Lecture Notes on Exceptions

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
Tuesday, October 8, 2019

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 pattern matching and also exceptions. As always, we consider statics and dynamics and the important theorems showing that they cohere. Before that, we'll complete our discuss of pattern matching.

2 Dynamics of Pattern Matching

We recall the critical rule in the dynamics of pattern matching, where we make a small change in case there is no match: instead of simply stating that we have a MatchException we actually *raise* the exception Match. We postpone further discussion of this to the next section.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ B \mapsto \mathsf{case}\ e_0'\ B}\ \mathsf{step/case}_0$$

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

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

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

The two rules $\mathsf{step/case/match}$ and $\mathsf{step/case/nomatch}$ are distinguished by whether or not there is a simultaneous substitution η for all the variables in the pattern p such that $v = [\eta]p$. This raises the excellent question if we can actually decide between these two rules and how to effectively compute the substitution.

We define a (simultaneous, dropping this adjective from now on) substitution η of values for variables. As for contexts, the variables v_i must all be distinct.

$$\text{Substitution} \quad \eta \quad ::= \quad (v_1/x_1, \ldots, v_n/x_n)$$

Following our general strategy to communicate via inferences rules, we now define a judgment

$$v = [\eta]p$$

Declaratively, it means that applying η to p yields v. Algorithmically, we assume we are given v and p and try to compute η, if it exists, or fail otherwise. So, viewed as an algorithm the rules take a (closed) value v and a pattern p and compute an optional η. The easiest case are variables.

$$\frac{}{v = [v/x]x}\ \mathsf{match/var}$$

Algorithmically, matching a value v against a variable x yields the singleton substitution v/x.

For pairs, both the value and the pattern must be pairs, and the components must again match. Note that typing (discussed below) should guarantee that both the value and the pattern have this form.

$$\frac{v_1 = [\eta_1]x_1 \quad v_2 = [\eta_2]x_2}{\langle v_1, v_2\rangle = [\eta_1, \eta_2]\langle x_1, x_2\rangle}\ \textsf{match/prod}$$

Algorithmically, this makes sense. In the conclusion, we are given $\langle v_1, v_2\rangle$ and $\langle p_1, p_2\rangle$ so the premises have the right information to either fail (in which case we also have no match) or compute η_1 and η_2.

But we need to be concerned about whether the two substitutions η_1 and η_2 might have a variable in common, in which case (η_1, η_2) would violate our presupposition that all variables are distinct. Fortunately, we assumed earlier that the variables in a pattern must all be distinct, usually stated as "*patterns are linear*". This is implied in the following crucial typing rule for patterns:

$$\frac{x : \_ \notin \Gamma \quad \Gamma, x : \tau \ ; \Phi \vdash e : \sigma}{\Gamma \ ; (x : \tau)\ \Phi \vdash e : \sigma}\ \textsf{pat/var}$$

Typing with a pattern context Φ would eventually fail if a variable occurred more than once. Because the variables are already in the context, we cannot simply rename them in the same way we do for bound variable, say, in the rule for λ-abstraction.

The remaining rules are now straightforward.

$$\frac{}{\langle\,\rangle = [\cdot]\langle\,\rangle}\ \textsf{match/unit} \qquad \frac{v = [\eta]p}{\textsf{fold}\ v = [\eta](\textsf{fold}\ p)}\ \textsf{match/rec}$$

$$\frac{v = [\eta]p}{i \cdot v = [\eta](i \cdot p)}\ \textsf{match/sum}$$

The match/sum rule is the only one that can fail if the value v and pattern p have the same type, which you can derive from the canonical form theorem. The way it would fail is if the label in the value did not match the label in the pattern. This is of course the point and the core of pattern matching.

Now it should be clear that the two conditions for the rules step/case/match and step/case/nomatch in the dynamics can be effectively decided by the algorithmic interpretation of these rules. So we can execute our dynamics. Moreover, the expense of matching is linear in the size of the pattern, since we only have to break down the values to the extent prescribed by the pattern. If we had nonlinear patterns, then matching, for example, $\langle v_1, v_2\rangle$ against $\langle x, x\rangle$ would require deciding $v_1 = v_2$. Besides the fact that this make no sense for functions, it is also a linear time in the size of the values.

3 Preservation for Pattern Matching

From looking at the rules for substitution and considering the typing of the collected variables, we see that a substitution η is typed by a context Γ, written as $\eta : \Gamma$, requiring that for each v/x in η, the value v has the type prescribed by $x : \tau$ in Γ.

$$\frac{v\ \mathit{val} \quad \cdot \vdash v : \tau}{(v/x) : (x : \tau)}\ \textsf{subst/var} \qquad \frac{}{(\cdot) : (\cdot)}\ \textsf{subst/empty}$$

$$\frac{\eta_1 : \Gamma_1 \quad \eta_2 : \Gamma_2 \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{(\eta_1, \eta_2) : (\Gamma_1, \Gamma_2)}\ \textsf{subst/join}$$

Here, we made the disjointness conditions explicit, writing $\mathrm{dom}(\Gamma) = \{x \mid x : \_ \in \Gamma\}$. This notation arises from the view of Γ as a function from variables to their types. With this definition, we get the following version of the substitution property.

Theorem 1 (Simultaneous Substitution)
If $\eta : \Gamma$ *and* $\Gamma \vdash e : \sigma$ *then* $\cdot \vdash [\eta]e : \sigma$

We prove this by generalizing to allow $\Gamma', \Gamma \vdash e : \sigma$, yielding $\Gamma' \vdash [\eta]e : \sigma$ and then proceeding by rule induction on the typing derivation of e.

Now we are (almost) ready for the preservation theorem.

