PFPL Supplement: Programming with Continuations*

Robert Harper

Fall, 2022

1 Introduction

The continuations type has as values *reified* control stacks representing the state of control at a particular point in a program execution. On a stack machine the state of control is a stack, k, which is reified as a value of the form $\mathtt{cont}(k)$. Because stacks are classified by the type τ of values they expect, correspondingly, continuations are classified by types $\tau\,\mathtt{cont}$. From this and the dynamics of `letcc` and `throw` it is easy to derive the statics given in **PFPL**.

Nevertheless, it can be quite tricky to understand how to write programs with continuations. It takes some experience to get a feel for it; powerful as they are, coninuations can be quite subtle to work with. To gain intuition it is useful to rely on the stack machine dynamics given in **PFPL** and in the supplementary note Harper (2018).

2 Examples

Contraposition

As a first example, consider the function `cp`, for "contrapositive," of type

$$(\,\tau_1 \rightharpoonup \tau_2\,) \rightharpoonup \tau_2\,\mathtt{cont} \rightharpoonup \tau_1\,\mathtt{cont}$$

that pre-composes a continuation with a function,

$$\lambda\,(\,f : \tau_1 \rightharpoonup \tau_2\,)\,\lambda\,(\,r_2 : \tau_2\,\mathtt{cont}\,)\,\mathtt{letcc}(\mathit{ret}\,.\,\mathtt{throw}(\mathtt{ap}(\,f\,;\mathtt{letcc}(r_1\,.\,\mathtt{throw}(r_1\,;\mathit{ret}))\,)\,;r_2)).$$

Consider the execution of the application $e \triangleq \mathtt{cp}(\,f\,)(\,\mathtt{cont}(k_2)\,)$ on a stack k, taking some liberties to skip steps and perform substitutions along the way:

$$\begin{aligned}
k \triangleright e &\longmapsto^* k \triangleright \mathtt{letcc}(\mathit{ret}\,.\,\mathtt{throw}(\mathtt{ap}(\,f\,;\mathtt{letcc}(r_1\,.\,\mathtt{throw}(r_1\,;\mathit{ret}))\,)\,;\mathtt{cont}(k_2))) \\
&\longmapsto^* k \triangleright \mathtt{throw}(\mathtt{ap}(\,f\,;\mathtt{letcc}(r_1\,.\,\mathtt{throw}(r_1\,;\mathtt{cont}(k)))\,)\,;\mathtt{cont}(k_2)) \\
&\longmapsto^* \underbrace{k\,;\mathtt{throw}(-\,;\mathtt{cont}(k_2))\,;\mathtt{ap}(\,f\,;-\,)}_{k'} \triangleright \mathtt{letcc}(r_1\,.\,\mathtt{throw}(r_1\,;\mathtt{cont}(k))) \\
&\longmapsto^* k \triangleright \mathtt{throw}(\mathtt{cont}(k')\,;\mathtt{cont}(k)) \\
&\longmapsto^* k \triangleleft \mathtt{cont}(k')
\end{aligned}$$

Now observe that

$$\begin{aligned}
k \triangleright \mathtt{throw}(v\,;\mathtt{cont}(k')) &\longmapsto^* \underbrace{k\,;\mathtt{throw}(-\,;\mathtt{cont}(k_2))\,;\mathtt{ap}(\,f\,;-\,)}_{k'} \triangleleft v \\
&\longmapsto^* k\,;\mathtt{throw}(-\,;\mathtt{cont}(k_2)) \triangleright \mathtt{ap}(\,f\,;v\,) \\
&\longmapsto^* k\,;\mathtt{throw}(-\,;\mathtt{cont}(k_2)) \triangleleft v' \\
&\longmapsto^* k_2 \triangleleft v'
\end{aligned}$$

That is, if v is thrown to k', then, assuming that f applied to v terminates with value v', the value v' is thrown to k_2, as desired.

Law of the Excluded Middle

For a fascinating example consider the program $\mathtt{lem}_\tau$ of type $\tau + \tau\,\mathtt{cont}$,

$$\mathtt{letcc}(\mathit{ret}\,.\,\mathtt{l}\cdot\mathtt{letcc}(x\,.\,\mathtt{throw}(\mathtt{r}\cdot x\,;\mathit{ret}))).$$

Bizarrely, for any type τ at all, this program computes *either* a value of type τ, or a value of type $\tau\,\mathtt{cont}$. But how could that be? Its behavior is independent of the choice of τ, which might or might not have any values in it. As will be seen shortly, $\mathtt{lem}_\tau$ is a *pusillanimous program* that "changes its mind" to achieve this improbable description!

