CS611 Lecture 16 — Control: Exceptions and More — October 5, 2001

Scribe: Bill McCloskey, Hubert Chao — Lecturer: Andrew Myers

Many different kinds of control transfers can be modeled in the uF language using CPS. In this lecture, various error recovery mechanisms from different languages are desugared into standard uF. The setjmp and longjmp functions from C, and the cwcc (call with current continuation) function from Scheme are examined, followed by a more lengthy analysis of Java exceptions.

1 Error Recovery in C

Two functions are available in C for changing the control flow of a program.

- int setjmp(jmp_buf a) places the register file in a and returns 0
- void longjmp(jmp_buf a, int val) restores the registers from a and returns val from the setjmp call

Since setjmp saves only the registers (and hence the stack pointer), but does not take a snapshot of memory, the programmer must assure that any call to longjmp is nested inside the setjmp caller. Otherwise, the stack frame of the setjmp caller will be obliterated, and longjmp will have undefined behavior. Here is an example of the use of these two functions in a C program.

```
jmp_buf env;

void function1() {
    int result;
    if ((result = setjmp(env)) == 0) compute();
    else report_error(result);
}

void compute() {
    ... call quadratic(a, b, c) in some complex way ...
}

float quadratic(float a, float b, float c) {
    if (c == 0) longjmp(env, DIVIDE_BY_ZERO);
}
```

The uF language can be augmented with these two functions, and the resulting language can be translated to a CPS uF. Two new functions will be introduced.

- setjmp e stores the current continuation at location e and returns 0
- longjmp e v applies the continuation stored in location e, passing it v

These functions do not have the same nesting restrictions as their C counterparts. They can be desugared in a natural fashion using CPS.

$$\begin{aligned}\mathcal{D}[\![\textsf{setjmp}\ e]\!]\rho k &= \mathcal{D}[\![e]\!]\ \rho\ (\textit{check-loc}\ (\lambda l \lambda\sigma\ k\ \langle 3,0\rangle\ \textit{update-store}(\sigma\ l\ \langle 7,k\rangle))))\\ \mathcal{D}[\![\textsf{longjmp}\ e_1\ e_2]\!]\rho k &= \mathcal{D}[\![e_1]\!]\ \rho\ (\textit{check-loc}\ (\lambda l\ \mathcal{D}[\![e_2]\!]\ \rho\ (\lambda v \lambda\sigma\ (\textit{check-cont}(\lambda k'\ (k'\ v)))\ (\textit{lookup}\ \sigma\ l)\ \sigma)))\end{aligned}$$

Recall here that 7 is the tag given to continuations and 3 is the tag given to integers. It is assumed that the return value of setjmp is an integer.

2 Other Languages

Other languages have control mechanisms that use continuations. ML has callcc which calls a function, passing it the current continuation. To jump to a given continuation, throw is used.

Scheme has cwcc, or "call with current continuation." A function is called. It is passed the current continuation, which appears as a normal function to the Scheme programmer. As an example,

```
let result = cwcc (lambda k ... if (bad) then (k err-code) ...)
in if (result = err-code) ...
```

cwcc can be desugared in uF as well.

$$\mathcal{D}[\![\mathsf{cwcc}\ e]\!]\rho k = \mathcal{D}[\![e]\!]\ \rho\ (\mathit{check\text{-}func}\ (\lambda f\ (f\ \langle 5, \lambda v \lambda k'\ (k\ v)\rangle\ k)))$$

3 Java Exceptions

Java exceptions are perhaps more familiar. A value, with a certain type, represents an exception. At some point in the program, an exception can be thrown when an error occurs. If the throw statement is enclosed by a catch clause, either lexically or in the current call tree, and the catch clause is labeled with the type of the exception, then control is passed to the catch clause.

Observe that the scoping rules for exceptions are identical to those for dynamic scope. Dynamic scoping had the disadvantage that it was difficult to determine what variables were in scope at a certain point in the program. Exceptions have a similar problem. It's difficult to know what sort of exceptions a certain library or function will throw or catch. For this reason, Java requires that all functions declare the exceptions that they may throw.

The uF language can be augmented with Java-style exceptions, but without the notion of types. Two new constructs will be added to the language.

- throw s e throws the exception s (represented by a string) and attaches the value e
- try e_1 catch $(s\ x)$ e_2 evaluates e_1, and then evaluates e_2 if the exception s is thrown

Since exceptions resemble dynamic scoping in so many ways, the same approach will be used to desugar them. A handling environment h will map exception names to continuations. The expression $\mathcal{D}[\![e]\!]\rho k h$ should either send the result of e to the continuation k or else invoke an exception handler in h. The desugaring rules are:

$$\begin{aligned}
\mathcal{D}[\![\mathsf{throw}\ s\ e]\!]\rho k h &= \mathcal{D}[\![e]\!]\ \rho\ (\lambda v\ (\mathit{lookup\text{-}handler}\ h\ s)\ v)\ h \\
\mathcal{D}[\![\mathsf{try}\ e_1\ \mathsf{catch}\ (s\ x)\ e_2]\!]\rho k h &= \mathcal{D}[\![e_1]\!]\ \rho\ k\ (\mathit{extend\text{-}handler}\ h\ s\ (\lambda v\ \mathcal{D}[\![e_2]\!]\ (\mathit{extend}\ \rho\ x\ v)\ k\ h))
\end{aligned}$$

Observe that according to the second definition, any exception thrown in a catch clause will be caught by some outer exception handler, not by the enclosing catch clause. Ordinary translations of abstraction and application must also be modified. Function values are translated roughly as follows: $\lambda x\ e \mapsto \lambda x \lambda k \lambda h\ e'$.

$$\begin{aligned}
\mathcal{D}[\![\lambda x\ e]\!]\rho k h &= k\ (\lambda x' \lambda k' \lambda h'\ \mathcal{D}[\![e]\!]\ (\mathit{extend}\ \rho\ x\ x')\ k'\ h') \\
\mathcal{D}[\![e_0\ e_1]\!]\rho k h &= \mathcal{D}[\![e_0]\!]\ \rho\ (\mathit{check\text{-}func}\ (\lambda f\ \mathcal{D}[\![e_1]\!]\ \rho\ (\lambda v\ f\ v\ k\ h)\ h))\ h
\end{aligned}$$

From these definitions, we can see that a function's handling context is inherited from where it's called, not from where it's declared.