Lecture Notes on Exceptions

15-814: Types and Programming Languages
Frank Pfenning

Lecture 13
Tuesday, October 13, 2020

1 Introduction

In the previous lecture we introduced general pattern matching, which naturally led to considering an exception if no branch matched. In this lecture we continue our investigation of exceptions. As always, we consider statics and dynamics and the important theorems showing that they cohere.

2 Preservation for Exceptions

Recall the typing of case-expressions and branches for general pattern matching from the last lecture:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (bs) : \sigma}\ \mathsf{case}$$

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid bs) : \sigma}\ \mathsf{tp/bs/alt} \qquad \frac{}{\Gamma \vdash \tau \triangleright (\cdot) : \sigma}\ \mathsf{tp/bs/none}$$

A key observation here is that when we reach the empty list of branches (rule tp/bs/none) the type σ can be anything—usually, it is determined from the other branches.

Now recall the dynamics of pattern matching from the last lecture.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (bs) \mapsto \mathsf{case}\ e_0'\ (bs)}\ \mathsf{step/case}_0$$

$$\frac{v\ \mathit{value} \quad v = [\eta]p}{\mathsf{case}\ v\ (p \Rightarrow e \mid bs) \mapsto [\eta]e}\ \mathsf{step/case/match}$$

$$\frac{v\ \mathit{value} \quad \text{there is no } \eta \text{ with } v = [\eta]p \quad \mathsf{case}\ v\ (bs) \mapsto e'}{\mathsf{case}\ v\ (p \Rightarrow e \mid B) \mapsto e'}\ \mathsf{step/case/nomatch}$$

$$\frac{v\ \mathit{value}}{\mathsf{case}\ v\ (\cdot) \mapsto \mathsf{raise\ Match}}\ \mathsf{step/case/none}$$

Here we imagine that we extended the syntax of expressions

$$\begin{array}{llll} \text{Expressions} & e & ::= & \ldots \mid \mathsf{case}\ e\ (bs) \mid \mathsf{raise}\ E \\ \text{Exceptions} & E & ::= & \mathsf{Match} \mid \ldots \end{array}$$

where there may be other (for now unspecified) exceptions such as DivByZero.

In order to obtain type preservation, we need raise E to have all possible types, because the expression on the left-hand side of the step/case/none rule (namely case $v\ (\cdot)$) can have any type.

$$\frac{}{\Gamma \vdash \mathsf{raise\ Match} : \tau}\ \mathsf{tp/raise}$$

Type preservation then obviously holds for the only rule so far that involves raising an exception. We just need to make sure that as we explore the dynamics of raising an exceptions preservation continues to hold.

3 Progress for Exceptions

We are aiming at the following version of the progress theorem.

Theorem 1 (Progress with Exceptions, v1) *If $\cdot \vdash e : \tau$ then*

(i) either $e \mapsto e'$ for some e',

(ii) or e val,

(iii) or $e = \mathsf{raise}\ E$ for an exception E.

Trying to prove this will uncover the fact that, currently, this theorem is false for our language. Consider, as a simple example, $\langle \mathsf{raise}\ \mathsf{Match}, \langle\rangle\rangle$. This has type $\tau \times 1$ for any τ, and yet it is stuck: it can not transition, it is not a value, and it is not of the form $\mathsf{raise}\ E$. To remedy this shortcoming, we need to add rules to the dynamics to propagate an exception to the top level. This is awkward, because we need to do it for every kind of expression we already have! This is a shortcoming of this particular style of defining the dynamics of our language, compounded by the fact that exceptions are a control construct, in some sense unrelated to our type structure.

We only show the rules related to pairs.

$$\frac{}{\langle \mathsf{raise}\ E, e\rangle \mapsto \mathsf{raise}\ E}\ \mathsf{step/pair/raise}_1 \qquad \frac{v\ \mathit{value}}{\langle v, \mathsf{raise}\ E\rangle \mapsto \mathsf{raise}\ E}\ \mathsf{step/pair/raise}_2$$

$$\frac{}{\mathsf{case}\ (\mathsf{raise}\ E)\ B \mapsto \mathsf{raise}\ E}\ \mathsf{step/case/raise}$$

It is insignificant here whether we have general pattern matching, or pattern matching specialized to pairs as in earlier versions of our language.

Now we can prove the progress theorem as usual.

