Exceptions

CPSC 509: Programming Language Principles

Ronald Garcia*

13 November 2011

Introduction

Abstract machine semantics give you a mechanical process for decomposing a program into an evaluation context and redex. This process has been left implicit in all of our reduction semantics before. In addition, abstract machine semantics have a much closer connection to a real implementation of a programming language. The expressions that make up a program act as an instruction set for the abstract machine, and the evaluation context (in inside-out form) acts as its runtime stack. Many actual language implementations have been developed by expanding upon such abstract machines.

In this section, we use the intuition provided by abstract machines to introduce a common language feature that manipulates program control: exceptions. It turns out that exceptions can also be presented using reduction semantics, but seeing how control really flows through a machine can give a better feel for what may appear too abstract in reduction semantics.

A Language with Exceptions

We extend the language of Boolean and arithmetic expressions with two mechanisms: one for raising exceptions and one for catching them. The syntax of the language follows.

Syntax

$$
\begin{array}{rcl}
x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad t \in \textsc{Term}, \quad v \in \textsc{Val}, \quad r \in \textsc{Redex}, \quad E \in \textsc{ECtxt} \\
t & ::= & x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid n \mid t + t \mid \mathsf{raise}\ t \mid \mathsf{try}\ t\ \mathsf{handle}\ x.t \\
v & ::= & n \mid \mathsf{true} \mid \mathsf{false} \\
r & ::= & v + v \mid \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{try}\ v\ \mathsf{handle}\ x.t \\
E & ::= & \Box \mid E[\Box + t] \mid E[v + \Box] \mid E[\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t] \mid E[\mathsf{raise}\ \Box] \mid E[\mathsf{try}\ \Box\ \mathsf{handle}\ x.t]
\end{array}
$$

The $\mathsf{raise}\ t$ expression evaluates t to a value and then raise it as an exception. The first part of that behavior is captured by the evaluation context form $E[\mathsf{raise}\ \Box]$. The counterpart expression to raise is $\mathsf{try}\ t_1\ \mathsf{handle}\ x.t_2$. This expression first evaluates t_1, as suggested by the evaluation context form $E[\mathsf{try}\ \Box\ \mathsf{handle}\ x.t_2]$. If t_1 evaluates to a normal value v_1, then the result of the expression is v_1. However, if t_1 raises an exception $\mathsf{raise}\ v$, then the handler t_2 is run, with x bound to the value v. The language of terms t is also extended with variables x so that exception handlers can refer to the raised value.

Extending the Machine

To extend the machine with support for these expressions, we first add the basic rules to refocus that point out where evaluation proceeds in the body of raise and try.

$$\begin{array}{rcl} \langle E, \mathsf{raise}\ t\rangle_{\text{refocus}} & \longrightarrow & \langle E[\mathsf{raise}\ \square], t\rangle_{\text{refocus}} \\ \langle E, \mathsf{try}\ t_1\ \mathsf{handle}\ x.t_2\rangle_{\text{refocus}} & \longrightarrow & \langle E[\mathsf{try}\ \square\ \mathsf{handle}\ x.t_2], t_1\rangle_{\text{refocus}} \end{array}$$

$$\begin{array}{rcl} \langle E[\square + t], v\rangle_{\text{return}} & \longrightarrow & \langle E[v + \square], t\rangle_{\text{refocus}} \\ \langle E[v_1 + \square], v_2\rangle_{\text{return}} & \longrightarrow & \langle E, v_1 + v_2\rangle_{\text{reduce}} \\ \langle E[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t], v\rangle_{\text{return}} & \longrightarrow & \langle E, \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t\rangle_{\text{reduce}} \end{array}$$

We also need to extend the return rules to consider the two new contexts. First, try is discarded if a value is returned to it; we'll do that during reduction.

