CS 411 Lecture 23 Exceptions 12 Nov 2003
Lecturer: Greg Morrisett

Exceptions

Imagine if ML or Java did not have exceptions. What would we do to signal an error? For instance, what would we do on a divide by zero?

One idea is to use options everywhere. The option datatype is defined as:

```
datatype 'a option = Some of 'a | None
```

Then we could define division as follows:

```
fun div (x:int, y:int):int option =
  if (y = 0) then None
  else Some(System.real_div(x / y))
```

where System.real_div was the actual division code. Anyone who called the div function would have to do a pattern match to get out the value. For instance, in normal ML with exceptions, we can just write:

```
fun foo(x,y,z,w) = (x div y) * (z div w)
```

but in the mythical language where exceptions are treated as options, we would have to write:

```
fun foo(x,y,z,w) =
    case (x div y) of
      None => None
    | Some(a) => (case (z div w) of
                   None => None
                 | Some(b) => a * b)
```

Note that in original ML, foo takes four ints and returns an int, whereas in the exceptionless ML, foo takes four ints and returns an int option. The idea is that in general, any function that calls a function that might throw an exception gets polluted — it now has to potentially return an option.

To understand this idea better, suppose we have this little source language:

$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid \mathsf{unit}$
$e ::= i \mid x \mid \lambda x : t.\, e \mid e_1\ e_2 \mid e_1\ \mathsf{op}\ e_2 \mid () \mid \mathsf{throw} \mid \mathsf{try}\ e_1\ \mathsf{catch}\ e_2$

Here, throw raises an exception, and try/catch executes e_1, if it raises an exception, then executes e_2, otherwise it returns the value that e_1 produced. We could compile this source language to a target language without exceptions, but with sums

$\tau' ::= \mathsf{int} \mid \tau_1' \to \tau_2' \mid \mathsf{unit} \mid \tau_1' + \tau_2'$
$e' ::= i \mid x \mid \lambda x : t.\, e' \mid e_1'\ e_2' \mid e_1'\ \mathsf{op}\ e_2' \mid () \mid \mathsf{inl}(e') \mid \mathsf{inr}(e') \mid (\mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}(x_1) \Rightarrow e_1' \mid \mathsf{inr}(x_2) \Rightarrow e_2')$

To begin, we first write down a way to translate types. In general, a source-level expression of type τ might throw an exception, so we'll translate the expression into a τ' option. A τ' option is a sum $\tau' + 1$.

$$\begin{array}{rcl} \mathcal{C}[\![\tau]\!] & = & \mathcal{V}[\![\tau]\!] + 1 \qquad \text{// type translation for computations} \\ \\ \mathcal{V}[\![\mathsf{int}]\!] & = & \mathsf{int} \qquad \text{// type translation for values} \\ \mathcal{V}[\![\tau_1 * \tau_2]\!] & = & \mathcal{V}[\![\tau_1]\!] * \mathcal{V}[\![\tau_2]\!] \\ \mathcal{V}[\![\tau_1 + \tau_2]\!] & = & \mathcal{V}[\![\tau_1]\!] + \mathcal{V}[\![\tau_2]\!] \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \mathcal{V}[\![\tau_1]\!] \to \mathcal{C}[\![\tau_2]\!] \end{array}$$

The key thing to note is that function types are special. In general, a function will take a τ *value* and return a τ *computation.* A τ computation is either a τ value or a unit, where the unit represents an uncaught exception.

With this type translation in mind, we can now compile source level programs so as to eliminate the throw and try/catch:

