CS611 Lecture 30 **Continuations Continued** 12 November 2003

Scribe: Karen McKeen, Jonathan Moon Lecturer: Radu Rugina

1 Continuation Passing Style (CPS) Semantics

In the previous lecture, we provided the CPS semantics for several constructs in the FUN language. We complete that presentation here. Recall that:

$$\begin{aligned} \mathcal{C}[\![e]\!] &: \mathit{Env} \to \mathit{Cont} \to \mathit{Result} \\ \mathit{Functions} &: \mathit{Value} \to \mathit{Cont} \to \mathit{Result} \end{aligned}$$

The CPS denotations for abstractions, applications, and recursive functions are:

$$\begin{aligned} \mathcal{C}[\![\lambda x.e]\!]\rho\kappa &= \kappa\ (\lambda v.\lambda\kappa_{dyn}.\ \mathcal{C}[\![e]\!]\rho[x \mapsto v]\kappa_{dyn}) \\ \mathcal{C}[\![e_0 e_1]\!]\rho\kappa &= \mathcal{C}[\![e_0]\!]\,\rho\,(\mathit{check\_fun}(\lambda f.\,\mathcal{C}[\![e_1]\!]\,\rho\,(\mathit{check\_val}(\lambda v.\ f\ v\ \kappa)))) \\ \mathcal{C}[\![\mathsf{rec}\ y\ (\lambda x.e)]\!]\rho\kappa &= \kappa\ (\mathit{fix}(\lambda f.\,\mathcal{C}[\![\lambda x.\,e]\!]\,\rho[y \mapsto f]\ \mathit{id}))) \end{aligned}$$

Note that in the abstraction definition, the original (lexical) continuation κ is dropped, so that we use the continuation from where the function is called, rather than from where it is defined.

Alternatively, we can "inline" the definition of $\lambda x.e$ and define $\mathsf{rec}\ y\ (\lambda x.e)$ as:

$$\mathcal{C}[\![\mathsf{rec}\ y\ (\lambda x.e)]\!]\rho\kappa \;=\; \kappa\ (\mathit{fix}(\lambda f.\ \lambda v.\ \lambda\kappa_{dyn}.\,\mathcal{C}[\![e]\!]\,\rho[x \mapsto v, y \mapsto f]\ \kappa_{dyn}))))$$

This equivalent definition makes it more explicit that the denotation of a recursive function drops the lexical environment and yields a function which is parameterized on the dynamic continuation.

2 FUN! with CPS

It is relatively straightforward to define denotational semantics for FUN! using continuations. It is first necessary to redefine the following domains:

$$\begin{aligned} \mathcal{C}[\![e]\!] &: \mathit{Env} \to \mathit{State} \to \mathit{Cont} \to \mathit{Result} \\ \mathit{Function} &: \mathit{Value} \to \mathit{State} \to \mathit{Cont} \to \mathit{Result} \end{aligned}$$

In other words, evaluations $\mathcal{C}[\![e]\!]$ take as arguments a naming environment ρ, a state σ, which maps dynamic locations to their values, and a continuation κ, which is invoked with the value of e. Applying the continuation yields a final result in the *Result* domain. This domain is the same as in FUN!:

$$\mathit{Result} \;=\; (\mathit{Value} \times \mathit{State} + \mathit{Error})_\perp$$

The following are example denotations for FUN!.

$$\mathcal{C}[\![\mathsf{ref}\ e]\!]\rho\sigma\kappa \;=\; \mathcal{C}[\![e]\!]\rho\sigma(\mathit{check\_val}\,(\lambda v.\,\lambda\sigma'.\,\mathit{let}\ l = \mathit{alloc}(\sigma')\ \mathit{in}\ \kappa\ (l, \sigma' \cup \{l \mapsto v\}))) \qquad (l \notin \mathit{Dom}(\sigma'))$$

$$\mathcal{C}[\![e_1; e_2]\!]\rho\sigma\kappa \;=\; \mathcal{C}[\![e_1]\!]\rho\sigma(\mathit{check\_val}\,(\lambda v.\,\lambda\sigma'.\,\mathcal{C}[\![e_2]\!]\rho\sigma'(\mathit{check\_val}\,(\lambda v.'\lambda\sigma''.\,\kappa(v', \sigma'')))))$$

Note that *check_val* now actually checks for a value-state pair, rather than just for a value as in FUN.

Semantics for the assignments $e_1 := e_2$ and dereferences $!e$, as well as for all the remaining expressions in the grammar are left as an exercise to the reader.

In the remainder of these notes we will extend the FUN! language with additional control flow constructs, and express their denotations using CPS.