Proof: (Progress with Exceptions, Theorem 1) By rule on induction on the derivation of $\cdot \vdash e : \tau$. In comparison with earlier proofs, when we apply the induction hypothesis we obtain three cases to distinguish. In case a subexpression raises an exception, the expression does as well (as long as it is not a value) because we have added enough rules to propagate exception to the top level. □

4 Catching Exceptions

Most languages allow programs not only to raise exceptions but also to catch them. Let's consider the simplest such construct, $\mathsf{try}\ e_1\ e_2$. The intention is for it to evaluate e_1 and return its value if that is successful. If it raises an exception, evaluate e_2 instead. This time, we begin with the dynamics.

$$\frac{e_1 \mapsto e_1'}{\mathsf{try}\ e_1\ e_2 \mapsto \mathsf{try}\ e_1'\ e_2}\ \mathsf{step/try}_0 \qquad \frac{v_1\ \mathit{value}}{\mathsf{try}\ v_1\ e_2 \mapsto v_1}\ \mathsf{step/try/value}$$

$$\frac{}{\mathsf{try}\ (\mathsf{raise}\ E)\ e_2 \mapsto e_2}\ \mathsf{step/try/raise}$$

What type do we need to assign to try e_1 e_2 in order to guarantee type preservation. We start with what we know:

$$\frac{\Gamma \vdash e_1 : \boxed{} \quad \Gamma \vdash e_2 : \boxed{}}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \boxed{}}\ \mathsf{tp/try}$$

We should be able to "try" an expression of arbitrary type τ, so

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \boxed{}}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \boxed{}}\ \mathsf{tp/try}$$

Because of the rule step/try/value, the type of the overall expression needs to be equal to τ as well.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \boxed{}}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \tau}\ \mathsf{tp/try}$$

Finally, in case e_1 fails we step to e_2, so we also must have $e_2 : \tau$.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \tau}\ \mathsf{tp/try}$$

One issue here is that in e_2 we cannot tell which exception may have been raised, even if we may want to take different actions for different exceptions. That is, we would like to be able to *match* against different exceptions. The generalizations do not introduce any new ideas, so we leave it to Exercise 1 to work out the details.

Exceptions in this lecture and Exercise 1 are not first class, which means that exceptions are not values. This in turn means that functions cannot take exceptions as arguments or return them. If we want exceptions to carry values (for example, error messages) then either exceptions and expression will be mutually recursive syntactic classes, or we lift exceptions and make them first class. The merits of this approach are debatable, but its formalization is not much more difficult than what we have already done (see [Har16, Chapter 29]).

Exercises

Exercise 1 We would like to generalize the try construct to so it can branch on the exception that may have been raised. So we have

Expressions	e	::=	$\ldots \mid \mathsf{raise}\ E \mid \mathsf{try}\ e\ (ms)$
Exceptions	E	::=	$\mathsf{Match} \mid \mathsf{DivByZero} \mid \ldots$
Exception Handlers	ms	::=	$\cdot \mid (E \Rightarrow e \mid ms)$

Note that exception handlers are not already covered by regular pattern matching, because exceptions are neither values nor patterns.

1. Write out typing rules for the generalized try construct and exception handlers.
2. Write out the dynamics for the new constructs. Exception handlers should be tried in order.

You do not have to prove preservation or progress, but you should make sure your rules posses these properties (when taken together with the language we have developed in the course so far).

Exercise 2 We would like to generalize exceptions further so they can be returned by functions, passed as arguments, and dynamically created. For this purpose we create a new type exn. We think of the type exn as a disjoint sum

$$\mathsf{exn} = (\mathbf{Match} : 1) + (\mathbf{DivByZero} : 1) + \cdots$$

except that we can add new alternatives to the sum with a declaration

$$\mathsf{exn} = \mathsf{exn} + (i : \tau)$$

For example, in the absence of strings in the language, we could number different error exceptions instead of relying on the general Match by including

$$\mathsf{exn} = \mathsf{exn} + (\mathbf{Error} : \mathit{nat})$$

and then let the programmer raise Error k to indicate error number k.

A key property to keep in mind that $e : \mathit{exn}$ is an ordinary expression (and its value can be passed around) and raise e requires e to be an exception that can be raised, changing the control flow. Generalized pattern matching should now work to match against exceptions, as they are ordinary values.

Formally develop such a language extension, including the abstract syntax of the new constructs, statics, dynamics, precise statement and key cases in the proofs of preservation and progress.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.