

## CONTINUATION PASSING STYLE

After CPS conversion, we will resolutely use continuations for everything. This can be seen as a way of making control flow explicit. There are results saying that the output of CPS conversion is invariant under interpretation as pass-by-name or pass-by-value, though we will not go into those results in this class. CPS conversion gives us named intermediate results. Thirdly, we reify control-flow as data. The first two of these three properties are commonly called “monadic form.”

### 1. IL-CPS

We first must define the target language for this transformation. Notably, we split terms into two syntactic classes; *expressions* and *values*. One may think of expressions as values that are computed and then thrown away.

We may formalize this intuition as follows:

$$\begin{aligned}
 v ::= & x \\
 & | \lambda x : \tau. e \\
 & | \text{pack } [c, v] \text{ as } \exists \alpha : k. \tau \\
 & | \langle v_1, \dots, v_n \rangle \\
 e ::= & vv \\
 & | \text{unpack } [\alpha, x] = v \text{ in } e \\
 & | \text{let } x = \pi_i v \text{ in } e \\
 & | \text{let } x = v \text{ in } e \\
 & | \text{halt}
 \end{aligned}$$

IL-CPS has the following typing rules:

$$\begin{array}{c}
 \frac{\Gamma \vdash \tau : T \quad \Gamma, x : \tau \vdash e : 0}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow 0} \qquad \frac{\Gamma \vdash v_1 : \neg \tau \quad \Gamma \vdash v_2 : \tau}{\Gamma \vdash v_1 v_2 : 0} \\
 \\
 \frac{\Gamma \vdash c : k \quad \Gamma \vdash v : [c/\alpha]\tau \quad \Gamma, \alpha : k \vdash \tau : T}{\Gamma \vdash \text{pack } [c, v] \text{ as } \exists \alpha : k. \tau : \exists \alpha : k. \tau} \\
 \\
 \frac{\Gamma \vdash v : \exists \alpha : k. \tau \quad \Gamma, \alpha : k, x : \tau \vdash e : 0}{\Gamma \vdash \text{unpack } [\alpha, x] = v \text{ in } e : 0} \qquad \frac{\Gamma \vdash v_i : \tau_i \quad (\text{for } i = 1 \dots n)}{\Gamma \vdash \langle v_1, \dots, v_n \rangle : \times[\tau_1, \dots, \tau_n]} \\
 \\
 \frac{\Gamma \vdash v : \times[\tau_1, \dots, \tau_n]}{\Gamma \vdash \text{let } x = \pi_i v \text{ in } e : 0} \qquad \frac{\Gamma \vdash v : \tau \quad \Gamma, x : \tau \vdash e : 0}{\Gamma \vdash \text{let } x = v \text{ in } e : 0} \qquad \frac{}{\Gamma \vdash \text{halt} : 0} \\
 \\
 \frac{\Gamma \vdash \tau : T}{\Gamma \vdash \neg \tau : T}
 \end{array}$$

Note that in constructive logic, the proposition “ $\tau \rightarrow 0$ ” is exactly  $\neg\tau$ . So we may perhaps cloyingly say that continuations are negation.

A careful reader may notice our usual sleight of hand in the **unpack** rule: the  $\alpha$ ’s mentioned are all asserted to be equal.

## 2. CPS CONVERSION: COMPILER PASS

Kind, constructor, and type translation are all still syntax-directed. Most every transformation is an identity mapping, with one exception:

$$\tau_1 \rightarrow \tau_2 = \neg(\tau_1 \times \neg\tau_2).$$

There’s a neat connection to constructive logic here; by the Curry-Howard Isomorphism, this is analogous to the transformation  $A \supset B$  goes to  $\neg(A \wedge \neg B)$ . We’re effectively DeMorgan-ing our code here.

Context translation is just the usual map of kind and type translation.

### 2.1. TRANSFORMING TERMS

We have

$$\Gamma \vdash e : \tau \rightarrow x.e$$

Here,  $e$  is a continuation that passes its value to the bound variable  $x$ . We maintain the invariant that “If  $\Gamma \vdash e : \tau \rightarrow x.e$ , then  $\Gamma x : \neg\tau \vdash e : 0$ .”

