CS 6110 Lecture 14 Nonlocal Control: Errors and Exceptions 22 February 2009
Lecturer: Andrew Myers

Last time we introduced CPS as a restriction on the λ-calculus. This was helpful because programs written in this restricted λ-calculus have a much simpler operational semantics. In fact, we defined the operational semantics using only a single rule. Another advantage to CPS is that evaluation order decisions are already determined. In general, CPS style is a more primitive model of computation and therefore easier to compile.

Now we can give CPS semantics for uML as a translation to a restricted form of uML. Our translation will also produce strongly-typed uML programs. Then we will extend the translation to uML!. Finally, we show how to extend uML to support exception handling.

1 CPS Semantics for uML with Strong Typing

1.1 Value Translation

To support strong typing, we introduce type tags that can be used to tag each value with its type.

error	0	**null**	1	booleans	2
integers	3	pairs	4	functions	5

We represent a value v as a pair (t, v'), where t is the tag and v' is the target-language term that corresponds to v. Rather than construct these pairs explicitly, we define injection functions *NULL*, *BOOL*, *INT*, *PAIR*, *FUN*, etc. to tag a raw value with its type; for example, $\mathit{BOOL}(\mathbf{true}) = (0, \mathbf{true})$. These injections can all be defined in terms of a function $\mathit{TAG} \triangleq \lambda tx.\,(t, x)$. Then $\mathit{BOOL} = \mathit{TAG}\,2$, etc.

Similarly, we also define functions to check tags: *CHECK-NULL*, *CHECK-BOOL*, *CHECK-INT*, *CHECK-PAIR*, *CHECK-FUN*, etc.. These functions check that a given tagged value is of the correct type, extract the original raw value, and pass it to a continuation. For example, *CHECK-PAIR* is defined as:

$$\mathit{CHECK\text{-}PAIR} \;\triangleq\; \lambda kv.\,\mathbf{let}\ (t, b) = v\ \mathbf{in}\ \mathbf{if}\ t = 4\ \mathbf{then}\ k\ b\ \mathbf{else}\ \mathbf{halt}\ \mathit{ERROR}$$

where the parameter k is a continuation and the parameter v is a tagged value. If the tag is 4, indicating that the raw value is a pair, then we pass the raw value to the continuation. Otherwise we have encountered a runtime type error, so we pass an error value $\mathit{ERROR} = (0, \mathbf{null})$ to the $\mathbf{halt}$ continuation, which ends the program. We can also define these functions uniformly in terms of a function

$$\mathit{CHECK} \;\triangleq\; \lambda tkv.\,\mathbf{if}\ \#1\ v = t\ \mathbf{then}\ k\ (\#2\ v)\ \mathbf{else}\ \mathbf{halt}\ \mathit{ERROR}$$

Then $\mathit{CHECK\text{-}PAIR} = \mathit{CHECK}\,3$, etc.

The precise way that we implement the tagging and checking functions is not crucial. We require only that these implementations satisfy this equation:

$$\mathit{CHECK}\,t\,k\,(\mathit{TAG}\,t'\ v) \;=\; \begin{cases} k\ v, & \text{if } t = t', \\ \mathbf{halt}\ \mathit{ERROR}, & \text{if } t \neq t'. \end{cases}$$

Note that the continuation-passing style affords some flexibility in the way errors are handled. We need not call the continuation k, but may instead call a different continuation ($\mathbf{halt}$ in this example) corresponding to an error or exception handler.

