therefore $\text{fact}' \; n \; \text{id} = \text{fact } n$. 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 we will see later.

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.

1.1 CPS Semantics

In pure λ-calculus, our grammar was

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\, e \mid e_1\ e_2 \\ v & ::= & \lambda x.\, e. \end{array}$$

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.

$$\begin{array}{rcl} e & ::= & v \mid e\ v \\ v & ::= & x \mid \lambda x.\, e \end{array}$$

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.

1.2 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 $k \notin \text{FV}(e)$.

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 $[\![e]\!] \triangleq \lambda k.\, e'$.

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

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.

1.3 An Example

In CBV λ-calculus, we have

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

The CPS translations of those two are:

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

$$\begin{aligned}
[\![\lambda y.\, 1]\!]\ k &= k\ (\lambda yk'.\, [\![1]\!]\ k') \\
&= k\ (\lambda yk'.\, k'\ 1)
\end{aligned}$$

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.

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

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.

2 Exceptions

Exceptions are a language feature that provide for non-local control flow in exceptional situations. They are generally considered a double-edged sword from a software engineering and maintenance perspective: exceptional control flow is harder to understand and reason about (and thus maintain), but factoring out some control flow into an exceptional path often makes it easier to understand and maintain the other control flow paths. As a result, exceptions are typically used to signify and handle abnormal, unexpected, or rarely occurring events and simplify code for the common cases.

To add exceptions to FL, we extend the syntax with two new constructs: raise and try–catch.

$$e \quad ::= \quad \cdots \mid \mathsf{raise}\ e \mid \mathsf{try}\ e_1\ \mathsf{catch}\ (\lambda x.\, e_2)$$

Informally, raise e throws an exception with value e. Meanwhile, try e_1 catch $(\lambda x.\, e_2)$ provides the handler $\lambda x.\, e_2$ for any exception raised while executing e_1. In other words, if e_1 terminates normally with value v, the result of try e_1 catch $(\lambda x.\, e_2)$ will also be v. If it raises an exception, the handler $\lambda x.\, e_2$ will be invoked and provided the value of that exception. Note that it is relatively simple to generalize this to multiple types of exceptions that are thrown and caught separately.

Most languages use a dynamic scoping mechanism to find the handler for a given exception. When an exception occurs, the language walks up the runtime call stack until it finds a suitable exception handler. We will take the same appraoch in FL and define it both using operational semantics and a modification to the CPS translation.

2.1 Operational Semantics of Exceptions

One way to formalize the definition of exceptions is to define a small-step operational semantics. Here we will extend the semantics for FL to define the semantics for our new exception terms.

First, we extend our evaluation contexts as follows:

$$E \quad ::= \quad \cdots \mid \mathsf{raise}\ E.$$

Note that we do not include try–catch statements in our evaluation contexts. This is because we want to use our evaluation contexts to succinctly define how exceptions propagate. In particular, we can define the following exception propagation rule.

$$[\textsc{Raise}]\ \frac{E \neq [\cdot]}{E[\mathsf{raise}\ v] \longrightarrow \mathsf{raise}\ v}$$

This rule is what creates the non-local control flow described earlier. Any evaluation context surrounding an exception is simply destroyed without executing any pending computations it specifies. The side condition that $E \neq [\cdot]$ is a technical requirement to prevent and infinite sequence of reduction steps that do nothing by reducing $\mathsf{raise}\ s\ v$ to itself.

Note that, if we included the body of a try–catch statement as an evaluation context, RAISE would bypass the handlers and propagate exceptions in ways it should not. Instead, we include an explicit evaluation order rule along with the other semantic rules for try–catch.

$$[\textsc{TryE}]\ \frac{e_1 \longrightarrow e_1'}{\mathsf{try}\ e_1\ \mathsf{catch}\ (\lambda x.\, e_2) \longrightarrow \mathsf{try}\ e_1'\ \mathsf{catch}\ (\lambda x.\, e_2)} \qquad [\textsc{TryV}]\ \frac{}{\mathsf{try}\ v\ \mathsf{catch}\ (\lambda x.\, e) \longrightarrow v}$$

$$[\textsc{Catch}]\ \frac{}{\mathsf{try}\ (\mathsf{raise}\ v)\ \mathsf{catch}\ (\lambda x.\, e) \longrightarrow e[x \mapsto v]}$$

The first rule (TRYE) is a simple evaluation order rule. The second rule (TRYV) says that if the body of the try–catch block terminates normally with a value, there is no catching to be done and the block should return the same value. The other two rules address the case where the body e_1 raises an uncaught exception. The third rule (CATCH) handles exceptions. If the body raises an exception, it will apply the handler function the value of the exception and execute that.

There is a slightly subtle decision hiding in the CATCH rule. Because CATCH discards the try–catch and steps to only the body of the handler, if the handler body itself throws an exception, it will not recursively handle itself (though a different handler in a larger try–catch block might).

Note also that we did not change our definition of values, so, in particular raise v is *not* a value. However, there is no semantic rule for what to do with a top-level raise expression. These globally uncaught exceptions are considered errors, so the semantics simply gets stuck.

2.2 CPS and Exception Handlers

Another way to cleanly define the semantics of raise and try–catch is to extend the CPS conversion in Section 1.2 with an exception handler h. We write this extended translation $\mathcal{E}[\![e]\!]$, and note that it takes both a continuation, as before, and a new exception handler. For most of the expressions in FL, this new translation looks the same as standard CPS conversion, but with h being passed in to nearly everything. Here are the cases

for some representative expressions. Changes from the CPS conversion in Section 1.2 are highlighted in red.

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

The cases for raise and try–catch make interesting use of h.

$$
\begin{aligned}
\mathcal{E}[\![\mathsf{raise}\ e]\!]\ k\ h &\triangleq \mathcal{E}[\![e]\!]\ h\ h \\
\mathcal{E}[\![\mathsf{try}\ e_1\ \mathsf{catch}\ (\lambda x.\, e_2)]\!]\ k\ h &\triangleq \mathcal{E}[\![e_1]\!]\ k\ (\lambda x.\, \mathcal{E}[\![e_2]\!]\ k\ h)
\end{aligned}
$$

The translation of raise e simply evaluates e and passes the result as an argument to the handler h. The translation of try e_1 catch $(\lambda x.\, e_2)$ makes a new continuation by translating the handler $\lambda x.\, e_2$ with the current continuation k and outer handler h, and then updates h to map the specified name s to that new continuation. It then runs the body e_1 with the same continuation passed to the try–catch block and this new updated handler environment.

There are some subtle decisions captured by this translation. First, in the translation of try e_1 catch $(\lambda x.\, e_2)$, we match the decision we made in the operational semantics in Section 2.1 and do not allow the try–catch to recursively handle exceptions thrown in e_2. Second, in the translation of raise e, the continuation k disappears completely. That means that whatever computation is pending will simply be ignored and we will instead execute the exception handler—which will include the continuation from outside of the corresponding try–catch block. This behavior also matches something we saw in Section 2.1: the Raise rule destroyed the evaluation context $E[\cdot]$, which corresponds to the continuation k.