Continuations

Vincent Rahli (rahli@cs.cornell.edu)

CS6110 Lecture 12

Wednesday Feb 18, 2015

1 Object language

In these notes, we follow Danvy et al.'s methodology to build an abstract state machine for a functional programming language [1]. We have already seen the closure conversion phase in a previous lecture. We will now cover the CPS transformation, defunctionalization and finally the last transformation gives us an abstract state machine.

We will use the same source language `SL` as before, and in addition we add a fixpoint operator:

t	$\in$	Term	$::=$	x	(variable)
			$\mid$	$\underline{n}$	(natural number)
			$\mid$	$\underline{\lambda} x.t$	(λ-abstraction)
			$\mid$	$t_1 t_2$	(application)
			$\mid$	$t_1 \underline{+} t_2$	(addition)
			$\mid$	$\underline{\texttt{let}}\ x = t_1\ \underline{\texttt{in}}\ t_2$	(let-expression)
			$\mid$	$\underline{\texttt{fix}}(t)$	(fixpoint)

Remember that this language is lazy and that we use static scoping.

2 Simple evaluator using substitution

Let us start by defining the big-step operational semantics of this language using substitution. The following eval function is of type Term $\to$ Term (it is actually a partial function because it can get stuck in the application case, or it can diverge):

$\text{eval}(x)$	$=$	x
$\text{eval}(\underline{n})$	$=$	$\underline{n}$
$\text{eval}(\underline{\lambda} x.t)$	$=$	$\underline{\lambda} x.t$
$\text{eval}(f a)$	$=$	let $\underline{\lambda} x.b = \text{eval}(f)$ in $\text{eval}(b[x \backslash a])$
$\text{eval}(n \underline{+} m, c)$	$=$	let $\underline{v} = \text{eval}(n)$ in let $\underline{w} = \text{eval}(m)$ in $\underline{v + w}$
$\text{eval}(\underline{\texttt{let}}\ x = t\ \underline{\texttt{in}}\ b)$	$=$	let $v = \text{eval}(t)$ in $\text{eval}(b[x \backslash v])$
$\text{eval}(\underline{\texttt{fix}}(t))$	$=$	$\text{eval}(t(\underline{\texttt{fix}}(t)))$

Let us consider the same example as before:

$$\begin{array}{l}\underline{\texttt{let}}\ x = \underline{1}\ \underline{\texttt{in}} \\ \quad \underline{\texttt{let}}\ y = \underline{\lambda} z.x\ \underline{\texttt{in}} \\ \quad\quad \underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}} \\ \quad\quad y(\underline{17})\end{array}$$

and let's reduce that term:

$$\begin{array}{l}\text{eval}(\underline{\texttt{let}}\ x = \underline{1}\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ y = \underline{\lambda} z.x\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}}\ y(\underline{17})) \\ = \text{let } v = \text{eval}(\underline{1}) \text{ in eval}(\underline{\texttt{let}}\ y = \underline{\lambda} z.v\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}}\ y(\underline{17})) \\ = \text{let } v = \underline{1} \text{ in eval}(\underline{\texttt{let}}\ y = \underline{\lambda} z.v\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}}\ y(\underline{17})) \\ = \text{eval}(\underline{\texttt{let}}\ y = \underline{\lambda} z.\underline{1}\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}}\ y(\underline{17})) \\ = \text{let } v = \text{eval}(\underline{\lambda} z.\underline{1}) \text{ in eval}(\underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}}\ v(\underline{17})) \\ = \text{let } v = \underline{\lambda} z.\underline{1} \text{ in eval}(\underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ v(17)) \\ = \text{eval}(\underline{\texttt{let}}\ x = \underline{3}\ \underline{\texttt{in}}\ (\underline{\lambda} z.\underline{1})(\underline{17})) \\ = \text{let } v = \text{eval}(\underline{3}) \text{ in eval}((\underline{\lambda} z.\underline{1})(\underline{17})) \\ = \text{let } v = \underline{3} \text{ in eval}((\underline{\lambda} z.\underline{1})(\underline{17})) \\ = \text{eval}((\underline{\lambda} z.\underline{1})(\underline{17})) \\ = \text{let } \underline{\lambda} x.b = \text{eval}(\underline{\lambda} z.\underline{1}) \text{ in eval}(b[x \backslash \underline{17}]) \\ = \text{let } \underline{\lambda} x.b = \underline{\lambda} z.\underline{1} \text{ in eval}(b[x \backslash \underline{17}]) \\ = \text{eval}(\underline{1}[z \backslash \underline{17}]) \\ = \underline{1}\end{array}$$

