CS611 Lecture 29 **CPS Semantics** 10 November 2003

Scribe: Revelino Guron, Jr. Lecturer: Radu Rugina

1 Continuation-Passing Style

So far, we've discussed the formal semantics for constructs with structured (or local) control-flow, such as if commands, while loops, of function calls. Next, we would like to express the semantics of language constructs such as goto's or exceptions, which have unstructured (or non-local) flow of control. To make it easier to express the semantics of these constructs, we will introduce another kind of semantics, called continuation passing style semantics (CPS).

Continuations represent the main concept in this style of semantics. A continuation models the remaining computation in the program at a certain point during the execution of the program. Consider the following example program.

$$\mathsf{fact\ n} \ \equiv \ \mathsf{if\ (n = 0)\ then\ 1\ else\ n * fact(n-1)}$$

The evaluation of this expression proceeds by first evaluating the test $\mathsf{n = 0}$, and then executing one of the branches, depending on the result of the test. The computation remaining after the evaluation of the test is the continuation at that point. To make this continuation explicit, we can re-write this program as:

$$(\lambda \mathsf{b.\ if\ b\ then\ 1\ else\ n * fact(n-1)})\ (\mathsf{n = 0})$$

This explicitly shows that we need to evaluate $\mathsf{n = 0}$, then pass the result to the continuation $(\lambda \mathsf{b.\ if\ b\ then\ 1\ else\ n* fact(n-1)})$.

2 Example

For a better intuition behind continuations, we present an example ML program written with explicit continuations. This program adds up the all the numbers in a list of integers:

```
fun sum s =
    let fun sum' [] k = k 0
          | sum' (n::t) k = sum' t (fn x => k(x+n))
    in
        sum' s (fn x => x)
    end
```

The helper function $\mathtt{sum}'$ takes two arguments, a list `l` and a continuation `k`. The function sums up the elements in l and then calls the continuation `k` with the result of the summation. Finally, the original function `sum` calls `sum'` with the identity function as the continuation.

3 Semantics

We will next define the semantics for the FUN language (without dynamic allocation). With CPS, the denotation of an expression will take both a variable environment and a continuation as arguments:

$$\mathcal{C}[\![e]\!] : \mathit{Env} \to \mathit{Cont} \to \mathit{Result}$$

where the domain of continuations is:

$$\mathit{Cont} = \mathit{Result} \to \mathit{Result}$$

We also need to change the domain for functions. With CPS, a function takes as arguments a variable environment, an input value, and the continuation to call when the function is ready to return:

$$\mathit{Function} \ = \ \mathit{Value} \to \mathit{Cont} \to \mathit{Result}$$

The evaluation of a program e is:

$$\mathcal{C}[\![\rho]\!]_0\kappa_0$$

The continuation κ_0 that we pass when we evaluate the whole program is called the *halt continuation.* Typically, this is the identity function $\kappa_0 = \lambda x.\, x$, as in the example above. However, κ_0 can be any other continuation we want. For example, κ_0 can be the function that returns zero if the computation returns an error and one otherwise. Then, all we know about the program is whether or not the computation ended in an error.

The CPS semantics for integer constants and variables is:

$$\begin{aligned} \mathcal{C}[\![n]\!]\rho\kappa &= \kappa\ n \\ \mathcal{C}[\![x]\!]\rho\kappa &= \kappa\ (\rho\ x) \end{aligned}$$

The CPS denotation for an if command must first evaluate the condition expression, then pass the result to the continuation k':

$$\mathcal{C}[\![\textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2]\!]\rho\kappa \;=\; \mathcal{C}[\![e_0]\!]\rho\kappa'$$

where the continuation κ' chooses one of the two branches, as shown in the example at the beginning of the lecture:

$$\begin{aligned} \kappa' \;=\; \lambda x \in \mathit{Result}.\ \mathit{kcase}\ x \quad \mathit{of} \quad & \mathbb{T}(\mathit{true}) \Rightarrow \mathcal{C}[\![e_1]\!]\rho\kappa \\ \mid \quad & \mathbb{T}(\mathit{false}) \Rightarrow \mathcal{C}[\![e_2]\!]\rho\kappa \end{aligned}$$

The construct *kcase* is similar to *ecase* we used before, but it acts slightly differently for non-termination and errors in that it invokes the halt continuation in these cases. If the argument expression yields $\perp$ (that is, the evaluation of the expression doesn't terminate), the *kcase* construct returns $\kappa_0\ \perp$. If the argument expression evaluates to an error, then the construct returns $\kappa_0\ \mathit{error}$.

