Recitation 10:  
Continuations and Parallelism
15-312: Foundations of Programming Languages
Yue Yao, Jeanne Luning Prak
October 30, 2019

1 Continuations

Last week, we saw how we can use control stacks to describe exception raising and handling. This week, we discuss explicit use of control stacks: continuations. Continuations allow us to save the current control stack as a value, and to reinstate this control stack at any point later in the program. This allows us access to unlimited, safe “time travel.” We can go back to a previous evaluation step of a program whenever we choose.

To illustrate this, consider a program where we might want to abort computation early: multiplying together all the numbers in a list. If our algorithm sees a 0 at any point, we know automatically that the overall product is 0, and so it’s not necessary to traverse the rest of the list. Using continuations, we can save the state of the stack before we start computing the product of the list, and if we see a zero, we can reinstate the old stack and return 0 to it.

Say we start with a stack $k$ that we will return the result of our multiplication to, and we save $k$.

$$k \triangleright \text{mult}_\text{list}(L)$$

We then begin multiplying together elements, adding more stack frames to the stack, and at some point we see a zero

$$k; \text{mult}(s(z); -); \text{mult}(s(s(z)); -); \text{mult}(s(z); -) \triangleright z$$

We can then replace the entire stack with $k$, and return $z$ to $k$:

$$k \triangleleft z$$

In the next section, we will see how to implement this “save and replace” operation by extending PCF with continuations.
2 KPCF

We extend PCF with continuations to create KPCF\(^1\).

2.1 Grammar

\[
\begin{align*}
\text{Type } \tau &::= \tau \text{ cont} \\
\text{Expr } e &::= \text{letcc}\{\tau\}(x.e) \\
& \quad \text{throw}\{\tau\}(e_1;e_2) \\
& \quad \text{cont}(k)
\end{align*}
\]

We add two new constructs to the language, \texttt{letcc} and \texttt{throw}. \texttt{letcc}\{\tau\}(x.e) saves the current continuation \texttt{cont}(k) in \texttt{x} for use in \texttt{e}, and \texttt{throw}\{\tau\}(e_1;e_2) replaces the current control stack with \texttt{e}_2 and returns \texttt{e}_1 to that stack. Note that \texttt{cont}(k) only exists in the dynamics of the language: it is only possible to create a continuation through \texttt{letcc}.

2.2 Statics

\[
\begin{align*}
\Gamma, x : \tau \text{ cont} & \vdash e : \tau \\
\Gamma \vdash \text{letcc}\{\tau\}(x.e) & \sim \tau \\
\Gamma \vdash \text{throw}\{\tau\}(e_1;e_2) & \sim \tau \\
\Gamma \vdash \text{cont}(k) : \tau \text{ cont}
\end{align*}
\]

Just like with nullary case analysis, the type of a \texttt{throw} can be arbitrary, since it does not evaluate to a value. As we all know, nullary case analysis is the elimination form for values of type \texttt{void}, which corresponds to a constructive \texttt{falsehood}. The similarity in typing is our first hint to the logical aspect of continuations.

2.3 Dynamics

The two most interesting rules (in a modal separated setting) concern how stacks are bound in \texttt{letcc} and replaced in \texttt{throw}:

\[
\begin{align*}
k \triangleright \text{letcc}\{\tau\}(x.e) & \mapsto k \triangleright [\text{cont}(k)/x]e \\
k \triangleright \text{throw}\{\tau\}(v;\text{cont}(k')) & \mapsto k' \triangleleft v
\end{align*}
\]

In a modal separated setting these are really the only two rules we need.

2.4 Example

To see how we could use these constructs we’ve defined, let’s return to our example of multiplying a list. We can do this in KPCF.

\textit{Note:} For simplicity, we’ll treat \texttt{natlist} as a primitive, though it could be encoded in KPCF as a function, or by introducing inductive types.

\(^1\text{Since, as everyone knows, continuation starts with k.}\)
2.5 Logical Aspect of Continuations

One of the central theme is the Curry-Howard correspondence: types are propositions, and values of such type are proof for the propositions. We briefly list those correspondences as follows: suppose $\tau_1$ and $\tau_2$ corresponds to propositions $A$ and $B$ respectively,

\[
\begin{align*}
A \lor B & \leftrightarrow \tau_1 + \tau_2 \\
A \land B & \leftrightarrow \tau_1 \times \tau_2 \\
A \supset B & \leftrightarrow \tau_1 \rightarrow \tau_2 \\
T & \leftrightarrow \text{unit} \\
F & \leftrightarrow \text{void}
\end{align*}
\]

Then there’s a nice correspondence between proofs and terms. Take sums and products as an example:

- To show $A \land B$, you will need a proof of $A$ and along with a proof of $B$. Again, proofs of propositions are just values of corresponding types. So a proof of $A$ amounts to an expression $e : \tau_1$, and a proof of $B$ amounts to an expression $e : \tau_2$. Now the proof of $A \land B$ is will be an expression that contains both pieces of information. Such term is the product $\langle e_1, e_2 \rangle$.