Here is another example:

$$((\underline{\lambda} x.\underline{\lambda} y.x \underline{+} y)\underline{2})\underline{3}$$

Let's reduce that term:

$$\begin{array}{l}\text{eval}(((\underline{\lambda} x.\underline{\lambda} y.x \underline{+} y)\underline{2})\underline{3}) \\ = \text{let } \underline{\lambda} x.b = \text{eval}((\underline{\lambda} x.\underline{\lambda} y.x \underline{+} y)\underline{2}) \text{ in eval}(b[x \backslash \underline{3}]) \\ = \text{let } \underline{\lambda} x.b = (\text{let } \underline{\lambda} x.b = \text{eval}(\underline{\lambda} x.\underline{\lambda} y.x \underline{+} y) \text{ in eval}(b[x \backslash \underline{2}])) \text{ in eval}(b[x \backslash \underline{3}]) \\ = \text{let } \underline{\lambda} x.b = (\text{let } \underline{\lambda} x.b = \underline{\lambda} x.\underline{\lambda} y.x \underline{+} y \text{ in eval}(b[x \backslash \underline{2}])) \text{ in eval}(b[x \backslash \underline{3}]) \\ = \text{let } \underline{\lambda} x.b = \text{eval}(\underline{\lambda} y.\underline{2} \underline{+} y) \text{ in eval}(b[x \backslash \underline{3}]) \\ = \text{let } \underline{\lambda} x.b = \underline{\lambda} y.\underline{2} \underline{+} y \text{ in eval}(b[x \backslash \underline{3}]) \\ = \text{eval}(2 \underline{+} 3) \\ = \text{let } \underline{v} = \text{eval}(\underline{2}) \text{ in let } \underline{w} = \text{eval}(\underline{3}) \text{ in eval}(\underline{v + w}) \\ = \text{let } \underline{v} = \underline{2} \text{ in let } \underline{w} = \text{eval}(\underline{3}) \text{ in eval}(\underline{v + w}) \\ = \text{let } \underline{w} = \text{eval}(\underline{3}) \text{ in eval}(\underline{2 + w}) \\ = \text{let } \underline{w} = \underline{3} \text{ in eval}(\underline{2 + w}) \\ = \text{eval}(\underline{2 + 3}) \\ = \underline{5}\end{array}$$

3 Closure conversion

Let us now use environments and closures instead of substitution. We have seen two ways the laziness of the language can be dealt with when using environments. In the application case, we can either (1) recursively call eval in the

environment, which means that closures are pairs of a value and an environment and in the variable case the only thing we have to do is fetch the right closure from the environment, which will trigger the lazy call to eval that we've stored in our environment in the application case; or (2) we can store non-values in the environment and explicitly call eval in the variable case. Here we choose the second solution (environments are of type $\mathsf{Env} = \mathsf{Var} \to (\mathsf{Term} \times \mathsf{Env})$ and this evaluator is of type $(\mathsf{Term} \times \mathsf{Env}) \to (\mathsf{Value} \times \mathsf{Env})$, where Value is the type of values—a subtype of Term):

$$
\begin{array}{lcl}
\mathrm{eval}(x, e) & = & \text{let } (t, e') = e(x) \text{ in } \mathrm{eval}(t, e') \\
\mathrm{eval}(\underline{n}, e) & = & (\underline{n}, \mathtt{em}) \\
\mathrm{eval}(\underline{\lambda} x.t, e) & = & (\underline{\lambda} x.t, e) \\
\mathrm{eval}(fa, e) & = & \text{let } (\underline{\lambda} x.b, e') = \mathrm{eval}(f, e) \text{ in} \\
 & & \mathrm{eval}(b, e'[x \mapsto (a, e)]) \\
\mathrm{eval}(n \mathbin{\underline{+}} m, e) & = & \text{let } (v, e_1) = \mathrm{eval}(n, e) \text{ in} \\
 & & \text{let } (w, e_2) = \mathrm{eval}(m, e) \text{ in} \\
 & & (v + w, \mathtt{em}) \\
\mathrm{eval}(\underline{\mathtt{let}}\ x = t\ \underline{\mathtt{in}}\ b, e) & = & \text{let } (v, e') = \mathrm{eval}(t, e) \text{ in} \\
 & & \mathrm{eval}(b, e[x \mapsto (v, e')]) \\
\mathrm{eval}(\underline{\mathtt{fix}}(t), e) & = & \mathrm{eval}(t(\underline{\mathtt{fix}}(t)), e)
\end{array}
$$

