1 Object language

Let our object language be:

\[ t \in \text{Term} ::= x \quad \text{(variable)} \]
\[ \quad | \quad \lambda x.t \quad \text{(\(\lambda\)-abstraction)} \]
\[ \quad | \quad t_1 t_2 \quad \text{(application)} \]

2 Simple evaluator using substitution

The following eval function is of type \(\text{Term} \rightarrow \text{Term}\) (it is actually a partial function because it can get stuck in the application case, or it can diverge):

\[
\begin{align*}
\text{eval}(x) &= x \\
\text{eval}(\lambda x.t) &= \lambda x.t \\
\text{eval}(fa) &= \text{let } \lambda x.b = \text{eval}(f) \text{ in } \text{eval}(b[x \mapsto a])
\end{align*}
\]

3 Closure conversion

This evaluator is of type \((\text{Term} \times \text{Env}) \rightarrow (\text{Value} \times \text{Env})\), where \(\text{Env} = \text{Var} \rightarrow (\text{Term} \times \text{Env})\) and where Value is the type of values—a subtype of Term:

\[
\begin{align*}
\text{eval}(x, e) &= \text{let } (t, e') = e(x) \text{ in } \text{eval}(t, e') \\
\text{eval}(\lambda x.t, e) &= (\lambda x.t, e) \\
\text{eval}(fa, e) &= \text{let } (\lambda x.b, e') = \text{eval}(f, e) \text{ in } \\
&\quad \text{eval}(b, e'[x \mapsto (a, e)])
\end{align*}
\]

Given a term \(t\), we evaluate \(t\) by first initializing the environment to \(\text{em}\) (the empty environment \(\lambda x.\text{error}\)): \(\text{eval}(t, \text{em})\).
4 CPS transformation

This evaluator is of type \((\text{Term} \times \text{Env} \times \text{Cont}) \rightarrow \text{VClosure}\), where \(\text{Cont} = \text{VClosure} \rightarrow \text{VClosure}\) and \(\text{VClosure} = \text{Value} \times \text{Env}\):

\[
\begin{align*}
\text{eval}(x, e, k) & = \text{let } (t, e') = e(x) \text{ in eval}(t, e', k) \\
\text{eval}(\lambda x.t, e, k) & = k(\lambda x.t, e) \\
\text{eval}(fa, e, k) & = \text{eval}(f, e, k') \text{, where } k' = \lambda (\lambda x.b, e').\text{eval}(b, e'[x \mapsto (a, e)], k)
\end{align*}
\]

The continuation \(k'\) says what the evaluator is supposed to do once \(f\) has been evaluated to a value. What \(k'\) does is that it takes as input a closure of the form \((v, e')\) and checks whether \(v\) is a \(\lambda\)-expression of the form \(\lambda x.b\). If it’s not then the computation gets stuck because we don’t get a \(\beta\)-redex. Otherwise the continuation says that as before, we have to keep evaluating the body \(b\) of the \(\lambda\)-expression, where \(x\) now gets bound to the argument \((a, e)\). This amounts to doing \(\beta\)-reduction.

Our initial continuation is simply the identity function \(\text{IK} = \lambda x.x\): \(\text{eval}(t, \text{em}, \text{IK})\).

5 Defunctionalization

Continuations are not encoded by a datatype:

\[
\begin{align*}
\text{CONT} & ::= \text{CONT}_I \\
& \quad | \text{CONT}_\text{LAM} \text{ of } \text{Term} \times \text{Env} \times \text{Cont}
\end{align*}
\]

This evaluator is of type \((\text{Term} \times \text{Env} \times \text{CONT}) \rightarrow \text{VClosure}\):

\[
\begin{align*}
\text{eval}(x, e, k) & = \text{let } (t, e') = e(x) \text{ in eval}(t, e', k) \\
\text{eval}(\lambda x.t, e, k) & = \text{apply}_\text{cont}(\lambda x.t, e, k) \\
\text{eval}(fa, e, k) & = \text{eval}(f, e, \text{CONT}_\text{LAM}(a, e, k))
\end{align*}
\]

Where \(\text{apply}_\text{cont}\) (of type \((\text{Value} \times \text{Env} \times \text{CONT}) \rightarrow \text{VClosure}\)) is defined as follows:

\[
\begin{align*}
\text{apply}_\text{cont}(t, e, \text{CONT}_I) & = (t, e) \\
\text{apply}_\text{cont}(\lambda x.b, e', \text{CONT}_\text{LAM}(a, e, k)) & = \text{eval}(b, e'[x \mapsto (a, e)], k)
\end{align*}
\]

Our initial continuation is now \(\text{IK} = \text{CONT}_I\): \(\text{eval}(t, \text{em}, \text{IK})\).

6 Abstract state machine

Let us now turn our defunctionalized evaluated into an abstract state machine (a variant of Kivine’s machine [1] that uses names instead of De Bruijn indices), where the environment part is our heap and the continuation part is our stack.

\[
\begin{align*}
\text{State} & ::= \text{EVAL} \\
& \quad | \text{APPLY}_\text{CONT}
\end{align*}
\]

Here is a simple abstract machine of type \((\text{State} \times \text{Term} \times \text{Env} \times \text{Cont}) \rightarrow \text{VClosure}\):
loop(EVAL, x, e, k) = let (t, e') = e(x) in loop(EVAL, t, e', k)
loop(EVAL, λx.t, e, k) = loop(APPLY_CONT, λx.t, e, k)
loop(EVAL, fa, e, k) = loop(EVAL, f, e, CONT_LAM(a, e, k))
loop(APPLY_CONT, t, e, CONT_I) = (t, e)
loop(APPLY_CONT, λx.b, e', CONT_LAM(a, e, k)) = loop(EVAL, b, e'[x ↦→ (a, e)], k)

Let’s get rid of State and inline the APPLY_CONT cases. We also turn our continuations into a list where CONT_I is now the empty list [] and CONT_LAM is turned into the list constructor “::”:

loop(t, e, []) = (t, e)
loop(x, e, l) = let (t, e') = e(x) in loop(t, e', l)
loop(λx.t, e, l) = match l with
  | [] ⇒ (λx.t, e)
  | (a, e') :: l ⇒ loop(b, e[x ↦→ (a, e')], l)
end
loop(fa, e, l) = loop(EVAL, f, e, (a, e) :: l)

References