Theorem 2 (Type Preservation, for Pattern Matching)
If $\cdot \vdash e : \sigma$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \sigma$

Proof: As before, by rule induction on $e \mapsto e'$, applying inversion on $\cdot \vdash e$.

Case:

$$\frac{e_0 \mapsto e_0'}{\textsf{case}\ e_0\ B \mapsto \textsf{case}\ e_0'\ B}\ \textsf{step/case}_0$$

Straightforward: by inversion we obtain a type τ of e_0 which is preserved by induction hypothesis. From that we can type e'.

Case:

$$\frac{v\ \mathit{val} \quad v = [\eta]p}{\textsf{case}\ v\ (p \Rightarrow e_1 \mid B) \mapsto [\eta]e_1}\ \textsf{step/case/match}$$

This is the most interesting case. There is one typing rule of case, which, by inversion, must have the form

$$\frac{\cdot \vdash v : \tau \quad \cdot \vdash \tau \triangleright (p \Rightarrow e_1 \mid B) : \sigma}{\Gamma \vdash \mathsf{case}\ v\ (p \Rightarrow e_1 \mid B) : \sigma}\ \mathsf{case}$$

We again apply inversion to the second premise, whose typing derivation must look like

$$\frac{\cdot\,;(p : \tau) \vdash e_1 : \sigma \quad \cdot \vdash \tau \triangleright B : \sigma}{\cdot \vdash \tau \triangleright (p \Rightarrow e_1 \mid B) : \sigma}\ \mathsf{branch/alt}$$

Assembling the relevant information, we have

v *val*
$\cdot \vdash v : \tau$
$v = [\eta]p$
$\cdot\,;(p : \tau) \vdash e_1 : \sigma$

To Show: $\cdot \vdash [\eta]e_1 : \sigma$

We would get this from simultaneous substitution (Theorem 1) if the information we have implied that there exists a Γ' such that $\eta : \Gamma'$ and $\Gamma' \vdash e_1 : \sigma$. This is indeed the case, but is a bit painful to show because of the way we defined $\Gamma\,;\Phi \vdash e : \sigma$. Inductively, we have a substitution $v_i = [\eta_i/x_i]p_i$ for every $p_i : \tau_i \in \Phi$ with $v_i : \tau_i$ which give use the above property. We do not formalize this further, but see the remark after this proof.

Case:

$$\frac{v\ \textit{val} \quad \text{there is no } \eta \text{ with } v = [\eta]p}{\mathsf{case}\ v\ (p \Rightarrow e_1 \mid B) \mapsto \mathsf{case}\ v\ B}\ \mathsf{step/case/nomatch}$$

This follows by two inversions on the giving typing derivation followed by a reapplication of the tp/case rule.

Case:

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

Here we realize that we need a type for raise Match. But this expression is not a value and never returns a value, so like $\bot$ it can have any type (see the next section). Therefore the type is preserved trivially.

□

From the proof we need

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

where E is an arbitrary exception. So far, we only have Match but will consider more in the next section.

Also, we realize we would have been better off if our typing rules for a branch in a pattern match had been:

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright B : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid B) : \sigma}\ \mathsf{branch/alt}'$$

where $\Gamma' \Vdash p : \tau$ forces each variable in Γ' to have exactly one occurrence in p. But since we made this decision in the last lecture, we leave this exploration to Exercise 1.

4 Progress for Pattern Matching

We can immediately see that the progress theorem will have to be modified, since the outcome of a computation could be either a value or a raise exception, when no pattern applies. We could avoid this superficially, for example, by simply requiring that the patterns are always *exhaustive*, that is, capture all the possible values of the given type. In practice, though, we often have property we know but the type system does not understand which make some cases impossible. So, like some compiler, one could always add a "catch-all" case at the end that would raise an exception. But this then also requires us to treat exceptions in the language. Instead of raising an exception, we could also just not terminate using $\mathsf{fix}\ f.\ f$ or $\bot$ (as introduced in Exercise L6.2), but from a pragmatic perspective this would be unfortunate.

The good news is that the rest of the progress theorem follows entirely familiar patterns, so we are moving on to the treatment of exceptions.

5 Progress for Exceptions

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

Theorem 3 (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.

Here we imagine that we extended the syntax of expressions

$$\begin{array}{llcl} \text{Expressions} & e & ::= & \ldots \mid \mathsf{case}\ e\ B \mid \mathsf{raise}\ E \\ \text{Exceptions} & E & ::= & \mathsf{Match} \mid \ldots \end{array}$$

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

Trying to prove this will uncover the fact that, currently, this theorem is false for our language. Consider, as a simple example, $\langle \mathsf{raise\ Match}, \langle\,\rangle\rangle$. 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 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 for pairs.

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

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

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 3) 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. □

6 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.

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

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

What type do we need to assign to $\mathsf{try}\ e_1\ e_2$ 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/success, 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 2 to work out the details.

Exceptions in this lecture and Exercise 2 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 as well. 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 $\Gamma' \Vdash p : \tau$ (as introduced at the end of Section 3) to express that $\Gamma' \vdash p : \tau$ but also that every variable in Γ' has exactly one occurrence in p.

1. Define the judgment $\Gamma' \Vdash p : \tau$ by inference rule so it has the stated properties. In additional your rules should have the following algorithmic interpretation: *Given p and τ either compute Γ' or fail.*

2. Redo the case for the rule step/case/match in the proof of preservation (Theorem 2) using the following alternative rule for typing an alternative in a branch expression:

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright B : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid B) : \sigma}\ \mathsf{branch/alt}'$$

   Carefully state any additional lemmas you might need over and above simultaneous substitution (Theorem 1).

Exercise 2 We would like to generalize the try construct to so it can branch on the exception that may have raised. So we have

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

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).

References

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