- There are two ways to show $A \lor B$: you either present a proof of $A$ or a proof of $B$. In other words, there are two ways to construct an expression that corresponds to the proof of $A \lor B$. You either construct it using a term $e_1 : \tau_1$, which is a proof of $A$, or using a term $e_2 : \tau_2$, which corresponds to the proof $B$. Such terms are $l \cdot e_1$ and $r \cdot e_2$.

Now the question is, what is the type that corresponds to logical negation? Well, turns out it depends on how you interpret logical negations:

- You can interpret logical negations constructively: $\neg A \leftrightarrow \tau_1 \rightarrow \text{void}$. In this sense, a proof of negation is a term, if eliminated using a proof $A$, provides you a piece of evidence (a value of type $\text{void}$) that witnesses logical inconsistency. The key here is that you are allowed to carry the evidence around.

- You can interpret logical negations classically: $\neg A \leftrightarrow \tau_1 \text{ cont}$. In this sense, a logical negations is an assertion of refutation: it bluntly claims there can be no proof of $A$. If a proof for $A$ do exists, that is, a term $e_1 : \tau_1$ do exists, a contradiction is formed and you are forced to backtrack to the point where you made the claim.
2.6 Programming with Continuations

To reduce amount of syntax, the following terms are written in forms without modal separations. We can always expands these terms in a modal separation setting.

2.6.1 Contraposition

Consider the proposition:

\[(A \supset B) \supset (\neg B \supset \neg A)\]

**Task 1.** Prove the proposition classically by demonstrating a term of type

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

**Solution:**

\[\lambda(f : \tau_1 \rightarrow \tau_2) \lambda(k : \tau_2 \text{cont}) \text{letcc}\{\tau_1 \text{cont}\}(r.\text{throw}\{\tau_1 \text{cont}\}(f(\text{letcc}\{\tau_1\}(k'.\text{throw}\{\tau_1\}(k'; r))); k))\]

Let’s observe what happens when we activate the term: Suppose we invoke the term with a function \(f : \tau_1 \rightarrow \tau_2\) and a continuation \(k_0 : \tau_2 \text{cont}\).

- After we substituted in the arguments, suppose we begin with
  \[k \triangleright \text{letcc}\{\tau_1 \text{cont}\}(r.\text{throw}\{\tau_1 \text{cont}\}(f(\text{letcc}\{\tau_1\}(k'.\text{throw}\{\tau_1\}(k'; r))); k_0))\]

- By the dynamics of \text{letcc}:
  \[k \triangleright \text{throw}\{\tau_1 \text{cont}\}(f(\text{letcc}\{\tau_1\}(k'.\text{throw}\{\tau_1\}(k'; \text{cont}(k)))); k_0)\]
  \[k; \text{throw}\{\tau_1 \text{cont}\}(\neg; k_0) \triangleright f(\text{letcc}\{\tau_1\}(k'.\text{throw}\{\tau_1\}(k'; \text{cont}(k))))\]
  \[k; \text{throw}\{\tau_1 \text{cont}\}(\neg; k_0); f(\neg) \triangleright \text{letcc}\{\tau_1\}(k'.\text{throw}\{\tau_1\}(k'; \text{cont}(k)))\]

- Define \(k'\) as “\(k; \text{throw}\{\tau_1 \text{cont}\}(\neg; k_0); f(\neg)\)”, By the dynamics of \text{letcc} once again:
  \[k; \text{throw}\{\tau_1 \text{cont}\}(\neg; k_0); f(\neg) \triangleright \text{throw}\{\tau_1\}(\text{cont}(k'); \text{cont}(k))\]

- By the dynamics of \text{throw}:
  \[k \triangleright \text{cont}(k')\]

Look at our return value. \text{cont}(k') expands to \text{cont}(k; \text{throw}\{\tau_1 \text{cont}\}(\neg; k_0); f(\neg)). It’s a continuation that takes a value of type \(\tau_1\), applies function \(f\) to it and then send the value to \(k_0\). The caller’s continuation \(k\) is dropped in the process. Essentially contraposition allows you to “prepend a function” \(f\) to an existing continuation.
2.6 Programming with Continuations

Task 1. Prove the proposition constructively by demonstrating a term of type, without using continuations:

\[(\tau_1 \to \tau_2) \to ((\tau_2 \to \text{void}) \to (\tau_1 \to \text{void}))\]

Solution: \(\lambda (f : \tau_1 \to \tau_2) \lambda (g : \tau_2 \to \text{void}) \lambda (x : \tau_1) \text{case } g(f(x)) \{\}\).

