Implementing delimited continuations in Guarded Interaction Trees

February 29, 2024

Notations

We write

$$\begin{aligned} \mathcal{S}k.\, e &:= \text{Shift}\,(\lambda k.\, e) \\ \mathcal{R}\ \ e &:= \text{Reset}\, e \end{aligned}$$

1 Language

Similar to $\lambda_{\mathsf{rec,call/cc}}$. We have `shift`, `reset` and separate constructors for function application and continuation application (resp. App & AppCont). The reason for that is explained in section 3. Their respective behaviour are detailed in the operational semantics (see 2). The essential parts of the language are detailed in 1

$$\begin{aligned} \texttt{cont} \ni \mathsf{k} ::= {} & \square \mid \text{IfK}\, e\, e'\, \mathsf{k} \mid \text{AppLK}\, v\, \mathsf{k} \mid \text{AppRK}\, e\, \mathsf{k} \\ & \mid \text{AppContLK}\, v\, \mathsf{k} \mid \text{AppContRK}\, e\, \mathsf{k} \\ & \mid \text{NatOpLK}\ \otimes\ v\, \mathsf{k} \mid \text{NatOpRK}\ \otimes\ e\, \mathsf{k} \end{aligned}$$

$$\texttt{val} \ni v ::= n \in \mathbb{N} \mid \text{RecV}_{f,x}\, e \mid \text{ContV}\, \mathsf{k}$$

$$\texttt{expr} \ni e ::= \ \dots \mid \text{App}\, e\, e \mid \text{AppCont}\, e\, e$$

Figure 1: Source Language, specifically continuations ()

2 Operational semantics

Following the "Abstract machine" style from [BBD05], we have configurations containing a *continuation* (of type `cont` given in 1); and a *meta-continuation* which is a stack of `cont`. Configurations are given by:

$$\begin{aligned}
\langle e,\ \mathsf{k},\ \mathsf{mk}\rangle_{\mathsf{eval}} &: \mathtt{expr} \to \mathtt{cont} \to \mathtt{mcont} \to \mathtt{config} \\
\langle \mathsf{k},\ v,\ \mathsf{mk}\rangle_{\mathsf{cont}} &: \mathtt{cont} \to \mathtt{val} \to \mathtt{mcont} \to \mathtt{config} \\
\langle \mathsf{mk},\ v\rangle_{\mathsf{mcont}} &: \mathtt{mcont} \to \mathtt{val} \to \mathtt{config} \\
\langle e\rangle_{\mathsf{term}} &: \mathtt{expr} \to \mathtt{config} \\
\langle v\rangle_{\mathsf{ret}} &: \mathtt{val} \to \mathtt{config}
\end{aligned}$$

The term and ret configurations represent an initial term and a final value, respectively.

The transitions between configurations are given in 2. Except for the rules for `shift` and `reset`, transitions from an eval configuration "do not compute", but rather narrow the focus on the term being reduced to a subterm, pushing the rest into the continuation, until a value is reached and we transition to a cont configuration; while `reset` pushes the current continuation onto the meta-continuation to then focus on its argument in the empty context, and `shift` "moves" the current continuation to its argument, focusing on it in the empty context.

Transitions from cont configurations move back the outermost element of the continuation onto the term under focus, until we reach an empty context and move to an mcont config to pop from the meta-continuation. Notice that the rule for applying a continuation pushes the current context onto the meta-continuation: this is to implement the fact that `shift` wraps the current continuation inside a `reset` when calling its argument.

Finally, when a term under context has finised reducing to a value, the first element of the meta-continuation is popped out and we resume reductions.

3 Interpretation

State We use a state to record the meta-continuation, as a list of $(\blacktriangleright \mathsf{IT} \to \blacktriangleright \mathsf{IT})$. We will describe how a meta-continuation is interpreted into the state later. This serves the same purpose as the stored continuation in the Filinski translation.

Effects We use four effects: `shift`, `reset`, `pop` and `app_cont`. We use `pop` to "pop" continuations out of the state, when the main term has finished reducing (corresponds to the operational semantics rules for mcont configurations), and as such, we define interpretation such that everything is always wrapped inside of a call (by value) to `pop`. The effect `app_cont` corresponds to the language construction of the same name, and is necessary because applying a continuation pushes the current context onto the meta-continuation (see 2), so we need to interact with the state.