Given a term t, we evaluate t by first initializing the environment to $\mathtt{em}$: $\mathrm{eval}(t, \mathtt{em})$. Let's now evaluate

$$
\begin{array}{l}
\underline{\mathtt{let}}\ x = \underline{1}\ \underline{\mathtt{in}} \\
\ \underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}} \\
\ \ \underline{\mathtt{let}}\ x = \underline{3}\ \underline{\mathtt{in}} \\
\ \ y(\underline{17})
\end{array}
$$

using this new evaluator. For simplicity, we will oftern write v for the closure $(v, \mathtt{em})$ and $\underline{i}$ for $(\underline{i}, e)$ in environments, and i for $\underline{i}$.

$$
\begin{aligned}
&\mathrm{eval}(\underline{\mathtt{let}}\ x = 1\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}) \\
&= \text{let } (v, e') = \mathrm{eval}(1, \mathtt{em}) \\
&\quad \text{in } \mathrm{eval}(\underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto (v, e')]) \\
&= \text{let } (v, e') = (1, \mathtt{em}) \\
&\quad \text{in } \mathrm{eval}(\underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto (v, e')]) \\
&= \mathrm{eval}(\underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1]) \\
&= \text{let } (v, e') = \mathrm{eval}(\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1]) \\
&\quad \text{in } \mathrm{eval}(\underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1][y \mapsto (v, e')]) \\
&= \text{let } (v, e') = (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1]) \\
&\quad \text{in } \mathrm{eval}(\underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1][y \mapsto (v, e')]) \\
&= \mathrm{eval}(\underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])]) \\
&= \text{let } (v, e') = \mathrm{eval}(3, \mathtt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])]) \\
&\quad \text{in } \mathrm{eval}(y(17), \mathtt{em}[x \mapsto (v, e')][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])]) \\
&= \text{let } (v, e') = (3, \mathtt{em}) \text{ in } \mathrm{eval}(y(17), \mathtt{em}[x \mapsto (v, e')][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])]) \\
&= \mathrm{eval}(y(17), \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])]) \\
&= \text{let } (\underline{\lambda} x.b, e') = \mathrm{eval}(y, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])]) \\
&\quad \text{in } \mathrm{eval}(b, e'[x \mapsto (17, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])])]) \\
&= \text{let } (\underline{\lambda} x.b, e') = (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1]) \\
&\quad \text{in } \mathrm{eval}(b, e'[x \mapsto (17, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])])]) \\
&= \mathrm{eval}(x, \mathtt{em}[x \mapsto 1][z \mapsto (17, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])])]) \\
&= 1
\end{aligned}
$$

Let's now evaluate

$$((\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y)\underline{2})\underline{3}$$

using our new evaluator:

