PFPL Supplement: Programming with Continuations

Robert Harper
November 9, 2019

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 \( \text{cont}(k) \). Because stacks are classified by the type \( \tau \) of values they expect, correspondingly, continuations are classified by types \( \text{cont}(\tau) \). From this and the dynamics of \text{letcc} and \text{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, continuations can also 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 \( \text{cp} \), for “contrapositive,” of type

\[
(\tau_1 \rightarrow \tau_2) \rightarrow \text{cont}(\tau_2) \rightarrow \text{cont}(\tau_1)
\]

that pre-composes a continuation with a function,

\[
\lambda(f : \tau_1 \rightarrow \tau_2) \lambda(k_2 : \text{cont}(\tau_2)) \text{letcc} \text{ret} \text{in} \text{throw} \text{ap}(f; \text{letcc} k_1 \text{in} \text{throw} k_1 \text{to} \text{ret}) \text{to} k_2.
\]

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

\[
k > e \rightarrow^* k > \text{letcc} \text{ret} \text{in} \text{throw} \text{ap}(f; \text{letcc} k_1 \text{in} \text{throw} k_1 \text{to} \text{ret}) \text{to} k \\
\rightarrow^* k > \text{throw} \text{ap}(f; \text{letcc} k_1 \text{in} \text{throw} k_1 \text{to} \text{cont}(k)) \text{to} \text{cont}(k_2) \\
\rightarrow^* k > \text{throw} \rightarrow \text{cont}(k_2); \text{ap}(f; -) \triangleright \text{letcc} k_1 \text{in} \text{throw} k_1 \text{to} \text{cont}(k) \\
\rightarrow^* k > \text{throw} \text{cont}(k') \text{to} \text{cont}(k) \\
\rightarrow^* k < \text{cont}(k')
\]
Now observe that

\[
\begin{align*}
k \triangleright & \; \text{throw } v \text{ to } \text{cont}(k') \triangleright \ast k ; \text{throw} - \text{to } \text{cont}(k_2) ; \text{ap}(f; -) \triangleleft v \\
\triangleright & \ast k ; \text{throw} - \text{to } \text{cont}(k_2) \triangleright \text{ap}(f; v) \\
\triangleright & \ast k ; \text{throw} - \text{to } \text{cont}(k_2) \triangleleft v' \\
\triangleright & \ast k_2 \triangleleft v'
\end{align*}
\]

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

**Law of the Excluded Middle**

For a fascinating example consider the program \( \text{lem}_\tau \) of type \( \tau + \text{cont}(\tau) \),

\[
\text{letcc } \text{ret in } l \cdot \text{letcc } k' \text{ in } \text{throw } \cdot k' \text{ to } \text{ret}.
\]

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

Consider the execution

\[
\begin{align*}
k \triangleright & \; \text{lem}_\tau \triangleright \ast k \triangleright l \cdot \text{letcc } k' \text{ in } \text{throw } \cdot k' \text{ to } \text{cont}(k) \\
\triangleright & \ast k ; l \cdot - \triangleright \text{letcc } k' \text{ in } \text{throw } \cdot k' \text{ to } \text{cont}(k) \\
\triangleright & \ast k' \triangleright \text{throw } \cdot \text{cont}(k') \text{ to } \text{cont}(k) \\
\triangleright & \ast k \triangleleft \text{r } \cdot \text{cont}(k')
\end{align*}
\]

Thus, if \( \text{lem}_\tau \) is executed on a stack \( k \), then it returns \( \text{r } \cdot \text{cont}(k') \) to \( k \), where \( k' \) is as defined above. One may say that it “bluffs” by simply returning the \( \tau \)-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:

\[
\begin{align*}
k \triangleleft \text{r } \cdot \text{cont}(k') \triangleright \ast k'' ; \text{case}\{x.e_1; y.e_2\}(-) \triangleleft \text{r } \cdot \text{cont}(k') \\
\triangleright \ast k'' \triangleright [\text{cont}(k') / y]e_2
\end{align*}
\]

The bluff has been called, and the continuation provided at the outset is passed to the right-hand branch of the \text{case}. This, too, may be considered a bluff in that evaluation might proceed from here to completion without ever making use of \( \text{cont}(k') \). 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:

\[
\begin{align*}
k'' \triangleright [\text{cont}(k') / y]e_2 \triangleright \ast k''' \triangleright \text{throw } v \text{ to } \text{cont}(k') \\
\triangleright \ast k ; l \cdot - \triangleleft v \\
\triangleright \ast k \triangleleft l \cdot v
\end{align*}
\]


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

\[ k \triangleleft 1 \cdot v \mapsto^* k'' ; \text{case}\{x.e_1; y.e_2\}(\_ \cdot \_ \cdot ) \triangleleft 1 \cdot v \]
\[ \mapsto^* k''' \triangleright [v/x]e_1 \]

Thus, the program has changed its mind, re-executing the case analysis, but this time with the *given*
value \( v \), which is propagated 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 \( \text{lem}_\tau \) 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, \( \text{lem}_\tau \) evade detection of its bluff,
and wins by default. If the caller performs a case analysis it can only call the bluff by providing a
value of type \( \tau \) to the prepared continuation. But then that continuation merely reminds the caller
that it need never have invoked \( \text{lem}_\tau \) 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 that which
is not known to be true may be regarded as false. In a world that transcends human capacities,
everything that is true is known to be true, and so everything that is not known to be true is, in
fact, false. But in a world in which only finitely many facts can have been verified to be true (that
is, the world we live in), there are and always will be conjectures that are neither known to be true,
nor known to be false (refuted). In that setting the excluded middle works by arranging that the
bald assertion that a conjecture is refuted cannot be refuted, because the only possible refutation
of its falsehood would provide a proof of its truth.

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press,