CS611 Lecture 24 Non-local control, exceptions October 20, 2000
Scribe: Adina Costea and Cristian Bucilă Lecturer: Andrew Myers

1 Introduction

Last time, we managed to present a continuation passing style semantics for uF, then we added a few more rules for the imperative features of uF$_!$ and obtained a semantics for this language. We found out that we could use CPS semantics to describe a lot of features that we already know how to write semantics for. So, where is the payoff here? Of course there were some nice features about the semantics that we didn't have in the direct style. Still it would be nice to see some new abilities. What we're going to do in this lecture is to give semantics for three different constructs which are going to show why having a continuation passing style is nice.

2 setjmp and longjmp

Let's talk about a feature that can be easily explained using continuations. This is a feature which in C programming language is implemented by setjmp(j) and longjmp(j). setjmp(j) is going to put the current continuation in the store at the location j and returns true - that will be the result you'll get the first time setjmp is called. longjmp(j) is going to transfer control to the continuation found at the location j, and when it does so, it will look as though the setjmp(j) just returned again, but this time it's going to return false. This is a way of describing what happens in setjmp and longjmp in terms of continuations.

The reason we want such a thing is to be able to stop the evaluation of an expression when an error occurs without having to test all kind of error conditions in different places.

```
let j = (ref #u) in
  if(setjmp j)
    (··· big computation ···)
    (··· handle error ···)
```

If the big computation evaluates (longjmp j) at any point, control will immediatly be transfered to the error-handling clause.

We can also use setjmp/longjmp to write other more perplexing code:

```
let j = ref #u in
  let c = ref 0 in
  let y = setjmp j in
    if(!c < 10) (let u = (c := !c + 1) in longjmp j) !c
```

which is just a fancy way of writing a loop.

We want to write the semantics for uF$_!$ + setjmp + longjmp. When we wrote the semantics for uF$_!$ we had things of form $\mathcal{C}[\![e]\!]\rho k\sigma$, where

$\mathcal{C}[\![e]\!] \in Env \to Cont \to Store \to Answer$
$Cont = Result \to Store \to Answer$
$Function = Value \to Cont \to Store \to Answer$

Now we want to write the semantics for setjmp. For that, *Store* has to be allowed to contain continuations. There are two ways of doing it:

$Store = Loc \to Value + Cont + Error$
or
$Value = \cdots + Cont$

We will use the first way. So

$\mathcal{C}[\![\text{setjmp } e]\!]\rho k = \mathcal{C}[\![e]\!]\rho(\textit{check-loc}(\lambda l\sigma.\, k\,(\text{true})\,\sigma[l \mapsto k]))$
$\mathcal{C}[\![\text{longjmp } e]\!]\rho k = \mathcal{C}[\![e]\!]\rho(\textit{check-loc}(\lambda l\sigma.\, (\textit{check-cont}(\lambda k'\sigma'.\, k'\,(\text{false})\,\sigma')(\sigma l))))$

Note: σ was η reduced.

3 calcc and throw

ML also has a similar mechanism called callcc. In (callcc e), the expression e has to evaluate to a function expecting a continuation as its argument. (callcc e) passes the current continuation to that function as a first-class value. We also have an expression (throw e_k e_v). This invokes the continuation e_k passing e_v.

(callcc e)	e evaluates to a function expecting a continuation
(throw e_k e_v)	invokes the continuation e_k passing e_v

In ML, these are actually statically typed and we're not going to deal with modeling that aspect now. What do we need to do in order to add callcc and throw to our language? First of all we need to be able to pass around values that are continuations, because we want to have the ability to send continuations around as first-class values.

$Value = \cdots + Cont$

These features don't do anything special with the store, so we expect that we shouldn't need to talk about the store. We saw last time that as long as the expressions we were writing semantics for didn't actually use the store, we didn't have to talk about the store.

