Matthew S. Harris

Last time we looked at a new way of writing the denotational semantics for a couple of the languages we've seen before. This technique was based on continuations, so rather than the direct semantics that we were looking at before, where we coupled the flow of control of the metalanguage to that of the language itself, we used continuations which explicitly exposed the notion of control transfers. That allows us to talk about control in a more powerful way.

In this semantics we said

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

Its traditional to call the thing a continuation finally produces an *answer* to emphasize that we're in a continuation-based semantics. We were able to write out semantics in which we really didn't care too much what the store was, so in fact we wrote the same semantics and—modulo the rules for ref and the dereferencing operator and assignment—we had exactly the same semantics for F_0 and $F_!$. And you can think of this in a couple of ways. One is that if we say $\mathit{Store} = \mathit{Unit}$, we get F_0—the functional version of the language. And if we say $\mathit{Store} = \mathit{Loc} \to \mathit{Value}$, we get $F_!$. In fact, we're able to η-reduce away the store arguments for the things that didn't actually have to use the store, so we were able to get rid of the store arguments in some sense entirely.

The real reason that we want continuations around is not so we can write one semantics and get two languages for free but because we can model nonlocal control transfers. Examples of these types of things are things like break and continue, which we've attempted to model already; less structured versions of control transfers, like gotos; exceptions and error handling; even things like threads; and some languages also give us first-class continuations, so in ML for example you get something called callcc; in Scheme you get something called cwcc (call with current continuation). And so if we want to model these kinds of control constructs, we need to have continuations around so we can talk about control in a first-class way.

In the semantics we saw last time, there was a continuation being passed around; whenever we evaluated an expression, we expected to have a continuation available. And this continuation was the place where we transfer control after we evaluate the expression; we transfer the value of the expression and the control to that continuation. So this thing is known as the *normal* continuation; this is where control is transferred if the expression terminates normally. And because the normal continuation is an argument to this semantic function $\mathcal{C}$, we can think of this as largely a static thing. We can analyze statically where control is going to be transferred. If you want to model these various nonlocal control constructs, we want to have the continuation exposed as something we pass around dynamically. So in passing around continuations in various dynamic ways, we can model transfers of control that are not static—that they're truly dynamic. We actually already have an example of such a thing. If you look at the domain for *Function*, you'll see that *Function* expects to have a continuation passed to it dynamically, and what that's telling us is that when you return from a function, you can't really tell where you're going. And that's true. When returning from a function, it depends on what the dynamic context is—the dynamic control context in which the function was invoked—that will tell us where to go back to.

1 setjmp and longjmp

What we're going to look at now is some more complicated ways of passing around continuations dynamically, and we can model various things. Last time I introduced a little construct I called label and break, and I think people were a little confused by that because I did it in the last two seconds before class ended. So I'm going to do a slightly modified version of the same thing. We're now going to introduce what I'll call explicitly setjmp and longjmp. setjmp is going to put the current continuation in the store at the location that e evaluates to, and its going to evaluate to false—that will be the result you get when you first call setjmp.

And longjmp is going to transfer control to the continuation in store at the same location, and when it does so, it will look as though the setjmp just returned again, except this time it's going to evaluate to true. This is roughly the way that the actual setjmp/longjmp library calls work in C. This will allow us, when we come back and transfer control to the end of the setjmp, we'll be able to tell whether we're coming back the second time. A typical call would look like:

```
(let (x (ref #u))
  (if (setjmp e)
    (error-handling code)
    (computation that might error...
      (longjmp x))))
```

If the computation is going to err in some way, then you test for that and at the point where you needed to do a test, you do a longjmp on the same e that the setjmp was. When you do this longjmp, the control will transfer back up to this point except now the result of the setjmp will be true, so you'll fall into the error-handling code.

It's very handy, because I can pass this x dynamically. This longjmp doesn't have to occur statically in the text here; it can be down n levels of function calls, dynamically somewhere way off in the code, and as long as I keep passing this x along, then I can do a longjmp to it and bomb up n levels, right up here, bypassing all the stuff in between. This is a way that people write error handlers.