Consider the execution

$$\begin{aligned}
k \triangleright \mathtt{lem}_\tau &\longmapsto^* k \triangleright \mathtt{l}\cdot\mathtt{letcc}(x\,.\,\mathtt{throw}(\mathtt{r}\cdot x\,;\mathtt{cont}(k))) \\
&\longmapsto^* \underbrace{k\,;\mathtt{l}\cdot -}_{k'} \triangleright \mathtt{letcc}(x\,.\,\mathtt{throw}(\mathtt{r}\cdot x\,;\mathtt{cont}(k))) \\
&\longmapsto^* k' \triangleright \mathtt{throw}(\mathtt{r}\cdot\mathtt{cont}(k')\,;\mathtt{cont}(k)) \\
&\longmapsto^* k \triangleleft \mathtt{r}\cdot\mathtt{cont}(k')
\end{aligned}$$

Thus, if $\mathtt{lem}_\tau$ is executed on a stack k, then it returns $\mathtt{r}\cdot\mathtt{cont}(k')$ to k, where k' is as defined above. One may say that it "bluffs" by simply returning the τ-accepting stack, k', injected into the right summand. It may be that execution continues from the last state above to completion, without ever examining this returned value. In that case the bluff has succeeded, and there is nothing more to be said. However, it is of course possible that the return value is inspected non-trivially, which means to perform a case analysis, and take the right-hand branch:

$$\begin{aligned}
k \triangleleft \mathtt{r}\cdot\mathtt{cont}(k') &\longmapsto^* k''\,;\mathtt{case}(\,-\,;x\,.\,e_1\,;y\,.\,e_2\,) \triangleleft \mathtt{r}\cdot\mathtt{cont}(k') \\
&\longmapsto^* k'' \triangleright \{\mathtt{cont}(k')/y\}e_2
\end{aligned}$$

The bluff has been called, and the continuation provided at the outset is passed to the right-hand branch of the `case`. This, too, may be considered a bluff in that evaluation might proceed from here to completion without ever making use of $\mathtt{cont}(k')$. If so, the bluff was successful, and no one is the wiser. But it is also possible that this value is meaningfully used by throwing a value v to it:

$$
\begin{aligned}
k'' \triangleright \{\texttt{cont}(k')/y\}e_2 &\longmapsto^* k''' \triangleright \texttt{throw}(v\,;\texttt{cont}(k')) \\
&\longmapsto^* \underbrace{k\,;\texttt{l}\cdot -}_{k'} \triangleleft v \\
&\longmapsto^* k \triangleleft \texttt{l}\cdot v
\end{aligned}
$$

Execution may proceed as it did the first time, reviving the case analysis on the returned value, but this time taking the left-hand branch,

$$
\begin{aligned}
k \triangleleft \texttt{l}\cdot v &\longmapsto^* k''\,;\texttt{case}(-\,;x\,.\,e_1\,;y\,.\,e_2) \triangleleft \texttt{l}\cdot v \\
&\longmapsto^* k''' \triangleright \{v/x\}e_1
\end{aligned}
$$

Thus, the program has "changed its mind," re-executing the case analysis, but this time with the *given* value v, which is propaged into the left-hand branch. From there execution continues as if nothing untoward has ever happened, because there is no record of there having been some "backtracking" involved during the execution.

Thus, although the type of $\texttt{lem}_\tau$ is a sum, it is never possible to say definitively in *which* summand lies its value. Initially it provides a continuation in the right summand that is carefully prepared to avoid potential embarrassment. If the caller folds, $\texttt{lem}_\tau$ evades detection, and wins by default. If the caller performs a case analysis, then it can only call $\texttt{lem}_\tau$'s bluff by providing a value of type τ to the continuation that it has prepared. But that continuation simply reminds the caller that it need never have invoked $\texttt{lem}_\tau$ in the first place, by re-doing the case analysis with the caller's own value injected into the left summand!

Putting it into logical terms, the law of the excluded middle expresses the idea "that which is not known to be true may be regarded as false." In a transcendent world exceeding human capacities, everything that is true is known to be true, and so everything that is not known to be true can only be false. But in the actual world only finitely many facts can have been verified to be true, so there are, and always will be, open questions, statements that are neither known to be true, nor known to be false. The proof of the law of the excluded middle uses "time travel" to return to a previous point in a proof, taking advantage of the fact that the only way to refute the falsehood of a proposition is to provide a proof of its truth.

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Stacks by-name and by-value. Supplement to Harper (2016), Fall 2018. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/stacks.pdf`.