We can re-write the denotation of the if construct more compactly by defining a function *check_bool* that filters out all non-boolean arguments. The function *check_bool* checks for errors and non-termination using *kcase*:

$$\begin{aligned} \mathit{check\_bool} \quad &: \quad (\mathbb{T} \to \mathit{Result}) \to \mathit{Result} \to \mathit{Result} \\ \mathit{check\_bool} \quad &= \quad \lambda f.\, \lambda v.\ \mathit{kcase}\ v\ \mathit{of}\ \mathbb{T}(t) \Rightarrow f\ t \end{aligned}$$

Now, we can simply write the denotation of if as:

$$\mathcal{C}[\![\textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2]\!]\rho\kappa \;=\; \mathcal{C}[\![e_0]\!]\rho\ (\mathit{check\_bool}(\lambda b.\, \mathit{cond}(b,\ \mathcal{C}[\![e_1]\!]\rho\kappa,\ \mathcal{C}[\![e_2]\!]\rho\kappa)))$$

Let's continue defining the semantics of more expressions. The denotation of an arithmetic operation is:

$$\mathcal{C}[\![e_0\ \oplus\ e_1]\!]\rho\kappa \;=\; \mathcal{C}[\![e_0]\!]\rho\kappa'$$

where κ' is the continuation that takes the result of evaluating the first argument and performs the rest of the computation:

$$\kappa' = \lambda r\ \in\ \mathit{Result}.\, \mathit{kcase}\ r\ \mathit{of}\ \mathbb{Z}(n_0) \Rightarrow \mathcal{C}[\![e_1]\!]\rho\kappa_{n_0}$$

Here κ_{n_0} is the continuation that takes the result of evaluating the second argument and performs the rest of the computation. The value n_0 of the first argument is a constant in this continuation.

$$\kappa_{n_0} = \lambda r\ \in\ \mathit{Result}.\, \mathit{kcase}\ r\ \mathit{of}\ \mathbb{Z}(n_1) \Rightarrow \kappa\ (n_0\ +\ n_1)$$

Again, we can re-write the definition in terms of our check functions. We won't define any more of the check functions, but all of them are similar to one for booleans that we've sown above:

$$\begin{aligned} \mathcal{C}[\![e_0\ \oplus\ e_1]\!]\rho\kappa \quad &= \quad \mathcal{C}[\![e_0]\!]\rho\ (\mathit{check\_int}(\lambda n_0.\, \mathcal{C}[\![e_1]\!]\rho\ (\mathit{check\_int}(\lambda n_1.\, \kappa\ (n_0\ \oplus\ n_1))))) \\ \mathcal{C}[\![\langle e_0,\ e_1\rangle]\!]\rho\kappa \quad &= \quad \mathcal{C}[\![e_0]\!]\rho\ (\mathit{check\_value}(\lambda v_0.\, \mathcal{C}[\![e_1]\!]\rho\ (\mathit{check\_value}(\lambda v_1.\, \kappa\langle v_0,\ v_1\rangle)))) \\ \mathcal{C}[\![\textsf{fst } e]\!]\rho\kappa \quad &= \quad \mathcal{C}[\![e]\!]\rho\ (\mathit{check\_pair}(\lambda p.\, k\ (\pi_1\ p))) \end{aligned}$$

So forth and so forth... Let's move onto something more interesting: the denotation of an abstraction. As mentioned, functions now also take a continuation as an argument:

$$\mathcal{C}[\![\lambda x.\, e]\!]\rho\kappa \;=\; \kappa\ (\lambda v \in \mathit{Value}.\, \lambda\kappa_{dyn} \in \mathit{Cont}.\, \mathcal{C}[\![e]\!]\rho[x \mapsto v]\ \kappa_{dyn})$$

We will continue with the denotation of applications and recursive abstractions in the next lecture.