$$\begin{array}{rcl} \mathcal{E}[\![i]\!] & = & \mathsf{inl}(i) \\ \mathcal{E}[\![x]\!] & = & \mathsf{inl}(x) \\ \mathcal{E}[\![()]\!] & = & \mathsf{inl}() \\ \\ \mathcal{E}[\![\mathsf{throw}]\!] & = & \mathsf{inr}() \\ \mathcal{E}[\![\mathsf{try}\ e_1\ \mathsf{catch}\ e_2]\!] & = & \mathsf{case}\ \mathcal{E}[\![e_1]\!]\ of \\ & & \quad \mathsf{inl}(x_1) => \mathsf{inl}(x_1) \\ & & \ \mid \mathsf{inr}(x_2) => \mathcal{E}[\![e_2]\!] \\ \\ \mathcal{E}[\![\lambda x : t.\, e]\!] & = & \lambda x : \mathcal{V}[\![\tau]\!].\, \mathcal{E}[\![e]\!] \\ \mathcal{E}[\![e_1\ e_2]\!] & = & \mathsf{case}\ \mathcal{E}[\![e_1]\!]\ of \\ & & \quad \mathsf{inl}(x_1) => (\mathsf{case}\ \mathcal{E}[\![e_2]\!]\ of \\ & & \qquad\qquad\qquad \mathsf{inl}(y_1) => x_1\ y_1 \\ & & \qquad\qquad\quad \mid \mathsf{inr}(y_2) => \mathsf{inr}()) \\ & & \ \mid \mathsf{inr}(x_2) => \mathsf{inr}() \\ \mathcal{E}[\![e_1\ \mathsf{op}\ e_2]\!] & = & \mathsf{case}\ \mathcal{E}[\![e_1]\!]\ of \\ & & \quad \mathsf{inl}(x_1) => (\mathsf{case}\ \mathcal{E}[\![e_2]\!]\ of \\ & & \qquad\qquad\qquad \mathsf{inl}(y_1) => \mathsf{inl}(x_1\ \mathsf{op}\ y_1) \\ & & \qquad\qquad\quad \mid \mathsf{inr}(y_2) => \mathsf{inr}()) \\ & & \ \mid \mathsf{inr}(x_2) => \mathsf{inr}() \end{array}$$

You can verify for yourself that if $\Gamma \vdash e : \tau$ in the source level, then $\mathcal{V} \circ \Gamma \vdash \mathcal{E}[\![e]\!] : \mathcal{C}[\![\tau]\!]$ at the target level. That is, a well-typed source expression maps to a well-typed target expression.

The bad thing about treating exceptions as options is that it's expensive. In the normal case, where there is no exception, we're having to construct an option (Some(result)) and pattern match on results of function calls. So, most languages do not implement exceptions in this fashion. Rather, they make the common case (no exception) fast and the uncommon case (exceptions) a bit slower.

This is usually achieved by having try/catch record something on the control-stack, and then by having throw unwind the stack until it runs into a try/catch frame. In this fashion, there is only some small overhead for try/catch and a bit more overhead for throw. (In practice, throw can be made cheap too by dedicating a register to point to the last try/catch frame.)

Here is a way to model this formally. We begin by introducing stacks and stack frames for our little exception language above:

$$\begin{aligned} S &::= \mathsf{nil} \mid F :: S \\ F &::= [\cdot]\ e \mid v\ [\cdot] \mid [\cdot]\ \mathsf{op}\ e_2 \mid v\ \mathsf{op}\ [\cdot] \mid \mathsf{try}\ [\cdot]\ \mathsf{catch}\ e_2 \end{aligned}$$

A stack is a list of frames (growing from the right to the left). A frame is an expression with a hole (denoted $[\cdot]$).

Our configurations are of the form (S, e) where S is a stack and e is the expression we're currently evaluating. If the stack is empty (nil) and e is a value, then this is a (good) terminal configuration — the value is the result of running the program.

Intuitively, if e is a compound expression, we extract the sub-expression to evaluate first (i.e., e_1 in either $e_1\ e_2$ or e_1 op e_2) and push a frame on the stack that records what to do when we're done evaluating that sub-expression.

Here are the rewriting rules for the exception-free fragment:

1. $(S, (\lambda x : \tau.\, e)\ v) \to (S, e[v/x])$
2. $(S, v_1\ \mathsf{op}\ v_2) \to (S, v) \qquad (v = v_1\ \mathsf{op}\ v_2)$
3. $(S, v\ e) \to ((v\ [\cdot]) :: S, e)$
4. $(S, e_1\ e_2) \to (([\cdot]\ e_2) :: S, e_1)$
5. $(S, v\ \mathsf{op}\ e) \to ((v\ \mathsf{op}\ [\cdot]) :: S, e)$
6. $(S, e_1\ \mathsf{op}\ e_2) \to (([\cdot]\ \mathsf{op}\ e_2) :: S, e_1)$
7. $(F :: S, v) \to (S, F[v])$

