Recitation 10: Continuations and Parallelism

15-312: Foundations of Programming Languages

Jeanne Luning Prak

March 28, 2018

1 Continuations

Last week, we saw how we can use control stacks to describe exception raising and handling. In particular, control stacks allowed us to implement non-local jumps, aborting the evaluation of the current expression and moving to a handler somewhere else in the program. This week, we discuss another 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.

$$k \triangleright \texttt{mult\_list}(L)$$

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

$$k; \texttt{mult}(s(z); -); \texttt{mult}(s(s(z)); -); \texttt{mult}(s(z); -) \triangleleft z$$

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

$$\begin{array}{rrcl} \text{Type} & \tau & ::= & \tau\ \texttt{cont} \\ \text{Expr} & e & ::= & \texttt{letcc}\{\tau\}(x.e) \\ & & & \texttt{throw}\{\tau\}(e_1; e_2) \\ & & & \texttt{cont}(k) \end{array}$$

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

2.2 Statics

$$\frac{\Gamma, x : \tau\ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\{\tau\}(x.e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\ \texttt{cont}}{\Gamma \vdash \texttt{throw}\{\tau\}(e_1; e_2) : \tau}$$

Just like with `raise`, the type of a `throw` can be arbitrary, since it does not evaluate to a value.

2.3 Dynamics

The two most interesting rules concern how stacks are bound in `letcc` and replaced in `throw`:

$$\overline{k \triangleright \texttt{letcc}\{\tau\}(x.e) \longmapsto k \triangleright [\texttt{cont}(k)/x]e} \qquad \overline{k; \texttt{throw}\{\tau\}(v; -) \triangleleft \texttt{cont}(k') \longmapsto k' \triangleleft v}$$

Note that the second rule implies that $\texttt{cont}(k)$ is a value. The rest of the rules can be seen in Chapter 30 of *PFPL*, and concern evaluating the arguments to `throw`.

[1] Since, as everyone knows, continuation starts with k.

2.4 Example

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

Note: For simplicity, we'll treat `natlist` as a primitive, though it could be encoded in KPCF as a function, or by introducing inductive types.

```
fn (L : natlist) letcc{nat}(x.
    (fix mult_list : natlist -> nat is
        fn (L' : natlist) case L' of
                              [] => s(z)
                              | y::ys => ifz(y;
                                             throw{nat}(z)(x);
                                             _.mult(y)(mult_list ys)))(L))
```

Notice that we've made a separate recursive helper function inside of our `letcc`, applied it to the argument of the function. This is because we need `letcc` to be outside of the recursive function, or we would not get any benefit from invoking `throw`.

3 Parallelism

Parallelism allows parts of a program that do not depend on each other to execute simultaneously, often increasing the running time 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 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 extends PCF with a *parallel let* which forks two parallel processes and joins them after computing their results

4.1 Grammar

$$\text{Expr} \quad e \quad ::= \quad \texttt{par}(e_1; e_2; x_1.x_2.e)$$

4.2 Statics

The statics of parallel let are almost identical to the statics of sequential let, except for two variables are bound in e instead of one.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{par}(e_1; e_2; x_1.x_2.e) : \tau}$$

4.3 Dynamics

Because parallel programs always have a sequential meaning as well as a parallel meaning, we can define both a sequential and a parallel dynamics for `par`.

Sequential Dynamics The sequential dynamics are identical to the dynamics for `let` in other languages, so we omit them here. They can be found in Chapter 37 of *PFPL*.

Parallel Dynamics We describe the parallel dynamics using a different form of dynamics than previously. Before, we defined dynamics in terms of two judgments: `val` and $\longmapsto$. Here, we'll define the dynamics using *evaluation dynamics*, which, instead of describing an individual step, describe the result of evaluating expressions to a value.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow v}{\texttt{par}(e_1; e_2; x_1.x_2.e) \Downarrow v}$$

5 Cost Dynamics

The main advantage of evaluation dynamics over transition dynamics (the system with `val` and $\longmapsto$) is that they give us an easy way of expressing the time cost of evaluating an expression. This is particularly important for parallelism, as the only difference between parallel and sequential evaluation is the runtime cost.

To do this, we annotate the $\Downarrow$ with the *cost graph* for an expression, which represents the number of steps required to evaluate an expression and the opportunities for parallelism.

We define a cost graph using the following grammar:

5.1 Grammar

Cost	c	::= $\mathbf{0}$	Zero Cost
		$\mathbf{1}$	Unit Cost
		$c_1 \otimes c_2$	Parallel Combination
		$c_1 \oplus c_2$	Sequential Combination

5.2 Dynamics

The cost dynamics for `par` are given below.

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_1 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\texttt{par}(e_1; e_2; x_1.x_2.e) \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v}$$

If it costs c_1 steps to evaluate e_1 and c_2 steps to evaluate e_2, and c steps to evaluate the substitution into e, then the cost of evaluating the `par` is $(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c$.

5.3 Work and Depth

We can define *work*, the number of steps it takes to evaluate an expression sequentially, and *depth* (also called *span*), which is the longest chain of dependencies in a program. Depth is a lower bound on the parallel complexity of evaluating an expression.

We can define them in terms of cost graphs using the following equations. Work is defined as

$$wk(c) = \begin{cases} 0 & \text{if c} = \mathbf{0} \\ 1 & \text{if c} = \mathbf{1} \\ wk(c_1) + wk(c_2) & \text{if c} = c_1 \otimes c_2 \\ wk(c_1) + wk(c_2) & \text{if c} = c_1 \oplus c_2 \end{cases}$$

and depth is defined as

$$dp(c) = \begin{cases} 0 & \text{if c} = \mathbf{0} \\ 1 & \text{if c} = \mathbf{1} \\ \max(dp(c_1), dp(c_2)) & \text{if c} = c_1 \otimes c_2 \\ dp(c_1) + dp(c_2) & \text{if c} = c_1 \oplus c_2 \end{cases}$$

Note that the depth takes the max of nodes representing parallel composition, but still adds together the nodes representing sequential composition.