$$\begin{array}{rcl} \langle E[\mathsf{try}\ \square\ \mathsf{handle}\ x.t_2], v_1\rangle_{\text{return}} & \longrightarrow & \langle E, \mathsf{try}\ v_1\ \mathsf{handle}\ x.t_2\rangle_{\text{reduce}} \\ \langle E, \mathsf{try}\ v_1\ \mathsf{handle}\ x.t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, v_1\rangle_{\text{refocus}} \end{array}$$

If a value is returned to raise , though, then we must prepare to raise an exception. To do so, we introduce a new configuration to the machine to raising an exception.

$$\begin{array}{rcl} C \in \textsc{Cfg} & & \\ C & ::= & \langle E, t\rangle_{\text{refocus}} \mid \langle E, r\rangle_{\text{reduce}} \mid \langle E, v\rangle_{\text{return}} \mid \langle E, \mathsf{raise}\ v\rangle_{\text{raise}} \end{array}$$

Then when a value is returned to raise, an exception is raised:

$$\langle E[\mathsf{raise}\ \square], v\rangle_{\text{return}} \longrightarrow \langle E, \mathsf{raise}\ v\rangle_{\text{raise}}$$

The machine rules for raise proceed to pop frames off of the evaluation context as the exception propagates upwards:

$$\begin{array}{rcl} \langle E[\square + t], \mathsf{raise}\ v\rangle_{\text{raise}} & \longrightarrow & \langle E, \mathsf{raise}\ v\rangle_{\text{raise}} \\ \langle E[v_1 + \square], \mathsf{raise}\ v\rangle_{\text{raise}} & \longrightarrow & \langle E, \mathsf{raise}\ v\rangle_{\text{raise}} \\ \langle E[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t], \mathsf{raise}\ v\rangle_{\text{raise}} & \longrightarrow & \langle E, \mathsf{raise}\ v\rangle_{\text{raise}} \\ \langle E[\mathsf{raise}\ \square], \mathsf{raise}\ v\rangle_{\text{raise}} & \longrightarrow & \langle E, \mathsf{raise}\ v\rangle_{\text{raise}} \end{array}$$

In principle, these reductions could have been worked into the reduce rules, but these separate configurations are more closely related to what happens in a real implementation. However, the try contexts catches any raised exception that reaches it and redirects execution to its associated handler.

$$\langle E[\mathsf{try}\ \square\ \mathsf{handle}\ x.t], \mathsf{raise}\ v\rangle_{\text{raise}} \longrightarrow \langle E, [v/x]t\rangle_{\text{refocus}}$$

Here we use substitution to bind the value v to the variable x in the body of the handler, and refocus continues computation in the handler.

To get a better sense of how exceptions work, consider the trace of a program that uses exceptions:

$$\begin{array}{c}
\langle \Box, 5 + \text{try (if (7 + raise 4) then true else false) handle x.x + 2} \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[\Box + \text{try (if (7 + raise 4) then true else false) handle x.x + 2}], 5 \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[\Box + \text{try (if (7 + raise 4) then true else false) handle x.x + 2}], 5 \rangle_{\text{return}} \\
\longrightarrow \langle \Box[5+\Box], \text{try (if (7 + raise 4) then true else false) handle x.x + 2} \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}], (\text{if (7 + raise 4) then true else false}) \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}], 7 + \text{raise 4} \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}][\Box + \text{raise 4}], 7 \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}][\Box + \text{raise 4}], 7 \rangle_{\text{return}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}][7+\Box], \text{raise 4} \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}][7+\Box][\text{raise } \Box], 4 \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}][7+\Box][\text{raise } \Box], 4 \rangle_{\text{return}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}][7+\Box], \text{raise 4} \rangle_{\text{raise}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}][\text{if } \Box \text{ then true else false}], \text{raise 4} \rangle_{\text{raise}} \\
\longrightarrow \langle \Box[5+\Box][\text{try } \Box \text{ handle x.x + 2}], \text{raise 4} \rangle_{\text{raise}} \\
\longrightarrow \langle \Box[5+\Box], 4 + 2 \rangle_{\text{refocus}} \longrightarrow \langle \Box[5+\Box][\Box+2], 4 \rangle_{\text{refocus}} \longrightarrow \langle \Box[5+\Box][\Box+2], 4 \rangle_{\text{return}} \\
\longrightarrow \langle \Box[5+\Box][4+\Box], 2 \rangle_{\text{refocus}} \longrightarrow \langle \Box[5+\Box], 4 + 2 \rangle_{\text{reduce}} \longrightarrow \langle \Box[5+\Box], 6 \rangle_{\text{refocus}} \\
\longrightarrow \langle \Box[5+\Box], 6 \rangle_{\text{return}} \longrightarrow \langle \Box, 5 + 6 \rangle_{\text{reduce}} \longrightarrow \langle \Box, 11 \rangle_{\text{refocus}} \longrightarrow \langle \Box, 11 \rangle_{\text{return}} .
\end{array}$$