The first two rules do primitive reductions (i.e., function call or arithmetic operation) and don't affect the stack. Rules 3-6 are breaking apart a compound expression to evaluate the first non-value expression. They do so by pushing on a frame that records what to do when the sub-expression is evaluated. For instance, if we have:

$$(S, (3 + 4) + (5 * 6))$$

then we first want to evaluate $(3 + 4)$ and remember that we should add the result to $(5 * 6)$. So, we step to:

$$(([\cdot] + (5 * 6)) :: S, 3 + 4)$$

via rule 6. Now we can step using rule 2 to get:

$$(([\cdot] + (5 * 6)) :: S, 7)$$

At this point, we're done evaluating the first sub-expression (we have a value) so we pop off the top stack frame and plug the value in for the hole:

$$(S, 7 + (5 * 6))$$

This is a compound expression where the first non-value sub- expression is the $5 * 6$. We can evaluate this, but we have to remember to add it to 7 when we're done. So, we use rule 5 to step to:

$$((7 + [\cdot]) :: S, 5 * 6)$$

This reduces via rule 2 to:

$$((7+[\cdot]) :: S, 30)$$

and again we use rule 7 to pop the frame and get:

$$(S, 7+30)$$

Then we use rule 2 again to get:

$$(S, 37)$$

If S is empty, then we're done with the computation. If S is non-empty, then we'd return 37 to the top frame of S using rule 7.

Now, what should we do about the throw and try/catch expressions? The basic idea is that when we get to the point where we are evaluating a throw:

$$(S, \mathsf{throw})$$

then we want to return control to the nearest enclosing try/catch and execute its handler. To achieve this, when we do a try/catch, we need to record a frame. So, the rules are:

8. $(S, \mathsf{try}\ v\ \mathsf{catch}\ e_2) \to (S, v)$
9. $(S, \mathsf{try}\ e_1\ \mathsf{catch}\ e_2) \to ((\mathsf{try}\ [\cdot]\ \mathsf{catch}\ e_2) :: S, e1)$
10. $(S_1 @ (\mathsf{try}\ [\cdot]\ \mathsf{catch}\ e_2) :: S_2, \mathsf{throw}) \to (S_2, e_2)$ (no try in S_1)

Rule 8 says that if we already have a value, then we can throw away the try/catch. (The value can't throw an exception.) If we have a non-value within a try/catch, then we push a try/catch frame on the stack and evaluate the body of the try/catch using rule 9.

When we encounter a throw, we look down the stack until we find the first "try $[\cdot]$ catch e_2" frame. We pop all of the other frames off the stack and continue by executing the handler for the try/catch.

For example:

$$\begin{array}{ll}
 & (\mathsf{nil}, (\mathsf{try}\ (3+\mathsf{throw}) * (5+6)\ \mathsf{catch}\ 42) + 1) \\
\to & (([\cdot]+1) :: \mathsf{nil}, (\mathsf{try}\ (3+\mathsf{throw}) * (5+6)\ \mathsf{catch}\ 42)) \\
\to & ((\mathsf{try}\ [\cdot]\ \mathsf{catch}\ 42) :: ([\cdot]+1) :: \mathsf{nil}, (3+\mathsf{throw}) * (5+6)) \\
\to & (([\cdot] * (5+6)) :: (\mathsf{try}\ [\cdot]\ \mathsf{catch}\ 42) :: ([\cdot]+1) :: \mathsf{nil}, 3+\mathsf{throw}) \\
\to & ((3+[\cdot]) :: ([\cdot][] * (5+6)) :: (\mathsf{try}\ [\cdot]\ \mathsf{catch}\ 42) :: ([\cdot]+1) :: \mathsf{nil}, \mathsf{throw}) \\
\to & (([\cdot]+1) :: \mathsf{nil}, 42) \\
\to & (\mathsf{nil}, 42+1) \\
\to & (\mathsf{nil}, 43)
\end{array}$$

What happens if there's no try/catch frame on the stack when we encounter a throw? This is an uncaught exception. We could just make this a "good" terminal state. If so, then our type-soundness theorem would say something like:

> "If an expression e is well-typed, then $(\mathsf{nil}, e) \to^* (\mathsf{nil}, v)$ or else $(\mathsf{nil}, e) \to^* (S, \mathsf{throw})$ where S has no try/catch or e runs forever."

In English, e either runs forever, evaluates to a value or has an uncaught exception, but does not otherwise get stuck due to a type-error. This is the guarantee that languages such as ML and Java give us.