$$
\begin{aligned}
&\mathrm{eval}(((\underline{\lambda}x.\underline{\lambda}y.x \mathbin{\underline{+}} y)\underline{2})\underline{3}, \mathtt{em}) \\
&= \text{let } (\underline{\lambda}x.b, e') = \mathrm{eval}((\underline{\lambda}x.\underline{\lambda}y.x \mathbin{\underline{+}} y)\underline{2}, \mathtt{em}) \text{ in } \mathrm{eval}(b, e'[x \mapsto \underline{3}]) \\
&= \text{let } (\underline{\lambda}x.b, e') = (\text{let } (\underline{\lambda}x.b, e') = \mathrm{eval}(\underline{\lambda}x.\underline{\lambda}y.x \mathbin{\underline{+}} y, \mathtt{em}) \text{ in } \mathrm{eval}(b, e'[x \mapsto \underline{2}])) \\
&\quad \text{in } \mathrm{eval}(b, e'[x \mapsto \underline{3}]) \\
&= \text{let } (\underline{\lambda}x.b, e') = (\text{let } (\underline{\lambda}x.b, e') = (\underline{\lambda}x.\underline{\lambda}y.x \mathbin{\underline{+}} y, \mathtt{em}) \text{ in } \mathrm{eval}(b, e'[x \mapsto \underline{2}])) \\
&\quad \text{in } \mathrm{eval}(b, e'[x \mapsto \underline{3}]) \\
&= \text{let } (\underline{\lambda}x.b, e') = \mathrm{eval}(\underline{\lambda}y.x \mathbin{\underline{+}} y, \mathtt{em}[x \mapsto \underline{2}]) \text{ in } \mathrm{eval}(b, e'[x \mapsto \underline{3}]) \\
&= \text{let } (\underline{\lambda}x.b, e') = (\underline{\lambda}y.x \mathbin{\underline{+}} y, \mathtt{em}[x \mapsto \underline{2}]) \text{ in } \mathrm{eval}(b, e'[x \mapsto \underline{3}]) \\
&= \mathrm{eval}(x \mathbin{\underline{+}} y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}]) \\
&= \text{let } (\underline{v}, e_1) = \mathrm{eval}(x, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}]) \\
&\quad \text{in let } (\underline{w}, e_2) = \mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}]) \\
&\quad \text{in } \mathrm{eval}(\underline{v + w}, \mathtt{em}) \\
&= \text{let } (\underline{v}, e_1) = \mathrm{eval}(\underline{2}, \mathtt{em}) \text{ in let } (\underline{w}, e_2) = \mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}]) \text{ in } \mathrm{eval}(\underline{v + w}, \mathtt{em}) \\
&= \text{let } (\underline{v}, e_1) = (\underline{2}, \mathtt{em}) \text{ in let } (\underline{w}, e_2) = \mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}]) \text{ in } \mathrm{eval}(\underline{v + w}, \mathtt{em}) \\
&= \text{let } (\underline{w}, e_2) = \mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}]) \text{ in } \mathrm{eval}(\underline{2 + w}) \\
&= \text{let } (\underline{w}, e_2) = \mathrm{eval}(\underline{3}, \mathtt{em}) \text{ in } \mathrm{eval}(\underline{2 + w}) \\
&= \text{let } (\underline{w}, e_2) = (\underline{3}, \mathtt{em}) \text{ in } \mathrm{eval}(\underline{2 + w}) \\
&= \mathrm{eval}(\underline{2 + 3}, \mathtt{em}) \\
&= \underline{5}
\end{aligned}
$$

4 CPS transformation

Note that, e.g., in the application case we have nested recursive calls to eval. A compiler might then have to generate a new stack frame[1] for the recursive call in addition to the one it is already using[2]. This is not satisfactory. A solution to that is to use tail-recursion, which means that recursive calls are only made as the final action of a procedure. A stack frame can then be replaced by the one for the tail-recursive call. Continuations[3] provide a way to transform a recursive function into a tail-recursive function by a simple shuffling of the different parts of the recursive function (here continuations are functions of type $\mathsf{Cont} = \mathsf{VClosure} \to \mathsf{VClosure}$, where $\mathsf{VClosure} = \mathsf{Value} \times \mathsf{Env}$, and this evaluator is of type $(\mathsf{Term} \times \mathsf{Env} \times \mathsf{Cont}) \to \mathsf{VClosure}$):

[1] `http://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_41.html`

[2] `https://wiki.haskell.org/Tail_recursion`

[3] See for example [11, 2, 3, 7, 5, 6, 9], or Dexter Kozen's notes `http://www.cs.cornell.edu/courses/CS6110/2010sp/lectures/lec14.pdf`, or Dan Licata's notes `http://www.cs.cmu.edu/~fp/courses/15317-f08/lectures/09-10-classical.pdf` for connections to logic, namely to the double negation translation that transforms classical proofs into intuitionistic proofs.

$$
\begin{array}{lcl}
\mathrm{eval}(x, e, k) & = & \text{let } (t, e') = e(x) \text{ in } \mathrm{eval}(t, e', k) \\
\mathrm{eval}(\underline{n}, e, k) & = & k(\underline{n}, \mathtt{em}) \\
\mathrm{eval}(\underline{\lambda} x.t, e, k) & = & k(\underline{\lambda} x.t, e) \\
\mathrm{eval}(f\,a, e, k) & = & \mathrm{eval}(f, e, \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto (a, e)], k)) \\
\mathrm{eval}(n \mathbin{\underline{+}} m, e, k) & = & \mathrm{eval}(n, e, \lambda(\underline{v}, e_1).\mathrm{eval}(\underline{m}, e, \lambda(w.e_2).k(\underline{v + w}, \mathtt{em}))) \\
\mathrm{eval}(\underline{\mathtt{let}}\ x = t\ \underline{\mathtt{in}}\ b, e, k) & = & \mathrm{eval}(t, e, \lambda c.\mathrm{eval}(b, e[x \mapsto c], k)) \\
\mathrm{eval}(\underline{\mathtt{fix}}(t), e, k) & = & \mathrm{eval}(t(\underline{\mathtt{fix}}(t)), e, k)
\end{array}
$$

