Lecture Notes on Pattern Matching

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
Thursday, October 8, 2020

1 Introduction

As we have seen in the last lecture, tagged sums together with pattern matching allow us to combine the elimination forms for a variety of types, namely products $\tau_1 \times \tau_2$, unit 1, sums $\sum_{i \in I}(i : \tau_i)$, and recursive types $\rho\alpha.\,\tau$. Allowing patterns to be nested allows for shorter, more easily understandable programs, but they eventually lead to more fundamental changes in the language statics and dynamics. In this lecture we will make these changes to illustrate how a still foundational language can start to become closer and closer to a practical functional language.

2 Nested Cases

As a simple example from the last lecture, consider the specification of a function that divides a unary number by two, rounding down.

$$
\begin{array}{lcl}
\mathit{nat} & = & \rho\alpha.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \alpha) \\
\mathit{half} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{half}\ \mathit{zero} & = & \mathit{zero} \\
\mathit{half}\ (\mathit{succ}\ \mathit{zero}) & = & \mathit{zero} \\
\mathit{half}\ (\mathit{succ}\ (\mathit{succ}\ n'')) & = & \mathit{succ}\ (\mathit{half}\ n'')
\end{array}
$$

With nested patterns, we could write this as

$$
\begin{array}{l}
\mathit{half} = \mathsf{fix}\, \mathit{half}.\, \lambda n. \\
\quad \mathsf{case}\ n\ \ (\ \mathsf{fold}\ \mathbf{zero} \cdot \langle\,\rangle \Rightarrow \mathit{zero} \\
\qquad\quad \mid \mathsf{fold}\ \mathbf{succ} \cdot \mathsf{fold}\ \mathbf{zero} \cdot \langle\,\rangle \Rightarrow \mathit{zero} \\
\qquad\quad \mid \mathsf{fold}\ \mathbf{succ} \cdot \mathsf{fold}\ \mathbf{succ} \cdot n'' \Rightarrow \mathit{succ}\ (\mathit{half}\ n'')\ )
\end{array}
$$

This is now quite close to the specification, but using fold and tags **zero** and **succ** instead of using the constructor functions *zero* and *succ* we use in the mathematical specification.

This example also shows why we prefer the destructor for recursive types to be a fold pattern rather than the primitive unfold: without it, we would have to interrupt our nested pattern and distinguish cases further after unfolding subterms explicitly. Using it as a pattern constructor is also justified by the fact that it is eager, so it exposes a value underneath that we can match against.

3 General Pattern Matching

Based on these examples, we now unify all the different case expressions into a single one. For this, we need two new categories of syntax: *branches* bs and *patterns* p. Patterns are either variables, or value constructors for one of types (omitting those that are opaque). We elide here fixed points as well as values whose structure should not be observable, namely functions $\lambda x.\, e$ and type function $\Lambda\alpha.\, e$.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid \langle e_1, e_2\rangle \mid \langle\,\rangle \mid i \cdot e \mid \mathsf{fold}\ e \mid \mathsf{case}\ e\ (bs) \mid \ldots \\
\text{Patterns} & p & ::= & x \mid \langle p_1, p_2\rangle \mid \langle\,\rangle \mid i \cdot p \mid \mathsf{fold}\ p \\
\text{Branches} & bs & ::= & \cdot \mid (p \Rightarrow e \mid bs)
\end{array}
$$

Because we have new forms of expression, there will also be new judgments for typing the constructs. Let's see what these might be by starting with the rule for case expressions.

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

The new judgment here is

$$\Gamma \vdash \tau \triangleright bs : \sigma$$

We read this as

> *Match a case subject of type τ against the branches bs, each of which must have type σ.*

The reason all branches must have the same type is the same as for the conditionals or branching over a sum: we don't know which branch will be taken when the program runs. Furthermore, each pattern in bs should match the type τ. Because there are two alternatives for branches in the syntax, we have two typing rules for branches. The first (tp/bs/alt) checks the first branch and then the remaining ones. The second (tp/bs/none) expresses that once all branches have been checked, there are no further constraints on τ and σ.

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

The first rule here uses a new judgment, $\Gamma' \Vdash p : \tau$. This is almost like the judgment $\Gamma \vdash p : \tau$, noting that every pattern is also an expression. However, it is more restrictive in that the variables in Γ' must be *exactly* the variables in p and, moreover, variables in p may be not occur more than once.[1] We define it with the rules below. When we think about the rules in this set, it may be helpful to keep in mind that when type-checking we know the type τ and the pattern p and we try to *generate* the context Γ', assigning a type to each free variable in p (assuming such a Γ' exists; otherwise there will be no derivation for the given p and τ).

