Scribe: Kamen Yotov and Rohit Fernandes Lecturer: Andrew Myers

Introduction

In the previous lecture, we looked at continuations as a programming language construct by defining an extended version of the λ-Calculus.In this lecture, we try to use continuation as a feature used for describing denotational semantics. Continuation can be thought of as the part of the program yet to be executed. Continuations help us to model non-hierarchical control transfers in programs.

Continuation is modelled as a function whose domain equation is given by :

$$Cont = Result \rightarrow Answer$$

We do not actually care about the domain of *Answer*. So, continuation can also be thought of as a function which does not return as we are usually not concerned with the return value while writing the semantics. Hence, we are free to choose any domain for *Answer*. A possible choice is taking *Answer* to be identical to the domain of *Result*. Another possibility is the *unit* domain. The domain equations pertaining to these two choices are shown below.

$$Answer = Result \quad (1)$$
$$Answer = \mathbb{U} \quad (2)$$

Recall our definition for *Result*:

$$Result = Value + Error$$

The general form of the CPS semantics are of the form

$$\mathcal{C}[\![e]\!]\rho k$$

So our meaning function for an expression takes an environment and a continuation. The understanding is that the expression e is evaluated and the value obtained is passed to the continuation K. Hence, the domain of the meaning function is given by

$$\mathcal{C}[\![e]\!] \in Env \rightarrow Cont \rightarrow Answer$$

Here *Env* can be thought of as the naming context that is the domain from which all variables get their meaning and *Cont* can be thought of as the control context which will become clearer later.

When we write the denotation of a program, we are particularly interested in whether it terminates or not and in the case that it terminates, the particular value it evaluates to. Unfortunately, continuations do not allow us to care about what they evaluate to. So, we need to define a special continuation called the halting continuation which is denoted by K_0. This is

the continuation in which all programs are evaluated. K_0 is the only continuation in which we care about what *Answer* evaluates to,namely the result of the program. So K_0 is a member of

cont as shown below :

$$K_0 \in Cont$$

It follows that the denotation of a program is given by

$$\mathcal{C}[\![e]\!]\rho_0 K_0$$

So, K_0 is analogous to the special value *halt* that was defined for the extended λ-calculus in the previous lecture.

For the rest of the lecture, we will try to come up with a CPS semantics for the language uF that we have seen before.

Recall the domain for values in our semantics for uF:

$$Value = \mathbb{U} + \mathbb{Z} + Pair + Function + \mathbb{T}$$

While all other values have the same meaning, we will now model functions as having two arguments.The first argument will be the return continuation, that is, the continuation where the value the function body evaluates to will be passed. The second argument is the actual argument the function uses in evaluating its body. So the domain equation for functions is

$$Function = Value \rightarrow Cont \rightarrow Answer$$

We will first write the CPS semantics for uF using Call-by-Value style of parameter passing. We will then show how to tranform these semantics to the Call-by-Name case. Finally we will look at how to extend these semantics to describe uF!

CPS Semantics for Call-by-Value uF

First we write the two easy rules for the cases where e is an integer n or a variable x. When e is an integer, the meaning function simply passes it to the continuation K that it is supplied with. When e is a variable x, the meaning function first reduces x to a value by looking up the environment ρ that it is supplied with and then passing the value to the continuation K that it is supplied with. The equations pertaining to these two cases are given by :

$$\begin{aligned} \mathcal{C}[\![n]\!]\rho K &= K\lfloor n \rfloor \\ \mathcal{C}[\![x]\!]\rho K &= K\lfloor \rho x \rfloor \end{aligned}$$

When e is of the form $if\ e_0\ e_1\ e_2$, then we first evaluate e_0 in the environment ρ and then pass it to a new continuation which evaluates e_1 if e_0 evaluates to true and e_2 if e_0 evaluates to false. However, we also need to check for errors in the evaluation of e_0 and the meaningful case becomes pretty obscure with the inclusion of various error cases as shown below :

$$\begin{aligned} \mathcal{C}[\![if\ e_0\ e_1\ e_2]\!]\rho K = \mathcal{C}[\![e_0]\!]\rho(\lambda r.case \quad & r\ of \\ \bot \ . \ & K_0 \bot \\ value(t) \ . \ & if\ t\ then\ \mathcal{C}[\![e_1]\!]\rho K\ else\ \mathcal{C}[\![e_2]\!]\rho K \\ value(u) \ . \ & K_0\ error \\ value(n) \ . \ & K_0\ error \\ value(f) \ . \ & K_0\ error \\ value(p) \ . \ & K_0\ error) \end{aligned}$$