$$\langle e \rangle_{\mathsf{term}} \Longrightarrow_{(0,0)} \langle e,\ \Box,\ [] \rangle_{\mathsf{eval}}$$

$$\begin{aligned}
\langle \mathrm{Val}\, v,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(0,0)} \langle \mathsf{k},\ v,\ \mathsf{mk} \rangle_{\mathsf{cont}} \\
\langle e_0\, e_1,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(0,0)} \langle e_1,\ \mathrm{AppRK}\, e_0\, \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle \mathrm{AppCont}\, e_0\, e_1,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(0,0)} \langle e_1,\ \mathrm{AppContRK}\, e_0\, \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle e_0 \otimes e_1,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(0,0)} \langle e_1,\ \mathrm{NatOpRK}\ \otimes\ e_0\, \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle \mathtt{if}\ e_b\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(0,0)} \langle e_b,\ \mathrm{IfK}\, e_t\, e_f\, \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle \mathcal{R}\, e,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(1,1)} \langle e,\ \Box,\ \mathsf{k}\ ::\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle \mathcal{S}k.\, e,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} &\Longrightarrow_{(1,1)} \langle e[\mathsf{k}/k],\ \Box,\ \mathsf{mk} \rangle_{\mathsf{eval}}
\end{aligned}$$

$$\begin{aligned}
\langle \Box,\ v,\ \mathsf{mk} \rangle_{\mathsf{cont}} &\Longrightarrow_{(0,0)} \langle \mathsf{mk},\ v \rangle_{\mathsf{mcont}} \\
\langle \mathrm{AppRK}\, e\, \mathsf{k},\ v,\ \mathsf{mk} \rangle_{\mathsf{cont}} &\Longrightarrow_{(0,0)} \langle e,\ \mathrm{AppLK}\, v\, \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle \mathrm{AppContRK}\, e\, \mathsf{k},\ v,\ \mathsf{mk} \rangle_{\mathsf{cont}} &\Longrightarrow_{(0,0)} \langle e,\ \mathrm{AppContLK}\, v\, \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
\langle \mathrm{AppLK}\, v\, \mathsf{k},\ \mathrm{RecV}_{f,x}\, e,\ \mathsf{mk} \rangle_{\mathsf{cont}} &\Longrightarrow_{(1,0)} \langle e[v/x][\mathrm{RecV}_{f,x}\, e/f],\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \\
&\text{If, NatOp} \ldots \\
\langle \mathrm{AppContLK}\, v\, \mathsf{k},\ \mathrm{ContV}\, \mathsf{k}',\ \mathsf{mk} \rangle_{\mathsf{cont}} &\Longrightarrow_{(2,1)} \langle \mathsf{k}',\ v,\ \mathsf{k} :: \mathsf{mk} \rangle_{\mathsf{cont}}
\end{aligned}$$

$$\begin{aligned}
\langle \mathsf{k}\ ::\ \mathsf{mk},\ v \rangle_{\mathsf{mcont}} &\Longrightarrow_{(1,1)} \langle \mathsf{k},\ v,\ \mathsf{mk} \rangle_{\mathsf{cont}} \\
\langle [],\ v \rangle_{\mathsf{mcont}} &\Longrightarrow_{(1,1)} \langle v \rangle_{\mathsf{ret}}
\end{aligned}$$

Figure 2: Small step, preemptively indexed by corresponding number of ticks and state accesses ( )

Signatures

$$
\begin{aligned}
\mathrm{Ins}_{\texttt{shift}} &= (\blacktriangleright \cdot \to \blacktriangleright \cdot) \to \blacktriangleright \cdot & \mathrm{Ins}_{\texttt{reset}} &= \blacktriangleright \cdot \\
\mathrm{Outs}_{\texttt{shift}} &= \blacktriangleright \cdot & \mathrm{Outs}_{\texttt{reset}} &= \blacktriangleright \cdot
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Ins}_{\texttt{pop}} &= \blacktriangleright \cdot & \mathrm{Ins}_{\texttt{app\_cont}} &= (\blacktriangleright \cdot) \times (\blacktriangleright (\cdot \to \cdot)) \\
\mathrm{Outs}_{\texttt{pop}} &= \bot & \mathrm{Outs}_{\texttt{app\_cont}} &= \blacktriangleright \cdot
\end{aligned}
$$

Reifiers

To reflect the operational semantics: `shift` calls its argument on the current continuation; `reset` pushes the continuation onto the state and reduces to its argument; `pop` pops the head of the meta-continuation and reduces, and if it is empty, applies the identity continuation; and finally `app_cont` reifies like `throw`, except that it pushes the current continuation onto the meta-continuation rather than dropping it.

$$
\begin{aligned}
r_{\texttt{shift}}(f, \sigma, k) &= ((f\ k), \sigma) & r_{\texttt{pop}}(e, k :: \sigma, \_) &= ((k\ e), \sigma) \\
r_{\texttt{reset}}(e, \sigma, k) &= (e, k :: \sigma) & r_{\texttt{pop}}(e, [], \_) &= (e, []) \\
r_{\texttt{app\_cont}}((e, k), \sigma, k') &= ((k\ e), k' :: \sigma)
\end{aligned}
$$