Our initial continuation is simply the identity function $\mathtt{IK} = \lambda x.x$: $\mathrm{eval}(t, \mathtt{em}, \mathtt{IK})$. Let's now evaluate our first example:

$$
\begin{array}{l}
\mathrm{eval}(\underline{\mathtt{let}}\ x = 1\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}, \mathtt{IK}) \\
= \mathrm{eval}(1, \mathtt{em}, \lambda c.\mathrm{eval}(\underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto c], \mathtt{IK})) \\
= \mathrm{eval}(\underline{\mathtt{let}}\ y = \underline{\lambda} z.x\ \underline{\mathtt{in}}\ \underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1], \mathtt{IK}) \\
= \mathrm{eval}(\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1], \lambda c.\mathrm{eval}(\underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1][y \mapsto c]), \mathtt{IK}) \\
= \mathrm{eval}(\underline{\mathtt{let}}\ x = 3\ \underline{\mathtt{in}}\ y(17), \mathtt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])], \mathtt{IK})) \\
= \mathrm{eval}(3, \mathtt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])], \\
\qquad \lambda c.\mathrm{eval}(y(17), \mathtt{em}[x \mapsto c][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])], \mathtt{IK})) \\
= \mathrm{eval}(y(17), \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])], \mathtt{IK}) \\
= \mathrm{eval}(y, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])], \\
\qquad \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto (17, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])])], \mathtt{IK})) \\
= \mathrm{eval}(\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1], \\
\qquad \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto (17, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])])], \mathtt{IK})) \\
= \mathrm{eval}(x, \mathtt{em}[x \mapsto 1][z \mapsto (17, \mathtt{em}[x \mapsto (3, \mathtt{em})][y \mapsto (\underline{\lambda} z.x, \mathtt{em}[x \mapsto 1])])], \mathtt{IK}) \\
= \mathtt{IK}(1, \mathtt{em}) \\
= 1
\end{array}
$$

Let's now evaluate our second example:

$$
\begin{array}{l}
\mathrm{eval}(((\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y)\underline{2})\underline{3}, \mathtt{em}, \mathtt{IK}) \\
= \mathrm{eval}((\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y)\underline{2}, \mathtt{em}, \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto \underline{3}], \mathtt{IK})) \\
= \mathrm{eval}(\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y, \mathtt{em}, \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto \underline{2}], \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto \underline{3}], \mathtt{IK}))) \\
= \mathrm{eval}(\underline{\lambda} y.x \mathbin{\underline{+}} y, \mathtt{em}[x \mapsto \underline{2}], \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto \underline{3}], \mathtt{IK})) \\
= \mathrm{eval}(x \mathbin{\underline{+}} y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \mathtt{IK}) \\
= \mathrm{eval}(x, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \lambda(\underline{v}, e_1).\mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], (\underline{v + w}, \mathtt{em}))) \\
= \mathrm{eval}(\underline{2}, \mathtt{em}, \lambda(\underline{v}, e_1).\mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], (\underline{v + w}, \mathtt{em}))) \\
= \mathrm{eval}(y, \mathtt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], (\underline{2 + w}, \mathtt{em})) \\
= \mathrm{eval}(\underline{3}, \mathtt{em}, (\underline{2 + w}, \mathtt{em})) \\
= (\underline{5}, \mathtt{em}))
\end{array}
$$