1.2 Expression Translation

Translations are of the form $\mathcal{E}[\![e]\!]\rho k$, which means, "Send the value of the expression e evaluated in the environment ρ to the continuation k." The translations are:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\,\rho\,k &= k\,(\textit{LOOKUP}\,\rho\,\text{“}x\text{”})\\
\mathcal{E}[\![n]\!]\,\rho\,k &= k\,(\textit{INT}\,n)\\
\mathcal{E}[\![(e_1, e_2)]\!]\,\rho\,k &= \mathcal{E}[\![e_1]\!]\,\rho\,(\lambda v_1.\,\mathcal{E}[\![e_2]\!]\,\rho\,(\lambda v_2.\,k\,(\textit{PAIR}\,(v_1,\,v_2))))\\
\mathcal{E}[\![\mathbf{let}(x, y) = e_1\ \mathbf{in}\ e_2]\!]\,\rho\,k &= \mathcal{E}[\![e]\!]\,\rho\,(\textit{CHECK-PAIR}\,(\lambda p.\,\mathbf{let}\ (x', y') = p\ \mathbf{in}\\
&\qquad \mathcal{E}[\![e_2]\!](\textit{EXTEND}\,(\textit{EXTEND}\,\rho\,\text{“}x\text{”}\,x')\,\text{“}y\text{”}\,y')k))\\
\mathcal{E}[\![\lambda x.\,e]\!]\,\rho\,k &= k\,(\textit{FUN}(\lambda y k'.\,\mathcal{E}[\![e]\!]\,(\textit{EXTEND}\,\rho\,\text{“}x\text{”}\,y)\,k'))\\
\mathcal{E}[\![e_0\ e_1]\!]\,\rho\,k &= \mathcal{E}[\![e_0]\!]\,\rho\,(\textit{CHECK-FUN}\,(\lambda f.\,\mathcal{E}[\![e_1]\!]\,\rho\,(\lambda v.\,fvk)))\\
\mathcal{E}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\,\rho\,k &= \mathcal{E}[\![e_0]\!]\,\rho\,(\textit{CHECK-BOOL}\,(\lambda b.\,\mathbf{if}\ b\ \mathbf{then}\ \mathcal{E}[\![e_1]\!]\,\rho\,k\ \mathbf{else}\ \mathcal{E}[\![e_2]\!]\,\rho\,k)).
\end{aligned}$$

2 CPS Semantics for uML!

2.1 Syntax

Since uML! has references, we need to add a store σ to our notation. Thus we now have translations with the form $\mathcal{E}[\![e]\!]\rho k\sigma$, which means, "Evaluate e in the environment ρ with store σ and send the resulting value and the new store to the continuation k." A continuation is now a function of a value and a store; that is, a continuation k should have the form $\lambda v\sigma.\ \cdots$.

The translation is:

- Variable: $\mathcal{E}[\![x]\!]\,\rho\,k\,\sigma = k\,(\textit{LOOKUP}\,\rho\,\text{“}x\text{”})\,\sigma$.

  If we think about this translation as a function and η-reduce away the σ, we obtain

$$\mathcal{E}[\![x]\!]\,\rho\,k \quad = \quad \lambda\sigma.\,k\,(\textit{LOOKUP}\,\rho\,\text{“}x\text{”})\,\sigma \quad = \quad k\,(\textit{LOOKUP}\,\rho\,\text{“}x\text{”}).$$

Note that in the η-reduced version, we have the same translation that we had when we translated uML. In general, any expression in uML! that is not state-aware can be η-reduced to the same translation as uML. Thus in order to translate to uML!, we need to add translation rules only for the functionality that is state-aware.

We assume that we have a type tag for locations and functions *LOC* and *CHECK-LOC* for tagging values as locations and checking those tags. We also assume that we have extended our *LOOKUP* and *UPDATE* functions to apply to stores.

$$\begin{aligned}
\mathcal{E}[\![\mathbf{ref}\,e]\!]\,\rho\,k\,\sigma &= \mathcal{E}[\![e]\!]\,\rho\,(\lambda v\sigma'.\,\mathbf{let}\ \ell = (\textit{MALLOC}\,\sigma')\ \mathbf{in}\ \mathbf{let}\ \sigma'' = \textit{UPDATE}\,\sigma'\ l\ v\ \mathbf{in}\ k\,(\textit{LOC}\,\ell)\ \sigma'')\ \sigma\\
\mathcal{E}[\![!e]\!]\,\rho\,k &= \mathcal{E}[\![e]\!]\,\rho\,(\textit{CHECK-LOC}\,(\lambda\ell\sigma'.\,k\,(\textit{LOOKUP}\,\sigma'\ \ell)\ \sigma'))\\
\mathcal{E}[\![e_1 := e_2]\!]\,\rho\,k &= \mathcal{E}[\![e_1]\!]\,\rho\,(\textit{CHECK-LOC}\,(\lambda\ell.\,\mathcal{E}[\![e_2]\!]\,\rho\,(\lambda v\sigma'.\,k\,(\textit{NULL}\,\mathbf{null})\ (\textit{UPDATE}\,\sigma'\ \ell\ v))))
\end{aligned}$$

3 Exceptions

An exception mechanism allows non-local transfer of control in exceptional situations. It is typically used to handle abnormal, unexpected, or rarely occurring events. It can simplify code by allowing programmers to factor out these uncommon cases.

To add an exception handling mechanism to uML, we first extend the syntax:

$$e \quad ::= \quad \ldots \ \mid\ \mathbf{raise}\ s\ e\ \mid\ \mathbf{try}\ e_1\ \mathbf{catch}\ (s\ x)\ e_2$$

Informally, the idea is that **catch** provides a handler e_2 to be invoked when the exception named s is encountered inside the expression e_1. To raise an exception, the program calls **raise** s e, where s is the name of an exception and e is an expression that will be passed to the handler as its argument x.