Here is how we compile callcc:

$\mathcal{C}[\![\text{callcc } e]\!]\rho k = \mathcal{C}[\![e]\!]\rho(\textit{check-fun}(\lambda f.\, f\, k\, k))$
$\mathcal{C}[\![\text{throw } e_k\ e_v]\!]\rho k = \mathcal{C}[\![e_k]\!]\rho(\textit{check-cont}(\lambda k'.\, \mathcal{C}[\![e_v]\!]\rho(\textit{check-val}(\lambda v.\, k'\, v))))$

We were able to write down a precise formal specification of what these constructs do. This is nice because you can waste a lot of English sentences trying to explain what they do informally.

4 cwcc

What if we wanted to have this ability to do callcc but we didn't want to have this special throw form in our language? So, what if we wanted the thing that callcc pass to look like a function in our language only that when you call it doesn't return normally instead passes the control somewhere else?

(cwcc e)

It turns out that Scheme has such a mechanism. In Scheme we have something called *call with current continuation* and what it does is: it evaluates e to a function and passes to that function the current continuation of this expression, but it makes that current continuation look like a function so in this case we don't need to touch our notion of what is a value in our language. How do we do that?

$\mathcal{C}[\![\text{cwcc } e]\!]\rho k = \mathcal{C}[\![e]\!]\rho(\textit{check-fun}(\lambda f.\, f\, (\lambda v k'.\, k\, v)\, k))$

These mechanisms look a lot like exception mechanisms. You can see that we can use them to provide a lot of the functionality that we get with exceptions – the ability to terminate an existing computation and return control up an arbitrarily distance in the static and dynamic hierarchy of calls.

5 exceptions

It is natural to ask at this point whether we can model exceptions (like Java exceptions...)

throw s e
try e catch (s x) e'

We need a notion of the exception handling environment.

$h \in \textit{Handlers} = \textit{Symbols} \to \textit{Cont}$
$h_0 = \lambda x.\, k_0$, h_0 being the default exception handler

$\mathcal{C}[\![e]\!] \in \textit{Env} \to \textit{Handlers} \to \textit{Cont} \to \textit{Store} \to \textit{Answer}$
$\textit{Cont} = \textit{Result} \to \textit{Store} \to \textit{Answer}$

$$\mathit{Function} = \mathit{Value} \to \mathit{Handlers} \to \mathit{Cont} \to \mathit{Store} \to \mathit{Answer}$$

$$\mathcal{C}[\![\mathsf{throw}\ s\ e]\!]\rho hk = \mathcal{C}[\![e]\!]\rho h(\lambda r.\, h\, s\, r)$$
$$\mathcal{C}[\![\mathsf{try}\ e\ \mathsf{catch}\ (s\ x)\ e']\!]\rho hk = \mathcal{C}[\![e]\!]\rho h[s \mapsto (\lambda v.\, \mathcal{C}[\![e']\!]\rho[x \mapsto v]h)k]k$$
$$\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]\rho hk = k(\lambda v h' k'.\, \mathcal{C}[\![e]\!]\rho[x \mapsto v]h'k')$$
$$\mathcal{C}[\![e_1\ e_2]\!]\rho hk = \mathcal{C}[\![e_1]\!]\rho h(\mathit{check\text{-}fun}(\lambda f.\, \mathcal{C}[\![e_2]\!]\rho h(\mathit{check\text{-}val}(\lambda v.\, f\, v\, h\, k))))$$

If we look at how h is used in this semantics, we notice that it looks just like the use of ρ in uF with dynamic scope. The function body is evaluated with a handle environment that is dynamic – not the environment in effect when the function term was evaluated. This makes sense because we could simulate exceptions handlers in a language with dynamic scope and first-class continuations putting the handlers into ordinary variables. We argued earlier that dynamic scope is bad for modularity, and in fact it can cause trouble for exception handling too – for example, it is possible to "accidentally" override a dynamically scoped variable.