Continuations

Vincent Rahli (rahli@cs.cornell.edu)

CS6110 Lecture 12

Wednesday Feb 18, 2015

1 Object language

Let our object language be:

$$\begin{array}{lllll} t & \in & \mathsf{Term} & ::= & x & \text{(variable)} \\ & & & \mid & \underline{\lambda} x.t & (\lambda\text{-abstraction}) \\ & & & \mid & t_1 t_2 & \text{(application)} \end{array}$$

2 Simple evaluator using substitution

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

$$\begin{array}{lll} \text{eval}(x) & = & x \\ \text{eval}(\underline{\lambda} x.t) & = & \underline{\lambda} x.t \\ \text{eval}(f a) & = & \text{let } \underline{\lambda} x.b = \text{eval}(f) \text{ in eval}(b[x \backslash a]) \end{array}$$

3 Closure conversion

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

$$\begin{array}{lll} \text{eval}(x, e) & = & \text{let } (t, e') = e(x) \text{ in eval}(t, e') \\ \text{eval}(\underline{\lambda} x.t, e) & = & (\underline{\lambda} x.t, e) \\ \text{eval}(f a, e) & = & \text{let } (\underline{\lambda} x.b, e') = \text{eval}(f, e) \text{ in} \\ & & \text{eval}(b, e'[x \mapsto (a, e)]) \end{array}$$

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

4 CPS transformation

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

$$
\begin{array}{lcl}
\mathrm{eval}(x, e, k) & = & \mathrm{let}\ (t, e') = e(x)\ \mathrm{in}\ \mathrm{eval}(t, e', k) \\
\mathrm{eval}(\underline{\lambda} x.t, e, k) & = & k(\underline{\lambda} x.t, e) \\
\mathrm{eval}(f\,a, e, k) & = & \mathrm{eval}(f, e, k'),\ \mathrm{where}\ k' = \lambda(\underline{\lambda} x.b, e').\mathrm{eval}(b, e'[x \mapsto (a, e)], k)
\end{array}
$$

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 $\underline{\lambda}$-expression of the form $\underline{\lambda} x.b$. If it's not then the computation gets stuck because we don't get a β-redex. Otherwise the continuation says that as before, we have to keep evaluating the body b of the $\underline{\lambda}$-expression, where x now gets bound to the argument (a, e). This amounts to doing β-reduction.

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

5 Defunctionalization

Continuations are not encoded by a datatype:

$$
\begin{array}{lcccl}
k & \in & \mathtt{CONT} & ::= & \mathtt{CONT\_I} \\
 & & & | & \mathtt{CONT\_LAM}\ \mathrm{of}\ \mathsf{Term} \times \mathsf{Env} \times \mathsf{Cont}
\end{array}
$$

This evaluator is of type $(\mathsf{Term} \times \mathsf{Env} \times \mathtt{CONT}) \to \mathsf{VClosure}$:

$$
\begin{array}{lcl}
\mathrm{eval}(x, e, k) & = & \mathrm{let}\ (t, e') = e(x)\ \mathrm{in}\ \mathrm{eval}(t, e', k) \\
\mathrm{eval}(\underline{\lambda} x.t, e, k) & = & \mathrm{apply\_cont}(\underline{\lambda} x.t, e, k) \\
\mathrm{eval}(f\,a, e, k) & = & \mathrm{eval}(f, e, \mathtt{CONT\_LAM}(a, e, k))
\end{array}
$$

Where apply_cont (of type $(\mathsf{Value} \times \mathsf{Env} \times \mathtt{CONT}) \to \mathsf{VClosure}$) is defined as follows:

$$
\begin{array}{lcl}
\mathrm{apply\_cont}(t, e, \mathtt{CONT\_I}) & = & (t, e) \\
\mathrm{apply\_cont}(\underline{\lambda} x.b, e', \mathtt{CONT\_LAM}(a, e, k)) & = & \mathrm{eval}(b, e'[x \mapsto (a, e)], k)
\end{array}
$$

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

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.

$$
\begin{array}{lcccl}
s & \in & \mathsf{State} & ::= & \mathtt{EVAL} \\
 & & & | & \mathtt{APPLY\_CONT}
\end{array}
$$

Here is a simple abstract machine of type $(\mathsf{State} \times \mathsf{Term} \times \mathsf{Env} \times \mathsf{Cont}) \to \mathsf{VClosure}$:

$$
\begin{array}{lcl}
\mathrm{loop}(\mathtt{EVAL}, x, e, k) & = & \text{let } (t, e') = e(x) \text{ in } \mathrm{loop}(\mathtt{EVAL}, t, e', k) \\
\mathrm{loop}(\mathtt{EVAL}, \underline{\lambda} x.t, e, k) & = & \mathrm{loop}(\mathtt{APPLY\_CONT}, \underline{\lambda} x.t, e, k) \\
\mathrm{loop}(\mathtt{EVAL}, f a, e, k) & = & \mathrm{loop}(\mathtt{EVAL}, f, e, \mathtt{CONT\_LAM}(a, e, k)) \\
\mathrm{loop}(\mathtt{APPLY\_CONT}, t, e, \mathtt{CONT\_I}) & = & (t, e) \\
\mathrm{loop}(\mathtt{APPLY\_CONT}, \underline{\lambda} x.b, e', \mathtt{CONT\_LAM}(a, e, k)) & = & \mathrm{loop}(\mathtt{EVAL}, b, e'[x \mapsto (a, e)], k)
\end{array}
$$

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 "::":

$$
\begin{array}{lcl}
\mathrm{loop}(t, e, []) & = & (t, e) \\
\mathrm{loop}(x, e, l) & = & \text{let } (t, e') = e(x) \text{ in } \mathrm{loop}(t, e', l) \\
\mathrm{loop}(\underline{\lambda} x.t, e, l) & = & \quad \text{match } l \text{ with} \\
 & & \quad \mid [] \Rightarrow (\underline{\lambda} x.t, e) \\
 & & \quad \mid (a, e') :: l \Rightarrow \mathrm{loop}(b, e[x \mapsto (a, e')], l) \\
 & & \quad \text{end} \\
\mathrm{loop}(f a, e, l) & = & \mathrm{loop}(\mathtt{EVAL}, f, e, (a, e) :: l)
\end{array}
$$

References

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