In respect of convention, we’ll strive to use the variable  $k$  instead of  $x$  as the continuation variable here. One hopes that this does not cause the reader any great difficulty, as we also often use the variable  $k$  for kinds.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow k.(kx)}$$

$$\frac{\Gamma \vdash e : \times[\tau_0, \dots, \tau_{n-1}] \rightsquigarrow k'.e}{\Gamma \vdash \pi_i(e) : \tau_i \rightsquigarrow k.(\text{let } k' = (\lambda x : \times[\tau_0, \dots, \tau_{n-1}].\text{let } y = \pi_i k \text{ in } ky) \text{ in } e)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \rightsquigarrow k_i.e_i \quad (\text{for } i = 1, \dots, n)}{\Gamma \vdash \langle e_1, \dots, e_n \rangle : \times[\tau_1, \dots, \tau_n] \rightsquigarrow k. \left( \begin{array}{l} \text{let } k_1 = (\lambda x_i : \tau_1. \\ \text{let } k_2 = (\lambda x_i : \tau_2. \dots \\ \text{let } k_n = (k \langle x_1, \dots, x_n \rangle) \text{ in } e_n) \text{ in } e_{n-1}) \\ \text{in } \dots \text{ in } e_2) \text{ in } e_1 \end{array} \right)}$$

$$\begin{array}{c}
\frac{\Gamma \vdash \tau_1 : T \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow k'.e}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \rightsquigarrow k.k \left( \begin{array}{l} \lambda y : \tau_1 \times \neg \tau_2. \\ \text{let } x = \pi_0 y \text{ in} \\ \text{let } k' = \pi_1 y \text{ in } e \end{array} \right)} \\
\\
\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \rightsquigarrow k_1.e_1 \quad \Gamma \vdash e_2 : \tau \rightarrow \tau' \rightsquigarrow k_2.e_2}{\Gamma \vdash e_1 e_2 : \tau' \rightsquigarrow k. \left( \begin{array}{l} \text{let } k_1 = (\lambda f : \neg(\tau \times \neg \tau')). \\ \text{let } k_2 = (\lambda x : \tau. f(x, k)) \text{ in } e_2 \\ \text{in } e_1 \end{array} \right)} \\
\\
\frac{\Gamma \vdash c : k \quad \Gamma \vdash e : [c/\alpha] \tau \rightsquigarrow k'.e \quad \Gamma, \alpha : k \vdash \tau : T}{\Gamma \vdash \text{pack } [c, e] \text{ as } \exists \alpha : k. \tau : \exists \alpha : k. \tau \rightsquigarrow k. \left( \begin{array}{l} \text{let } k' = \\ \lambda x : [c/\alpha]. e.k (\text{pack } [c, x] \text{ as } \exists \alpha : k. \tau) \\ \text{in } e \end{array} \right)} \\
\\
\frac{\Gamma \vdash e_1 : \exists \alpha : k. \tau \rightsquigarrow k_1.e_1 \quad \Gamma, \alpha : k, x : \tau \vdash e_1 : e_2 : \tau' \rightsquigarrow k_2.e_2}{\Gamma \vdash \text{unpack } [\alpha, x] = e_1 \text{ in } e_2 \rightsquigarrow k. \left( \begin{array}{l} \text{let } k_1 = \lambda x_1 : (\exists \alpha : k. \tau). \\ (\text{unpack } [\alpha, x] = x_1 \text{ in } (\text{let } k_2 = \lambda x_2 : \tau'. k x_2 \text{ in } e_2)) \\ \text{in } e_1 \end{array} \right)} \\
\\
\frac{\Gamma \vdash k : \text{kind} \quad \Gamma, \alpha : k \vdash e : \tau \rightsquigarrow k'.e}{\Gamma \vdash \Lambda \alpha : k. e : \forall \alpha : k. e \rightsquigarrow k.k \left( \begin{array}{l} (\lambda x : (\exists \alpha : k. \neg \tau). \\ \text{unpack } [\alpha, k'] = x \text{ in } e) \end{array} \right)} \\
\\
\frac{\Gamma \vdash \forall \alpha : k. \tau \rightsquigarrow k'.e \quad \Gamma \vdash c : k}{\Gamma \vdash e[c] : [c/\alpha] \tau \rightsquigarrow k. \left( \begin{array}{l} \text{let } k' = \lambda f : (\neg(\exists \alpha : k. \neg \tau)). \\ f(\text{pack } [c, k] \text{ as } \exists \alpha : k. \neg \tau) \\ \text{in } e \end{array} \right)}
\end{array}$$

We also have the following type transformations:

$$\begin{aligned}
\tau_1 \rightarrow \tau_2 &= \neg(\tau_1 \times \neg \tau_2) \\
\forall \alpha : k. \tau &= \neg(\exists \alpha : k. \neg \tau)
\end{aligned}$$

We won't talk about sums, references, exns, primitives, or recursive types here. These are left as an exercise for the reader.