Let's first do one more example to show how evil this construct is. We can actually write a function that will save away the continuation. We'll call this thing the déjà vu operator, and it will construct a location in the store—this location is known as a `jmp_buf` (the type that it has in C)—and we'll do a setjmp, and then we'll return that location. And then we can do things like

```
(let dejavu (fn dummy (let (x (ref #u))
                         (seq (setjmp x) x)))
  (let k (dejavu 0)
    (longjmp k)))
```

What does this do? This turns out to be a very fancy way of writing an infinite loop.

How do we actually describe such a mechanism? Let's write a solution for it directly. (Note that I've reordered the arguments, so continuations now come before the environment, though this won't actually buy us much.)

$$\mathcal{C}[\![\mathsf{setjmp}\ e]\!]k = \mathcal{C}[\![e]\!](\mathsf{Check\text{-}loc}(\lambda l\sigma.\ k(\mathsf{false})\sigma[l \mapsto k]))$$
$$\mathcal{C}[\![\mathsf{longjmp}\ e]\!]k = \mathcal{C}[\![e]\!](\mathsf{Check\text{-}loc}(\lambda l\sigma'.\ \mathsf{case}\ \sigma'(l)\ \mathsf{of}\ \mathsf{Cont}(k') \Rightarrow k'(\mathsf{true})\sigma'\ \mathsf{else}\ k_0\ \mathsf{error}\ \sigma'))$$

2 Exceptions

How about a slightly more structured operation, such as exceptions? It might be nice to actually design a real exception handling mechanism like the one we have in Java. So let's give that a try.

Let's pretend we're in Java now. In Java, exceptions are hidden in the class hierarchy in a funky way, and we don't have classes, so let's not do exactly that. We'll say that (throw s e) throws the exception named s with the value e. And we'll have a catch statement. We'll say what symbol s we're catching, and then we evaluate some expression e_1, and if e_1 terminates with the exception s then we're going to evalute e_2 with x bound to this e from the corresponding throw. You can think of this as being

```
try { e1 } catch (s x) { e2 }
```

That's what we're modelling here.

There are actually a couple of different models that we can use for exceptions. The model that has come into favor is what's known as a *termination model*. The termination model says throw transfers control back to the corresponding catch. So the most recent dynamically invoked catch previous to the throw is where we're going to transfer control to.

There's another model for exceptions which is known as a *resumption model*. In a resumption model, instead of the throw transferring control up to the catch, the catch provides an expression which is supposed to be evaluated in place of the expression that was generating the exception. For example, if you generate a divide-by-zero error, the catch statement would provide an expression to be evaluated in place of the division. So you can say, "if you ever get a divide-by-zero in this block of code, just make it come out to -1 instead of generating a division by zero." It will substitute -1 into that expression for whatever was going on, allowing the computation to proceed without interruption by the exception.

We can actually model both of these different models of exceptions using continuations in a pretty convenient way.

The "matching catch" is very loosely defined here. We know that dynamically there's some catch that's supposed to catch the throw. Exactly specifying that is not that easy, and that's one of the nice things about formal semantics; we can actually capture exactly where the control is supposed to be transferred.

Let's take a crack at this. Let's start out with the termination model, which is a little more normal. First of all, what we need to do is introduce a new kind of environment. This environment is going to keep track of all of the exception handlers that we have in force. We're going to introduce an environment which we'll call *Handlers*, and *Handlers* is going to be a map from symbols to continuations. This will allow us to look up the proper continuation—where we're supposed to transfer control—if we see an exception on some symbol. And we need to extend our semantic function, so what we'll do is :

$$h \in \mathit{Handlers} = \mathit{Symbol} \to \mathit{Cont}$$
$$\mathcal{C}[\![e]\!] : \mathit{Cont} \to \mathit{Env} \to \mathit{Store} \to \mathit{Handlers} \to \mathit{Answer}$$

We also need to augment functions, since these handlers get passed around dynamically.

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

Now we can talk about what it actually means to throw things and catch them. So let's try throwing. After we evaluate e, we get to decide things, like if an exception occurs inside of e, what happens? There are various corner cases to think about. Since we have a formal description of what's going on, we can pin down exactly what it means for an exception to happen inside of e. That will be a question of what $\rho\sigma h$ we pass for e. We'll just assume that exceptions occurring inside e are handled by the same handler that we're using for the throw statement. So if an exception happens inside e, it will actually override the throw, so the throw won't happen, and instead whatever exception that is thrown inside e will be the one that happens.

$$\mathcal{C}[\![\textsf{throw}\ s\ e]\!]k\rho\sigma h = \mathcal{C}[\![e]\!](\textsf{Check-value}(\lambda v\sigma'.\ \text{case}\ h(s)\ \text{of}\ \mathit{Cont}(k') \Rightarrow k'v\sigma'\ \text{else}\ k_0\ \textsf{error}\ \sigma'))\rho\sigma h$$

We could actually change Check-value so that it throws an exception instead of terminating the program when it sees something that is not a value, and allow you to actually catch errors in the middle of execution, but let's not get into that.