5 Defunctionalization

Higher-order programming languages such as OCaml support anonymous functions which are first-class values. Defunctionalization [10, 4] provides a way to transform anonymous functions into first-order data, where each function gets assigned a unique identifier (or tag) which is the used by the code in charge

of doing function application to perform the right action. Defunctionalization transforms higher-order programs into first-order ones. Let us now defunctionalize our CPS evaluator. For that we introduce a datatype that stores the information necessary to apply our anonymous functions. This datatype has one constructor for each anonymous function (our continuations are now of type `CONT`):

$$
\begin{array}{lcll}
k & \in & \texttt{CONT} & ::= & \texttt{CONT\_I} \\
 & & & | & \texttt{CONT\_LAM} \text{ of } \mathsf{Term} \times \mathsf{Env} \times \mathsf{Cont} \\
 & & & | & \texttt{CONT\_ADD1} \text{ of } \mathsf{Term} \times \mathsf{Env} \times \mathsf{Cont} \\
 & & & | & \texttt{CONT\_ADD2} \text{ of } \mathsf{Term} \times \mathsf{Cont} \\
 & & & | & \texttt{CONT\_CBV} \text{ of } \mathsf{Term} \times \mathsf{Env} \times \mathsf{Cont}
\end{array}
$$

Here is our defunctionalized evaluator:

$$
\begin{array}{lcl}
\text{eval}(x, e, k) & = & \text{let } (t, e') = e(x) \text{ in eval}(t, e', k) \\
\text{eval}(\underline{n}, e, k) & = & \text{apply\_cont}(\underline{n}, \texttt{em}, k) \\
\text{eval}(\underline{\lambda} x.t, e, k) & = & \text{apply\_cont}(\underline{\lambda} x.t, e, k) \\
\text{eval}(f a, e, k) & = & \text{eval}(f, e, \texttt{CONT\_LAM}(a, e, k)) \\
\text{eval}(n \mathbin{\underline{+}} m, e, k) & = & \text{eval}(n, e, \texttt{CONT\_ADD1}(m, e, k)) \\
\text{eval}(\underline{\texttt{let}}\ x = t\ \underline{\texttt{in}}\ b, e, k) & = & \text{eval}(t, e, \texttt{CONT\_CBV}(\underline{\lambda} x.b, e, k)) \\
\text{eval}(\underline{\texttt{fix}}(t), e, k) & = & \text{eval}(t(\underline{\texttt{fix}}(t)), e, k)
\end{array}
$$

Where apply_cont is defined as follows:

$$
\begin{array}{lcl}
\text{apply\_cont}(t, e, \texttt{CONT\_I}) & = & (t, e) \\
\text{apply\_cont}(\underline{\lambda} x.b, e', \texttt{CONT\_LAM}(a, e, k)) & = & \text{eval}(b, e'[x \mapsto (a, e)], k) \\
\text{apply\_cont}(\underline{n}, e', \texttt{CONT\_ADD1}(m, e, k)) & = & \text{eval}(m, e, \texttt{CONT\_ADD2}(\underline{n}, k)) \\
\text{apply\_cont}(\underline{m}, e', \texttt{CONT\_ADD2}(\underline{n}, k)) & = & \text{eval}(\underline{n + m}, \texttt{em}, k) \\
\text{apply\_cont}(v, e', \texttt{CONT\_CBV}(\underline{\lambda} x.b, e, k)) & = & \text{eval}(b, e[x \mapsto (v, e')], k)
\end{array}
$$

Our initial continuation is now $\texttt{IK} = \texttt{CONT\_I}$: $\text{eval}(t, \texttt{em}, \texttt{IK})$. Let's now evaluate our first example:

$$
\begin{aligned}
&\text{eval}(\underline{\texttt{let}}\ x = 1\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ y = \underline{\lambda} z.x\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}, \texttt{IK}) \\
&= \text{eval}(1, \texttt{em}, \texttt{CONT\_CBV}(\underline{\lambda} x.\underline{\texttt{let}}\ y = \underline{\lambda} z.x\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}, \texttt{IK})) \\
&= \text{apply\_cont}(1, \texttt{em}, \texttt{CONT\_CBV}(\underline{\lambda} x.\underline{\texttt{let}}\ y = \underline{\lambda} z.x\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}, \texttt{IK})) \\
&= \text{eval}(\underline{\texttt{let}}\ y = \underline{\lambda} z.x\ \underline{\texttt{in}}\ \underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}[x \mapsto 1], \texttt{IK})) \\
&= \text{eval}(\underline{\lambda} z.x, \texttt{em}[x \mapsto 1], \texttt{CONT\_CBV}(\underline{\lambda} y.\underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}[x \mapsto 1], \texttt{IK})) \\
&= \text{apply\_cont}(\underline{\lambda} z.x, \texttt{em}[x \mapsto 1], \texttt{CONT\_CBV}(\underline{\lambda} y.\underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}[x \mapsto 1], \texttt{IK})) \\
&= \text{eval}(\underline{\texttt{let}}\ x = 3\ \underline{\texttt{in}}\ y(17), \texttt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK}) \\
&= \text{eval}(3, \texttt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \\
&\qquad \texttt{CONT\_CBV}(\underline{\lambda} x.y(17), \texttt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK})) \\
&= \text{apply\_cont}(3, \texttt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \\
&\qquad \texttt{CONT\_CBV}(\underline{\lambda} x.y(17), \texttt{em}[x \mapsto 1][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK})) \\
&= \text{eval}(y(17), \texttt{em}[x \mapsto 3][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK}) \\
&= \text{eval}(y, \texttt{em}[x \mapsto 3][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \\
&\qquad \texttt{CONT\_LAM}(17, \texttt{em}[x \mapsto 3][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK})) \\
&= \text{eval}(\underline{\lambda} z.x, \texttt{em}[x \mapsto 1], \texttt{CONT\_LAM}(17, \texttt{em}[x \mapsto 3][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK})) \\
&= \text{apply\_cont}(\underline{\lambda} z.x, \texttt{em}[x \mapsto 1], \texttt{CONT\_LAM}(17, \texttt{em}[x \mapsto 3][y \mapsto (\underline{\lambda} z.x, \texttt{em}[x \mapsto 1])], \texttt{IK})) \\
&= \text{eval}(x, \texttt{em}[x \mapsto 1][z \mapsto 17], \texttt{IK}) \\
&= \text{eval}(1, \texttt{em}, \texttt{IK}) \\
&= (1, \texttt{em})
\end{aligned}
$$

Let's now evaluate our second example:

$$
\begin{aligned}
&\text{eval}((\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y)\underline{2})\underline{3}, \texttt{em}, \texttt{IK}) \\
&= \text{eval}((\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y)\underline{2}, \texttt{em}, \texttt{CONT\_LAM}(\underline{3}, \texttt{em}, \texttt{IK})) \\
&= \text{eval}(\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y, \texttt{em}, \texttt{CONT\_LAM}(\underline{2}, \texttt{em}, \texttt{CONT\_LAM}(\underline{3}, \texttt{em}, \texttt{IK}))) \\
&= \text{apply\_cont}(\underline{\lambda} x.\underline{\lambda} y.x \mathbin{\underline{+}} y, \texttt{em}, \texttt{CONT\_LAM}(\underline{2}, \texttt{em}, \texttt{CONT\_LAM}(\underline{3}, \texttt{em}, \texttt{IK}))) \\
&= \text{eval}(\underline{\lambda} y.x \mathbin{\underline{+}} y, \texttt{em}[x \mapsto \underline{2}], \texttt{CONT\_LAM}(\underline{3}, \texttt{em}, \texttt{IK})) \\
&= \text{apply\_cont}(\underline{\lambda} y.x \mathbin{\underline{+}} y, \texttt{em}[x \mapsto \underline{2}], \texttt{CONT\_LAM}(\underline{3}, \texttt{em}, \texttt{IK})) \\
&= \text{eval}(x \mathbin{\underline{+}} y, \texttt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \texttt{IK}) \\
&= \text{eval}(x, \texttt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \texttt{CONT\_ADD1}(y, \texttt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \texttt{IK})) \\
&= \text{eval}(\underline{2}, \texttt{em}, \texttt{CONT\_ADD1}(y, \texttt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \texttt{IK})) \\
&= \text{apply\_cont}(\underline{2}, \texttt{em}, \texttt{CONT\_ADD1}(y, \texttt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \texttt{IK})) \\
&= \text{eval}(y, \texttt{em}[x \mapsto \underline{2}][y \mapsto \underline{3}], \texttt{CONT\_ADD2}(\underline{2}, \texttt{IK})) \\
&= \text{eval}(\underline{3}, \texttt{em}, \texttt{CONT\_ADD2}(\underline{2}, \texttt{IK})) \\
&= \text{apply\_cont}(\underline{3}, \texttt{em}, \texttt{CONT\_ADD2}(\underline{2}, \texttt{IK})) \\
&= \text{eval}(\underline{5}, \texttt{em}, \texttt{IK}) \\
&= \text{apply\_cont}(\underline{5}, \texttt{em}, \texttt{IK}) \\
&= (\underline{5}, \texttt{em})
\end{aligned}
$$