The existence of a term without the usage of continuations suggests that the proposition holds constructively.

Consider the proposition:

\[\neg A \supset \neg B \supset (B \supset A)\]

Task 3. Prove the proposition classically by demonstrating a term of type

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

Solution: \(\lambda (f : \tau_1 \text{ cont} \to \tau_2 \text{ cont}) \lambda (x : \tau_2) \text{letcc}\{\tau_1\}\{r.\text{throw}\{\tau_2\}(f(r); x)\}\)

Task 4. Prove the proposition constructively by demonstrating a term (without using continuations) of type

\[((\tau_1 \to \text{void}) \to (\tau_2 \to \text{void})) \to (\tau_2 \to \tau_1)\]

Solution: Hint: You can’t do it. The proposition does not hold constructively.

Task 5. Prove the proposition by demonstrating a term (with the help of continuations) of type

\[((\tau_1 \to \text{void}) \to (\tau_2 \to \text{void})) \to (\tau_2 \to \tau_1)\]

Solution: The solution is left as an exercise for the reader. Hint: you will need law of excluded middle or equivalent constructs.

2.6.2 Law of Excluded Middle

Exhibit a term of type

\[\tau + \tau \text{ cont}\]

Hint: There are two ways to bluff:

Solution 1.

\[\text{letcc}\{(\tau + \tau \text{ cont})\}(r. \text{letcc}\{\tau_1\}\{r'.\text{throw}\{\tau\}(R \cdot r'; r)\})\]

Notice the term is not a value! Let’s run the dynamics to see what happens. To make it more fun, let’s call our initial continuation \(me\) because apparently dynamics of \(K\) machine is really a two-person’s game!

- We start with

\[me \triangleright \text{letcc}\{(\tau + \tau \text{ cont})\}(r. \text{letcc}\{\tau_1\}\{r'.\text{throw}\{\tau\}(R \cdot r'; r)\})\]
• By the dynamics of letcc:

\[ \text{me} \triangleright L \cdot \text{letcc}\{\tau_1\}(r'.\text{throw}\{\tau\}(R \cdot r';\text{cont}(me))) \]

\[ \text{me}; L \cdot - \triangleright \text{letcc}\{\tau_1\}(r'.\text{throw}\{\tau\}(R \cdot r';\text{cont}(me))) \]

• By the dynamics of letcc:

\[ \text{me}; L \cdot - \triangleright \text{throw}\{\tau\}(R \cdot \text{cont}(me); L \cdot -);\text{cont}(me) \]

• Now \( R \cdot \text{cont}(me; L \cdot -) \) is already a value. According to the dynamics of throw:

\[ \text{me} \triangleright R \cdot \text{cont}(me; L \cdot -) \]

• Finally the right hand side is returned to you as a value:

\[ \text{me} \triangleleft R \cdot \text{cont}(me; L \cdot -) \]

Look at the continuation. It lie to your face (literally, in our setting) that every proposition is false. Before lying to you, it captures a snapshot of you. If you are able to refute it, instead of providing you an evidence of contradiction, it rollbacks the entire world by activating the snapshot. However, this time, it changes it’s mind due to the prepended \( L \cdot - \) frame.

**Solution 2.** Fill in the dots:

\[ \text{letcc}\{(\tau + \tau \text{ cont})\}(r. R \cdot ...) \]

*Answer:* (Please rotate the paper)

\[ (((a:\tau_1 \rightarrow \tau_2) \rightarrow (\tau_2 \text{ cont} \rightarrow \tau_1 \text{ cont})) \text{letcc}\{(\tau + \tau \text{ cont})\}(r. R \cdot a)) \]

Now this should be part of your instinct at this point: whenever you have two seemingly different things that describes the same idea, you should feel obligated to show either they are the under the hood the same thing, or they are actually different (in the latter case you just discovered something non-trivial). Convince yourselves that the two solutions are really the same thing by running the dynamics.

**Solution 3.** Recall that running solution 1 tells us that what we need is really a continuation that appends \( L \cdot - \) prepended to external continuation. This suggests that the machinery we developed in task 1 may be useful. This makes sense because The law of excluded middle is in fact derivable from contraposition. This is witness by the possibility to formulate a term of type \( \tau + \tau \text{ cont} \) using a term for contraposition.