This is tedious. We can encapsulate the above as

$$\begin{aligned} \mathcal{C}[\![if\ e_0\ e_1\ e_2]\!]\rho K = \mathcal{C}[\![e_0]\!]\rho(check\_bool(\lambda b.if\ b\ then \\ \mathcal{C}[\![e_1]\!]\rho K\ else \\ \mathcal{C}[\![e_2]\!]\rho K)) \end{aligned}$$

Here *check_bool* is a function which takes a continuation K that only accepts Boolean values and returns a continuation. The returned continuation if applied to a Boolean value simply passes the Boolean value to K and otherwise terminates the program by passing the value(non-Boolean) to the Halting Continuation K_0. The definition for *check_bool* is as follows:

$$\begin{aligned} check\_bool(K) = \lambda r.case \quad & r\ of \\ \bot \ &. \ K_0 \bot \\ Value(\pi(b)) \ &. \ Kb \\ \vdots \ &. \ K_0\ error \end{aligned}$$

where

$$K \in Bool \to Answer$$

We can similarly define other functions like *check_value*, *check_pair*, etc. as necessary. Many of these will be used in the rest of this lecture.

When the expression is a pair of the form $\langle e_1, e_2 \rangle$, we first evaluate e_1, then we need to check that it evaluates to a value using a *check_value* filter and pass it to a continuation which similarly evaluates and checks e_2. The values obtained for e_1 and e_2 are then passed as a pair $\langle v_1, v_2 \rangle$ to the original continuation K.

$$\mathcal{C}[\![\langle e_1, e_2 \rangle]\!]\rho K = \mathcal{C}[\![e_1]\!]\rho(check\_value(\lambda v_1.\mathcal{C}[\![e_2]\!]\rho(check\_value(\lambda v_2.K\langle v_1, v_2\rangle))))$$

The rule for binary arithmetic expressions of the form $e_1 \oplus e_2$ can similarly be written as

$$\mathcal{C}[\![e_1 \oplus e_2]\!]\rho K = \mathcal{C}[\![e_1]\!]\rho(check\_value(\lambda v_1.\mathcal{C}[\![e_2]\!]\rho(check\_value(\lambda v_2.K(v_1 \oplus_E v_2)))))$$

The rule for expressions of the form *first* e be written as:

$$\mathcal{C}[\![first\ e]\!]\rho K = \mathcal{C}[\![e]\!]\rho(check\_pair(\lambda p.K(\pi_1 p)))$$

Here we need to use the *check_pair* function and the π operator for extracting elements from a tuple.

The rule for function expressions of the form $fn; x; e$ is :

$$\mathcal{C}[\![fn\ x\ e]\!]\rho K = K(\lambda v K'.\mathcal{C}[\![e]\!]\rho[x \mapsto v]K')$$

As function is a value in our domain, the continuation K is simply applied to the value the function evaluates to. Recall that function is a value which takes two arguments, a parameter v and a return continuation K'. So the function evaluates to a λ-expression taking v and K' as arguments and having a body which evaluates the function body in an environment with x having the value of the parameter v and sends it to the return continuation K'.

The rule for function application is:

$$\mathcal{C}[\![e_1\ e_2]\!]\rho K = \mathcal{C}[\![e_1]\!]\rho(check\_fun(\lambda f.\mathcal{C}[\![e_2]\!]\rho(check\_value(\lambda v.fvk))))$$

We first evaluate e_1 and pass it through a $check\_fun$ filter to ensure that it is a function f. This function is then passed to a continuation in which we evaluate e_2 and pass it through a $check_val$ filter to ensure that it is a value v. The values f and v are finally passes through a continuation in which the value obtained by

applying f to v is passes to the original continuation K.

The *rec* expression semantics can be written as shown below. It is similar to the semantics for the function expression $fn\ x\ e$. However, we need to take the fixed point to express the recursion.

$$\mathcal{C}[\![rec\ y\ (fn\ x\ e)]\!]\rho K = K(fix\lambda f.\lambda v\ K'.\mathcal{C}[\![e]\!]\rho[x \mapsto v_1, y \mapsto f]K')$$