Phew!

Reduction Semantics for Exceptions

Now that we've seen how exceptions are represented in an abstract machine, we can consider how they might be represented using reduction semantics. The nice thing about the abstract machine semantics is that the behavior of the machine looks a lot like how a program would really run on a real language implementation. In fact, many language implementations (compilers and interpreters) are little more than an implementation of an abstract machine on a real machine. Here you can really see how raising an exception discards chunks of the stack until it is stopped by an exception handler.

The downside of abstract machine semantics is that they are big and noisy. All of the details about finding a redex, returning results, and transitioning between machine modes help us see how a concrete implementation works, but they add a lot of extra details which can obscure the essence of a language design.

Reduction semantics, on the other hand, are quite economical. They present the core of a language semantics, thereby emphasizing the interesting aspects. As it turns out, we can give a semantics for exceptions in reduction semantics, without resorting to abstract machines. We already have most of the necessary pieces to present the reduction semantics: syntax and evaluation contexts. Also, we have all of the notions of reduction that were introduced in the last lecture, and were ultimately became the `reduce` rules of the abstract machine. What are needed now are notions of reduction that address raising and catching exceptions. Those notions of reduction follow:

$$\begin{array}{rcl}
\text{raise } v + t & \rightsquigarrow & \text{raise } v \\
v_1 + \text{raise } v & \rightsquigarrow & \text{raise } v \\
\text{if raise } v \text{ then } t \text{ else } t & \rightsquigarrow & \text{raise } v \\
\text{raise (raise } v) & \rightsquigarrow & \text{raise } v \\
\text{try raise } v \text{ handle } x.t & \rightsquigarrow & [v/x]t \\
\text{try } v \text{ handle } x.t & \rightsquigarrow & v
\end{array}$$

Do these rules look familiar? Each of them corresponds to a `raise` rules of the abstract machine. The first 4 are quite boring: throw away stack frames that are not try-handle blocks. The second-to-last one handles

the exception that was raised. The last one propagates a value through a handler. Having extended the notions of reduction as above, we could elaborate this into a slightly different abstract machine semantics. All of the above rules would become `reduce` rules, rather than `raise` rules, and `raise` configurations would transition to `reduce` configurations

There are yet more succinct ways to express these exceptions as reduction semantics. We can represent the process of raising an exception and handling it as one big reduction step. To do this, we first define the subset of evaluation contexts that do not handle exceptions:

$$\begin{array}{l} F \in \mathsf{RCtxt} \subseteq \mathsf{ECtxt} \\ F \quad ::= \quad \Box \mid F[\Box + t] \mid F[v + \Box] \mid F[\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t] \mid F[\mathsf{raise}\ \Box] \end{array}$$

Then given this definition, raising and handling an exception can be captured in a single notion of reduction:

$$\mathsf{try}\ F[\mathsf{raise}\ v]\ \mathsf{handle}\ x.t \quad \longrightarrow \quad [v/x]t$$

Here, $F[\mathsf{raise}\ v]$ represents some expression that can be broken down into a restricted evaluation context (containing no try-handle frames) and an expression that is raising a value. Now in one fail swoop, one step of the reduction semantics, the exception is raised and handled.