We could actually look in the handler environment and see whether there's a handler for the "uncaught exception" exception and then try to throw it on that. Then we would require that every program start out with a handler for uncaught exceptions. Again, that's another complication that we don't want to get into.

How do we actually get the proper exception installed in the handler table? That's going to be done by the catch statement:

$$\mathcal{C}[\![\textsf{catch}\ s\ e_1\ x\ e_2]\!]k\rho\sigma h = \mathcal{C}[\![e_1]\!]k\rho\sigma h[s \mapsto \lambda v\sigma'.\ \mathcal{C}[\![e_2]\!]k\rho[k \mapsto v]\sigma' h]$$

For the final handler environment, we actually have some options, but the natural thing to do is h—the handler environment that doesn't have this exception handler bound into it. That's going to say that if we throw the exception s inside e_2, this catch doesn't try to catch it again.

That's how we model Java-style throw and try-catch statements in denotational semantics. We just had to produce this extra environment which is keeping track dynamically of the set of handlers that are in force.

We actually have to extend the function call and function expression rules slightly, too, to keep track of the fact that there's a handler here:

$$\mathcal{C}[\![\mathsf{call}\ e_1\ e_2]\!]k\rho\sigma h = \mathcal{C}[\![e_1]\!](\mathsf{Check\text{-}fun}(\lambda f.\ \mathcal{C}[\![e_2]\!](\mathsf{Check\text{-}value}(\lambda v\sigma'.\ fvk\sigma' h))))$$

Notice that we pass in the handler environment that was in force at the point of the call. That's important.

We're also going to need to update our evaluation of functions:

$$\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]k\rho\sigma h = k(\lambda vk'\sigma' h'.\ \mathcal{C}[\![e]\!]k'\rho[x \mapsto v]\sigma' h')$$

If we wrote h rather than h' at the end, we would get a completely different semantics: when functions terminated with an exception, they would actually go back and use the handler environment that was in force at the place the function expression occurred in the text.

3 Resumption Semantics

I'm going to skip over the resumption semantics, but I'll just mention that the key thing about the resumption semantics, rather than mapping a symbol to a continuation, what we'll do is map a symbol to a function:

$$h \in Handlers = Symbol \to Function$$

What we want to do is have the catch statement basically build a function that can be called when we generate a throw instead of transferring control up to the catch. So throw will actually pass a continuation up to the catch expression, and the catch will use that continuation to get back down where the throw was.

4 callcc

Let's do a really first-class continuation mechanism. Let's do the ML callcc construct. Actually the setjmp/longjmp I wrote was pretty first-class, too, but let's do a cleaner mechanism for manipulating continuations.

You provide callcc with an expression which is supposed to evaluate to a function. It calls f with the current continuation. So it basically says $f(k)$ for you, but it generates the k, and the idea here is that if f calls k with a value v, then it's as if the callcc terminates with the value v. So when the current continuation comes in, it's going to look just like a function; it's going to look like a function that expects to get some value, but if you call that function, it will never return, and in fact will cause the function f to terminate. The ML version of this is actually statically typed, whereas we're going to have a completely dynamic version here.

$$\mathcal{C}[\![callcce]\!]k = \mathcal{C}[\![e]\!](Check - fun(\lambda f.f(\lambda v'k'\sigma'.kv'\sigma')k))$$

This is a very special function; it's unlike the functions we generate with the fn expression. It's a function that throws away the continuation it was passed and substitutes in this other continuation k to which it sends its result. But otherwise it looks like an ordinary function.

The idea is that this is a way of exposing continuations, so if you want to capture a continuation as a value in the language you can manipulate normally, you can you callcc to snapshot the continuation. So if you were going to build a thread mechanism, for example, you could use callcc to get the current continuation so that you would have something you could call to resume a thread immediately after a callcc. So callcc gives you the ability to capture the current thread of execution.