Suppose we have the term:

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

where

\[ \text{lem} \triangleq \text{letcc}\{\tau + \tau \text{ cont}\}(r. R \cdot \text{cp}(f)(r)) \]

and

\[ f : \tau \rightarrow \tau + \tau \text{ cont} \triangleq \lambda (x : \tau) L \cdot x \]
Remarks. Recall Task 4 and Task 5. The fact that using continuations allows you create a term that is otherwise impossible to exhibit shows that use of negations in classical sense forbids us from distinguishing constructive terms against terms that involves proof by contradictions. In other words:

- Just by looking at the type of the term Task 5, you would think it’s a constructive proof because there are no continuations in the type signature.
- Yet, the proof implicitly uses law of excluded middle, and you have no way to separate these things!
- It also degenerates the strength of constructively provable propositions. For example, in Task 3, you proved a proposition without using lem. As you can imagine, there exists a way to prove it WITH lem, and you have no way tell them apart.

Classical negations robs us of the ability to tell constructively valid proposition and classically valid propositions apart.

3 Parallelism

Parallelism allows parts of a program that do not depend on each other to execute simultaneously, often decreasing the running time (span) of the program. Every parallel program has a sequential semantics; it will evaluate to the same value when run sequentially or run in parallel. This make parallelism distinct from concurrency, in which programs do not necessarily have a sequential semantics. In simpler terms:

Parallelism concerns sequencing of computation; Concurrency, as we will see in the following weeks, concerns communications between programs.

In this recitation, we explore a form of parallelism called nested or fork-join parallelism, in which a multiple parallel computations are forked and evaluated in parallel, and then their results are joined together.

4 PPCF

PPCF extends PCF with a lazy product (along with a multi-ary bind) that can be thought of a generalization of computation types. For simplicity here we only discuss the binary lazy product. (c.f. “PFPL Supplement: Types and Parallelism”)

4.1 Grammar

Type $\tau ::= \tau_1 \& \tau_2$

Expr $e ::= e_1 \& e_2$

| $\text{parbnd}(v, x_1.x_2,e)$

To reduce confusing, usual (value) products are now formed using $\otimes$ operator.
4.2 Statics

In a modal separated setting:

\[
\begin{align*}
\Gamma \vdash e_1 \sim \tau_1 & \quad \Gamma \vdash e_2 \sim \tau_2 \\
\Gamma \vdash e_1 \& e_2 : \tau_1 \& \tau_2 & \\
\Gamma \vdash \text{parbnd}(v; x, e) \sim \tau
\end{align*}
\]

Lazy products can be thought of generalizations of computations types: they allow you to suspend multiple computations and wrap them up in an value. \texttt{parbnd} is a generalization of a sequential \texttt{bnd}: it evaluates both computations to values, form a value product using both values, then substitute it into the expression \( e \).

4.3 Dynamics

Now the dynamics of \texttt{parbnd} is left unspecified: the key thing to notice is that the computation \( e \) depends on values resulting from the suspended computations, but not any specific evaluation orders.

**Sequential Dynamics**  We could evaluate them sequentially in some arbitrary order:

\[
\begin{align*}
& e_1 \mapsto e_1' & v_1 \text{ val } \quad e_2 \mapsto e_2' \\
\text{parbnd}(e_1 \& e_2) & \mapsto \text{parbnd}(e_1' \& e_2')
\end{align*}
\]

And the terminal rule:

\[
\begin{align*}
& v_1 \text{ val } \quad v_2 \text{ val } \\
\text{parbnd}(v_1 \& v_2) & \mapsto [v_1 \oplus v_2/x]e
\end{align*}
\]

**Parallel Dynamics**  We may also allow both of the components to take steps independently:

The parallel rule and terminal rule:

\[
\begin{align*}
& e_1 \mapsto e_1' \quad e_2 \mapsto e_2' \\
\text{parbnd}(e_1 \& e_2) & \mapsto \text{parbnd}(e_1' \& e_2')
\end{align*}
\]

\[
\begin{align*}
& v_1 \text{ val } \quad v_2 \text{ val } \\
\text{parbnd}(v_1 \& v_2) & \mapsto [v_1 \oplus v_2/x]e
\end{align*}
\]

Catch-up rules:

\[
\begin{align*}
& v_2 \text{ val } \quad e_1 \mapsto e_1' \\
\text{parbnd}(e_1 \& v_2) & \mapsto \text{parbnd}(e_1' \& v_2)
\end{align*}
\]

\[
\begin{align*}
& v_1 \text{ val } \quad e_2 \mapsto e_2' \\
\text{parbnd}(v_1 \& e_2) & \mapsto \text{parbnd}(v_1 \& e_2')
\end{align*}
\]

This particular formulation well justifies modal separation: the difference between a parallel language and a sequential one really boils down to how the dynamics regarding (n-ary) computation types are formulated.