Interpretation of effects

We write $\mathcal{P}e := \texttt{get\_val}\ (\lambda v.\mathsf{Vis}_{\texttt{pop}}\ (\mathrm{Next}\ v, !))\ e$ to interpret the effects as:

$$
\begin{aligned}
[\![\mathcal{S}k.\ e]\!]_\rho &= \mathsf{Vis}_{\texttt{shift}} \left(\mathcal{P} \circ (\lambda(k :\blacktriangleright \mathsf{IT} \to\blacktriangleright \mathsf{IT}).[\![e]\!]_{\rho \cdot \tilde{k}}), \mathrm{Id}\right) \\
[\![\mathcal{R}\ e]\!]_\rho &= \mathsf{Vis}_{\texttt{reset}} \left(\mathcal{P}[\![e]\!]_\rho, \mathrm{Id}\right) \\
[\![\mathrm{AppCont}\ k\ e]\!]_\rho &= \mathsf{Vis}_{\texttt{app\_cont}} \left(([\![e]\!]_\rho,\ [\![k]\!]_\rho), \mathrm{Id}\right)
\end{aligned}
$$

(where $\tilde{k} = \mathsf{Fun} \cdot \mathsf{Next}(\lambda x.\mathsf{Tau} \cdot k \cdot \mathsf{Next}\ x)$, *i.e.* k "transformed into" an iTree function value)

We wrap the argument for `shift` and `reset` inside a $\mathcal{P}$ as the one that would be at the top level of the term will be moved by reification.

Interpretation of meta-continuation

$$
[\![\mathsf{mk}]\!]_\rho = \mathrm{map}\ (\lambda k.\ \mathcal{P} \circ\ [\![k]\!]_\rho)\ \mathsf{mk}
$$

Again, we wrap every continuation of the stack into a $\mathcal{P}$ so that when a continuation is popped out of the state, when it finishes reducing, the next one will be popped out, until the state is empty.

Interpretation of configurations

We wrap everything into a call to `pop` to pop the meta-continuation when the term under focus has finished reducing. As every continuation in the stack is wrapped under a `pop`, each time a term finishes reducing, the next element of the meta-continuation will be popped.

$$
\begin{aligned}
\llbracket \langle e,\ \mathsf{k},\ \mathsf{mk} \rangle_{\mathsf{eval}} \rrbracket_\rho &:= \Big(\mathcal{P}\ \llbracket \mathsf{k}[e] \rrbracket_\rho,\ \llbracket \mathsf{mk} \rrbracket_\rho\Big) \\
\llbracket \langle \mathsf{k},\ \mathsf{v},\ \mathsf{mk} \rangle_{\mathsf{cont}} \rrbracket_\rho &:= \Big(\mathcal{P}\ \llbracket \mathsf{k}[e] \rrbracket_\rho,\ \llbracket \mathsf{mk} \rrbracket_\rho\Big) \\
\llbracket \langle \mathsf{mk},\ v \rangle_{\mathsf{mcont}} \rrbracket_\rho &:= \Big(\mathcal{P}\ \llbracket v \rrbracket_\rho, \llbracket \mathsf{mk} \rrbracket_\rho\Big) \\
\llbracket \langle e \rangle_{\mathsf{term}} \rrbracket_\rho &:= \Big(\mathcal{P} \llbracket e \rrbracket_\rho,\ []\Big) \\
\llbracket \langle v \rangle_{\mathsf{ret}} \rrbracket_\rho &:= \Big(\llbracket v \rrbracket_\rho,\ []\Big)
\end{aligned}
$$

Soundness We have the following soundness result:

For all configurations C, C', iTrees t, t', states σ, σ' and environment ρ such that $\llbracket C \rrbracket_\rho = (t, \sigma)$ and $\llbracket C' \rrbracket_\rho = (t', \sigma')$,

$$\text{if} \quad C \Longrightarrow^*_{(n,m)} C' \quad \text{then} \quad (t, \sigma) \rightsquigarrow^*_n (t', \sigma')$$

(where $\Longrightarrow^*$ is the reflexive transitive closure of the operational semantics transition indexed by the total number of ticks and state accesses, and $\rightsquigarrow^*$ the reflexive transitive closure of the tree reduction, indexed by the total number of ticks)

References

[BBD05] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the cps hierarchy. *Logical Methods in Computer Science*, Volume 1, Issue 2, November 2005.