Most languages use a dynamic scoping mechanism to find the handler for a given exception. When an exception is encountered, the language walks up the runtime call stack until a suitable exception handler is found.

3.1 Exceptions in uML

To add exception support to our CPS translation, we add a *handler environment* h, which maps exception names to continuations. We also extend our *LOOKUP* and *UPDATE* functions to accommodate handler environments. Applied to a handler environment, *LOOKUP* returns the continuation bound to a given exception name, and *UPDATE* rebinds an exception name to a new continuation.

Now we can add exception support to our translation:

$$\begin{aligned}
[\![\mathbf{raise}\, s\, e]\!]\,\rho\, k\, h &= [\![e]\!]\,\rho\,(\mathit{LOOKUP}\, h\, \text{“}s\text{”})\, h \\
[\![\mathbf{try}\, e_1\, \mathbf{catch}\,(s\, x)\, e_2]\!]\,\rho\, k\, h &= [\![e_1]\!]\,\rho\, k\,(\mathit{EXTEND}\, h\, \text{“}s\text{”}\ (\lambda v.\, [\![e_2]\!](\mathit{EXTEND}\,\rho\, \text{“}x\text{”}\ v)\, k\, h)) \\
[\![\lambda x.\, e]\!]\,\rho\, k\, h &= k\ (\mathit{FUN}\,(\lambda y k' h'.\, \mathcal{E}[\![e]\!]\ (\mathit{EXTEND}\,\rho\, \text{“}x\text{”}\ y)\, k'\, h')) \\
&= k\ (\mathit{FUN}\,(\lambda y.\, [\![e]\!]\ (\mathit{EXTEND}\,\rho\, \text{“}x\text{”}\ y))) \\
[\![e_0\, e_1]\!]\,\rho\, k\, h &= [\![e_0]\!]\,\rho\,(\mathit{CHECK\text{-}FUN}\,(\lambda f.\, [\![e_1]\!]\,\rho\,(\lambda v.\, f v k h)))
\end{aligned}$$

There are some subtle design decisions captured by this translation. For example, if e_2 raises exception s in $\mathbf{try}\ e_1\ \mathbf{catch}\ (s\, x)\ e_2$, in this translation e_2 will not be invoked again. That is, e_2 cannot be invoked recursively.

If we study this translation, in particular the rules for λ and application, we see that it is similar to the translation for dynamically scoped variables. In fact, the exception mechanism is very similar to having a set of dynamically scoped variables that contain continuations for the various possible exception handlers, with **raise** as the application operation for these variables. Exceptions therefore inherit both the power and the problems with dynamic scoping. In particular, it is possible to accidentally catch an exception, just as it is possible to accidentally override a dynamic variable, with hard-to-predict results.

3.2 Exceptions with resumption

The exception mechanism above has the property that raising an exception terminates execution of the evaluation context. Most modern programming languages have exceptions with this *termination semantics*. A different approach to exceptions is to allow execution to continue at the point where the exception was raised, after the exception handler gets a chance to repair the damage. This approach is known as exceptions with *resumption semantics*. In practice it seems to be difficult to use these mechanisms usefully. The Cedar/Mesa system supported both kinds of exceptions and found that resumption-style exceptions were almost never used, and often resulted in bugs when they were.

Operating system interrupts are one place where resumption semantics can be seen. When a process receives an interrupt, the interrupt handler is run, and then execution continues at the point in the program where the interrupt happened.

We can give a translation that captures the semantics of resumption-style exceptions. We add two constructs to uML:

$$e ::= \mathbf{interrupt}\ s\ e \ \mid\ \mathbf{try}\ e_1\ \mathbf{handle}\ (s\ x)\ e_2$$

The translation makes the exception-handling environment h a mapping from exception names to *functions* rather than to continuations:

$$\begin{aligned}
[\![\mathbf{interrupt}\ s\ e]\!]\rho k h &= [\![e]\!]\rho(\lambda v.\, h\ s\ v\ k) \\
[\![\mathbf{try}\ e_1\ \mathbf{handle}\ (s\ x)\ e_2]\!] &= [\![e_1]\!]\rho k(\mathit{EXTEND}\, h\, \text{“}s\text{”}\ (\lambda v k'.\, [\![e_2]\!]\rho k' h))
\end{aligned}$$

This translation shows that with resumption semantics, the exception handler is really a dynamically bound function that is invoked at the point where the exception happens.