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{pat/var} \qquad \frac{\Gamma_1 \Vdash p_1 : \tau_1 \quad \Gamma_2 \Vdash p_2 : \tau_2}{\Gamma_1, \Gamma_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2}\ \mathsf{pat/pair} \qquad \frac{}{\cdot \Vdash \langle\,\rangle : 1}\ \mathsf{pat/unit}$$

$$\frac{(k \in I) \quad \Gamma \Vdash p : \tau_k}{\Gamma \Vdash k \cdot p : \sum_{i \in I} (i : \tau_i)}\ \mathsf{pat/inject} \qquad \frac{\Gamma \Vdash p : [\rho\alpha.\,\tau/\alpha]\tau}{\Gamma \Vdash \mathsf{fold}\ p : \rho\alpha.\,\tau}\ \mathsf{pat/fold}$$

It is implicit in the rule for pairs that Γ_1 and Γ_2 are disjoint in their variables, which means that patterns may contain neither duplicate variables nor extraneous variables. For example, we have

$$x : \mathit{nat} \Vdash \mathsf{fold}\ \mathbf{succ} \cdot x : \mathit{nat}$$

but we can *not* have

$$x : \mathit{nat}, y : \mathit{nat} \Vdash \mathsf{fold}\ \mathbf{succ} \cdot x : \mathit{nat}$$

because such a judgment would allow e in the branch

$$\mathsf{fold}\ \mathbf{succ} \cdot x \Rightarrow e$$

[1] In lecture, we did not distinguish this judgment, but without this distinction this rule would be ambiguous.

to mention y (which will not be bound when the value is matched against a pattern). Therefore, the judgment $\Gamma \Vdash p : \tau$ must be "tight" in the sense that Γ contains precisely the variables in p.

For an ordinary hypothetical judgment $\Gamma \vdash e : \tau$ we have certain properties, the most important of which is substitution. But we also have *weakening* which means we can always adjoin another hypothesis without changing the validity of the derivation. That is, if $\Gamma \vdash e : \tau$ then also $\Gamma, x : \sigma \vdash e : \tau$ as long as $\Gamma, x : \sigma$ is a well-formed context. However, this is *not* the case for the typing of patterns, as explained above. This is an example of a *linear hypothetical judgment* where all hypotheses must be used exactly once.

This new set of typing rules is still *syntax-directed*, which now includes not only the typing of expressions, but also the typing of branches and patterns.

4 An Example: Equality on Binary Numbers

Before formalizing the operational semantics, we return to the binary numbers and write a function to increment and then a second one to test equality. We use here the concrete syntax of LAMBDA. Tags are preceded by a tick mark to distinguish them syntactically from variables. They generalize `'l` and `'r` from the binary sums. We begin with the "boilerplate" code, defining a recursive type and then the constructors with their types. In a slight deviation from the previous encoding, we say that $e : 1 \to \mathit{bin}$ so that each constructor, uniformly, takes the type of the correspondingly tagged value as an argument.

```
1 type bin = $a. ('b0 : a) + ('b1 : a) + ('e : 1)
2
3 decl b0 : bin -> bin
4 decl b1 : bin -> bin
5 decl e : 1 -> bin
6
7 defn b0 = \x. fold 'b0 x
8 defn b1 = \x. fold 'b1 x
9 defn e = \u. fold 'e u
```

Incrementing a number is now straightforward, using a simple pattern match based on the three possible tags of a value of type *bin*.

```
decl inc : bin -> bin
defn inc = $inc. \x.
   case x of ( fold 'b0 y => b1 y
             | fold 'b1 y => b0 (inc y)
             | fold 'e _ => b1 (e ()) )
```

A first cut in the implementation of equality uses pattern matching against a pair of binary numbers.

```
type bool = ('true : 1) + ('false : 1)
decl true : bool
decl false : bool
defn true = 'true ()
defn false = 'false ()

% warning: this implementation is incorrect!
decl eq : bin -> bin -> bool
defn eq = $eq. \x. \y.
   case (x,y) of ( (fold 'b0 u, fold 'b0 w) => eq u w
                 | (fold 'b1 u, fold 'b1 w) => eq u w
                 | (fold 'e  _, fold 'e  _) => true )
```

This implementation is clearly incorrect, because we somehow have to say "*when none of the other patterns match, return false*". But this is easy, either using variables that don't occur or the special wildcard which syntactically reminds us of this fact.

```
% warning: this implementation is still incorrect!
decl eq : bin -> bin -> bool
defn eq = $eq. \x. \y.
  case (x,y) of ( (fold 'b0 u, fold 'b0 w) => eq u w
                | (fold 'b1 u, fold 'b1 w) => eq u w
                | (fold 'e  _, fold 'e  _) => true
                | (_, _) => false )
```

In fact, we could also use a single _ to match against the pair in this last catch-all case.