Exceptions with Resumptions

In every mainstream language with exceptions, the process of raising an exception throws away part of the control context (i.e. stack, i.e. evaluation context) until arriving at a handler for the exception. Some language design researchers have wondered whether this behavior was strictly necessary, or whether it would make sense to handle an exception and go right back to whatever was going on when the exception was raised. This model of exceptions is called *exceptions with resumptions*, and we can easily model this variation[1].

The syntax of this kind of language is exactly the same as before. What changes is the *behavior*. We can capture this change of behavior simply by altering what the abstract machine does when an exception is raised. In order to resume the computation that was happening when the exception was raised, the machine must remember all the information that the other machine simply discarded. To this end, we change the `raise` configuration to keep track of the part of the context that used to be discarded

$$C ::= \ldots \mid \langle E, \mathsf{raise}\ v, E \rangle_{\text{raise}}$$

and the machine transitions that involve `raise` configurations are updated to save and restore context information.

$$\begin{array}{rcl} \langle E[\mathsf{raise}\ \Box], v \rangle_{\text{return}} & \longrightarrow & \langle E, \mathsf{raise}\ v, \Box \rangle_{\text{raise}} \\ \langle E_1[\Box + t], \mathsf{raise}\ v, E_2 \rangle_{\text{raise}} & \longrightarrow & \langle E_1, \mathsf{raise}\ v, \Box[\Box + t] {+}{+} E_2 \rangle_{\text{raise}} \\ \langle E_1[v_1 + \Box], \mathsf{raise}\ v, E_2 \rangle_{\text{raise}} & \longrightarrow & \langle E_1, \mathsf{raise}\ v, \Box[v_1 + \Box] {+}{+} E_2 \rangle_{\text{raise}} \\ \langle E_1[\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t], \mathsf{raise}\ v, E_2 \rangle_{\text{raise}} & \longrightarrow & \langle E_1, \mathsf{raise}\ v, \Box[\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t] {+}{+} E_2 \rangle_{\text{raise}} \\ \langle E_1[\mathsf{raise}\ \Box], \mathsf{raise}\ v, E_2 \rangle_{\text{raise}} & \longrightarrow & \langle E_1, \mathsf{raise}\ v, \Box[\mathsf{raise}\ \Box] {+}{+} E_2 \rangle_{\text{raise}} \\ \langle E_1[\mathsf{try}\ \Box\ \mathsf{handle}\ x.t], \mathsf{raise}\ v, E_2 \rangle_{\text{raise}} & \longrightarrow & \langle E_1[\mathsf{try}\ \Box\ \mathsf{handle}\ x.t] {+}{+} E_2, [v/x]t \rangle_{\text{refocus}} \end{array}$$

To define these transitions, we rely on the function $E_1 {+}{+} E_2$ (also called *append*), which sticks the E_2 context into E_1's hole (I leave it's definition as an exercise).

The `return` configuration that raises an exception must now make space to remember the part of the context between the exception and its handler. Then, each step of `raise` that does not handle the exception pops the current frame, but stores it in the second context. Finally when an exception handler is reached, the exception value is passed to the handler, using substitution, which is executed back in the original context from which the exception was raised.

As with the traditional exception semantics, we can recast exceptions with resumption as a reduction semantics where exception raising and handling happens in one reduction step:

$$\mathsf{try}\ F[\mathsf{raise}\ v]\ \mathsf{handle}\ x.t \longrightarrow \mathsf{try}\ F[v/x]t]\ \mathsf{handle}\ x.t.$$

[1] As it turns out, exceptions with resumptions have been deemed a **bad idea**!

Notice how this semantics differs from traditional exceptions The intermediate context, including the exception handler, is preserved. What happens in this step is that the raised exception value is substituted into a copy of the exception handler. This behavior is very similar to the process of looking up a function (i.e. $\lambda x.t$) and applying it to v.