## 2.2. EXCEPTIONS

We're basically just going to go through everything and redo it. However, the rewrites are pretty straightforward - we're basically just going to pass a failure continuation in through everything.

We have new type transformations:

$$\begin{aligned}
\tau_1 \rightarrow \tau_2 &= \neg(\times[\tau_1, \neg \tau_2, \neg \text{exn}]) \\
\forall \alpha : k. \tau &= \neg(\exists \alpha : k. (\neg \tau \times \neg \text{exn}))
\end{aligned}$$

We also change our judgement to have the form

$$\Gamma \vdash e : \tau \rightsquigarrow kk_{ex}.e$$

Most rules remain largely unchanged, just pushing the failure continuation through. For instance,

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow kk_{ex}.kx}$$

$$\frac{\Gamma \vdash e_i : \tau_i \rightsquigarrow k_i k_{ex_i}.e_i \quad (\text{for } i = 1, \dots, n)}{\Gamma \vdash \langle e_1, \dots, e_n \rangle : \times[\tau_1, \dots, \tau_n] \rightsquigarrow kk_{ex}. \left( \begin{array}{l} \text{let } k_1 = (\lambda x_i : \tau_1. \\ \text{let } k_2 = (\lambda x_i : \tau_2. \dots \\ \text{let } k_n = (k \langle x_1, \dots, x_n \rangle) \text{ in } e_n) \text{ in } e_{n-1}) \\ \text{in } \dots) \text{ in } e_2) \text{ in } e_1 \end{array} \right)}$$

However, the rules for exceptional control flow have yet to be defined:

$$\frac{\Gamma \vdash \tau : T \quad \Gamma \vdash e : \mathbf{exn} \rightsquigarrow k'k'_{ex}.e}{\Gamma \vdash \mathbf{raise}_\tau e : \tau \rightsquigarrow kk_{ex}. \left( \begin{array}{l} \text{let } k' = (\lambda x : \mathbf{exn}. (k_{ex}x)) \\ \text{in let } k'_{ex} = k_{ex} \\ \text{in } e \end{array} \right)}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightsquigarrow k_1 k_{ex_1}.e_1 \quad \Gamma, x : \mathbf{exn} \vdash e_2 : \tau \rightsquigarrow k_2 k_{ex_2}.e_2}{\Gamma \vdash \mathbf{handle}(e_1, x.e_2) : \tau \rightsquigarrow kk_{ex}. \left( \begin{array}{l} \text{let } k_1 = k \text{ in} \\ \text{let } k_{ex} = (\lambda x : \mathbf{exn}. \\ \quad \text{let } k_2 = k \text{ in let } k_{ex_2} = k_{ex} \text{ in } e_2) \\ \text{in } e_1 \end{array} \right)}$$

$$\frac{\Gamma \vdash \tau_1 : T \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow k'k'_{ex}.e}{\Gamma \vdash \lambda x : \tau_1. e : (\tau_1 \rightarrow \tau_2) \rightsquigarrow kk_{ex}.k \left( \begin{array}{l} \lambda(y : \times[\tau_1, \neg\tau_2, \neg\mathbf{exn}]). \\ \text{let } x = \pi_0 y \\ \text{let } k' = \pi_1 y \\ \text{let } k'_{ex} = \pi_2 y \\ \text{in } e \end{array} \right)}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \rightsquigarrow k_1 k_{ex}.e_1 \quad \Gamma \vdash e_2 : \tau \rightsquigarrow k_2 k_{ex}.e_2}{\Gamma \vdash e_1 e_2 : \tau' \rightsquigarrow kk_{ex}. \left( \begin{array}{l} \text{let } k_1 (\lambda f : \neg(\times[\tau, \neg\tau', \neg\mathbf{exn}])). \\ \text{let } k_2 = (\lambda x : \tau. f \langle x, k, k_{ex} \rangle) \\ \text{in } e_2) \text{ in } e_1 \end{array} \right)}$$