This implementation, however, is still incorrect. See if you can spot and fix the bug before moving on to the next page.

The problem is that the representation allows leading zeros, so that *eq* (*e* ⟨⟩) (*b0* (*e* ⟨⟩)) should return *true* but the current implementation returns *false*.

We could fix the problem by *standardizing* the numbers and then using equality only on numbers in standard form. It is easy to make a mistake there, however, unless you also track standard forms in the types, as suggested in Exercise L11.1.

Alternatively, we can change the definition to account for leading zeros by stripping them away during the comparisons against zero (represented here by the pattern fold $\mathbf{e} \cdot \_$).

```
% relax: this implementation is now correct!
decl eq : bin -> bin -> bool
defn eq = $eq. \x. \y.
  case (x,y) of ( (fold 'b0 u, fold 'b0 w) => eq u w
                | (fold 'b1 u, fold 'b1 w) => eq u w
                | (fold 'e  _, fold 'b0 w) => eq x w
                | (fold 'b0 u, fold 'e  _) => eq u y
                | (fold 'e  _, fold 'e  _) => true
                | _                        => false )
```

This definition now works correctly. Note that in the two new branches that strip tags `'b0'` we exploit the fact that we know the one of the elements of the pair are actually zero and are bound to a variable already (the function arguments x and y, respectively). We could write the slightly more verbose `eq (e ()) w` and `eq u (e ())` instead.

5 Dynamics of Pattern Matching

The dynamics now also has to deal with pattern matching, and up to a certain point it seems less complicated. When we match a value v against a pattern p, this match either has to fail or return to us a substitution η for all the variables in p. We write this as either $v = [\eta]p$ or "*there is no* η *with* $v = [\eta]p$". This η is a simultaneous substitution for all the variables in p which we write as $(v_1/x_1, \ldots, v_n/x_n)$. Matching proceeds sequentially through the patterns. If it reaches the end of the branches and no case has matched, it transitions to raising a Match exception, which is a new possible

outcome of a computation.

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

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

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

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

If we allow raise Match to have every possible type, then the preservation theorem still goes through. Furthermore, since it is not a value, the canonical forms theorem will continue to hold. However, the progress theorem now has to change: a closed well-typed expression either can take a step or is a value or raises a match exception.

This may be somewhat unsatisfactory because the slogan "*well-typed programs do not go wrong*" no longer applies in its purest form. However, the progress theorem (once carefully spelled out) still characterizes the possible outcomes of computations exactly.

In order to avoid this unpleasantness, in Standard ML (SML) it is assumed that pattern matches are exhaustive. If the compiler determines that a given set of patterns is not, it adds a catch-all final branch at the end. However, this branch reads "_ ⇒ raise Match" (exploiting the presence of exceptions in SML) which is therefore no different from the semantics we gave above.

We will complete the discussion of pattern matching and exceptions in the next lecture.

Exercises

Exercise 1 In this exercise we explore how to make the rules for pattern matching slightly less abstract.

(i) Define $v \triangleright p \mapsto [\eta]$, as a three-place judgment using inference rules. This judgment should be derivable exactly if $v = [\eta]p$ (but you do not need to prove that).

(ii) Define $\eta : \Gamma$ as a two-place judgment using inference rules. This judgment should be derivable if η substitutes a value of suitable type for each variable in Γ.

(iii) Prove that if $\eta : \Gamma$ and $\Gamma \Vdash p : \tau$ then $[\eta]p$ *value* and $\cdot \vdash [\eta]p : \tau$.

(iv) Define a new judgment $v \not\triangleright p$ using inference rules. This judgment should be derivable exactly if there is no substitution η such that $v = [\eta]p$.

(v) It should be the case that for any v and p, either $v \triangleright p \mapsto \eta$ or $v \not\triangleright p$. State any additional assumptions you may need (say, on the typing of v or p) and sketch the proof. Include two representative cases.

(vi) Revise the dynamics for pattern matching to use the new judgments devised above.

(vii) Revise the proof of *preservation* to account for general pattern matching. Essentially, remove all the cases for individual destructors and then add in a case for the generalized case construct. The key question is how to use the properties you have developed above. If you need any additional properties of simultaneous substitutions η please state them, but you do not need to prove them.

Exercise 2 In this exercise we explore the restriction that patterns cannot have any repeated variables.

(i) Explain why in our language we cannot reasonably allow patterns with repeated variables. Be as concrete as possible.

(ii) Explore to which extent repeated pattern variables might make sense and revise the judgment $\Gamma \Vdash p : \tau$ accordingly.