3 Labels and Jumps

We introduce labeled expressions and jump expressions:

$$e \quad ::= \quad \ldots \mid \mathsf{label}\ x\ e \mid \mathsf{goto}\ x\ e$$

The expression $\mathsf{label}\ x\ e$ binds the current continuation to variable x and then evaluates expression e. The construct $\mathsf{goto}\ x\ e$ evaluates e, then transfers the control to the program point where the label expression defines x. To express the semantics of these constructs, we need to extend the domain of *Value* with continuations:

$$\mathit{Value} = \ldots + \mathit{Cont}$$

The denotations for label and goto are very succinct with CPS, since this framework explicitly captures continuations:

$$\begin{aligned}\mathcal{C}[\![\mathsf{label}\ x\ e]\!]\rho\sigma\kappa &= \mathcal{C}[\![e]\!]\rho[x \to \kappa]\sigma\kappa \\ \mathcal{C}[\![\mathsf{goto}\ x\ e]\!]\rho\sigma\kappa &= \mathcal{C}[\![e]\!]\rho\sigma\rho(x)\end{aligned}$$

There is a small problem with the second denotation, because $\rho(x)$ may not be a continuation. We can re-write this denotation to check that $\rho(x)$ is indeed a continuation, or yield an error otherwise:

$$\mathcal{C}[\![\mathsf{goto}\ x\ e]\!]\rho\sigma\kappa = \mathcal{C}[\![e]\!]\rho\sigma(\mathit{check\_val}\,(\lambda v.\,\lambda\sigma'.\,\mathcal{C}[\![x]\!]\rho\sigma'(\mathit{check\_cont}\,(\lambda\kappa'.\,\kappa'\,v))))$$

Or simply:

$$\mathcal{C}[\![\mathsf{goto}\ x\ e]\!]\rho\sigma\kappa = \mathit{kcase}\ \rho(x)\ \mathit{of}\ \mathit{Cont}(\kappa') \Rightarrow \mathcal{C}[\![e]\!]\rho\kappa'$$

Note that our semantics require that the label expression must be evaluated before the goto expression; otherwise, the execution will produce an error. This is different than languages such as C where the corresponding label and goto constructs allow labels to occur after goto's.

Although contested by Wojciech, we claim that it would be difficult to model jumps with standard operational or denotational semantics, while the definitions using continuations are straightforward. The difficulty with expressing the meaning of such constructs with operational semantics is that one must encode continuations in the machine configuration. For instance, in an imperative language, one can model a continuation as a pair of a program and a point in that program: the continuation that program point p represents is the execution of the program starting at that point.

4 Setjump and Longjump

The constructs setjmp and longjmp allow a program to make non-local jumps - that is, jump into another function. The expression $\mathsf{setjmp}\ e$ evaluates e evaluates to a heap location, then stores the current continuation in that location, and yields value true. The expression $\mathsf{longjmp}\ e$ evaluates e evaluates to a heap location, then passes false to the continuation stored in that heap location. Hence, the return values true and false differentiate between the two ways of "returning" from a setjmp: in the initial pass through the program or via a longjmp.

Such constructs are available as library calls in C. Instead of true and false, they return 0 or a non-zero code. Here is an example program:

```
#include <setjmp.h>
jmp_buf buf;

void foo() {
   if (setjmp(buf) == 0) {
      compute();
   } else {
      handle_error();
   }
}
```

```
compute() {
   ...
   if (error)  longjmp(buf, 1);
   ...
}
```

There is an obvious similarity between this program and exceptions. In fact, setjmp/longjmp is way to implement exceptions. Such an implementation has the advantage to be portable, since most C systems support these functions.

We extend FUN! with setjmp and longjmp constructs as follows:

$$e \quad ::= \quad ... \mid \mathsf{setjmp}\ e \mid \mathsf{longjmp}\ e$$

The denotations for these are:

$$\begin{aligned}
\mathcal{C}[\![\mathsf{setjmp}\ e]\!]\rho\sigma\kappa &= \mathcal{C}[\![e]\!]\rho\sigma(\mathit{check_loc}(\lambda l.\,\lambda\sigma'.\,\kappa(\mathit{true},\sigma'[l \to \kappa])))\\
\mathcal{C}[\![\mathsf{longjmp}\ e]\!]\rho\sigma\kappa &= \mathcal{C}[\![e]\!]\rho\sigma(\mathit{check_loc}(\lambda l.\,\lambda\sigma'.\,\mathit{kcase}\ \sigma'(l)\ \mathit{of}\ \mathit{Cont}(\kappa') \to \kappa'(\mathit{false},\ \sigma')))
\end{aligned}$$

5 Call with Current Continuation

Call-with-current-continuation is a construct present in Scheme and ML. The construct callcc e_f takes a function that has a continuation as an argument and invokes the function with the current continuation as an argument. The construct throw e_k e_v takes a continuation and a value as inputs and calls the continuation with that value.

$$e \quad ::= \quad ... \mid \mathsf{callcc}\ e_f \mid \mathsf{throw}\ e_k\ e_v$$

Here is an example ML program that uses callcc. This program computes the product of all values in a list of integers. If a list element is 0, then the execution shortcircuits with callcc and directly returns 0:

```
fun mult l =
   callcc (fn ret =>
      let fun mult' [] = 1
            | mult'(0::t) = throw ret 0
            | mult'(n::t) = n * mult'(t)
      in
         mult' l
      end
   )
```

The semantics for callcc will be provided in the next lecture.