6 Abstract state machine

Let us now turn our defunctionalized evaluated into an abstract state machine (a variant of Kivine's machine [8]), where the environment part is our heap and the continuation part is our stack.

$$
\begin{array}{rcll}
s & \in & \mathsf{State} & ::= \quad \texttt{HALT} \\
& & & \quad\;\; | \quad \texttt{EVAL} \\
& & & \quad\;\; | \quad \texttt{APPLY\_CONT}
\end{array}
$$

Here is a simple abstract machine:

$$
\begin{array}{lcl}
\mathrm{loop}(\texttt{HALT}, t, e, k) & = & (t, e) \\
\mathrm{loop}(\texttt{EVAL}, x, e, k) & = & \text{let } (t, e') = e(x) \text{ in } \mathrm{loop}(\texttt{EVAL}, t, e', k) \\
\mathrm{loop}(\texttt{EVAL}, \underline{n}, e, k) & = & \mathrm{loop}(\texttt{APPLY\_CONT}, \underline{n}, \texttt{em}, k) \\
\mathrm{loop}(\texttt{EVAL}, \underline{\lambda} x.t, e, k) & = & \mathrm{loop}(\texttt{APPLY\_CONT}, \underline{\lambda} x.t, e, k) \\
\mathrm{loop}(\texttt{EVAL}, fa, e, k) & = & \mathrm{loop}(\texttt{EVAL}, f, e, \texttt{CONT\_LAM}(a, e, k)) \\
\mathrm{loop}(\texttt{EVAL}, n \mathbin{\underline{+}} m, e, k) & = & \mathrm{loop}(\texttt{EVAL}, n, e, \texttt{CONT\_ADD1}(m, e, k)) \\
\mathrm{loop}(\texttt{EVAL}, \underline{\texttt{let}}\ x = t\ \underline{\texttt{in}}\ b, e, k) & = & \mathrm{loop}(\texttt{EVAL}, t, e, \texttt{CONT\_CBV}(b, e, k)) \\
\mathrm{loop}(\texttt{EVAL}, \underline{\texttt{fix}}(t), e, k) & = & \mathrm{loop}(\texttt{EVAL}, t(\underline{\texttt{fix}}(t)), e, k) \\
\mathrm{loop}(\texttt{APPLY\_CONT}, t, e, \texttt{CONT\_I}) & = & (t, e) \\
\mathrm{loop}(\texttt{APPLY\_CONT}, \underline{\lambda} x.b, e', \texttt{CONT\_LAM}(a, e, k)) & = & \mathrm{loop}(\texttt{EVAL}, b, e'[x \mapsto (a, e)], k) \\
\mathrm{loop}(\texttt{APPLY\_CONT}, \underline{n}, e', \texttt{CONT\_ADD1}(m, e, k)) & = & \mathrm{loop}(\texttt{EVAL}, m, e, \texttt{CONT\_ADD2}(\underline{n}, k)) \\
\mathrm{loop}(\texttt{APPLY\_CONT}, \underline{m}, e', \texttt{CONT\_ADD2}(\underline{n}, k)) & = & \mathrm{loop}(\texttt{APPLY\_CONT}, \underline{n + m}, \texttt{em}, k) \\
\mathrm{loop}(\texttt{APPLY\_CONT}, v, e', \texttt{CONT\_CBV}(b, e, k)) & = & \mathrm{loop}(\texttt{EVAL}, b, e[x \mapsto (v, e')], k)
\end{array}
$$

In a next step, we can inline the `APPLY_CONT` cases so that we can get rid of the `EVAL` and `APPLY_CONT` tags, and we can also turn our `CONT` datatype into a list.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[4] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, September 5-7, 2001, Florence, Italy*, pages 162–174. ACM, 2001.

[5] M. Felleisen and D. Freidman. Control operators, the SECD machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–41. North-Holland, 1986.

[6] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. pages 131–141.

[7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

[8] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[9] D. MacQueen, B. Duba, and R. Harper. Typing first-class continuations in ML. In *Principles of Programming Languages*, pages 163–173, Orlando, FL, 1990.

[10] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[11] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *J. Funct. Program.*, 8(6):543–572, 1998.