Pattern Matching

CAS CS 320: Concepts of Programming Languages

November 10, 2026

Pattern matching is in my mind a PL superpower. It's not unique to OCaml; lots of PLs (particularly newer ones) implement some notion of ADTs, pattern matching, and exhaustiveness checking. Even Python hacks together something akin to pattern matching (albeit in a way that's certified batsh**). You should by now be familiar with pattern matching from the programmer's perspective. Our goal is to give a formal specification for it.

Base PL. We take the grammar in Figure 1 to specify the abstract syntax of our base PL. The meaning of the constructs therein should be self-evident. Notably we've introduced option types and list types; this'll make pattern matching generally speaking more interesting. The typing and semantics of this base PL are given in Figure 2 and Figure 3, respectively.

Exercise 1. *Write a couple typing derivations and semantic derivations for some basic expressions in our base PL. Make sure you understand how the abstract syntax of our language would correspond to an OCaml-like concrete syntax.*

Exercise 2. *Why do you think we annotate* $[\,]$ *and* None *with types?*

Shallow pattern matching. The easiest way to introduce pattern matching is to create match expressions for each data type that requires it, and fix the match arms to be the possible top-level "shapes" of the data. For example, we could extend our base PL with the syntax, typing, and semantics in Figure 4.

Exercise 3. *For each of the following expressions e, determine a type τ such that $\varnothing \vdash e : \tau$ and a value v such that $\langle\, \varnothing\ ,\ e\, \rangle \Downarrow v$. Also provide the derivations.*

1. $\mathsf{match}\ [\,]^{\mathsf{int}}\ \mathsf{with}\ [\,] \to 1\ \mid\ x :: l \to x + x$
2. $\mathsf{match}\ 1 :: 2 :: [\,]^{\mathsf{int}}\ \mathsf{with}\ [\,] \to 1\ \mid\ x :: l \to x + x$
3. $\lambda l^{\mathsf{List(int)}}.\mathsf{match}\ l\ \mathsf{with}\ [\,] \to \mathsf{true}\ \mid\ x :: l \to \mathsf{false}$

This "shallow" form of match expression is similar to what's called an ***eliminator***. We could, for example, instead just introduce a new higher-order term in our PL that operates on the constituent parts of a list, depending on its shape (Figure 5).

$$
\begin{array}{rcll}
e & ::= & \lambda x^{\tau}.e \mid ee \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e\ o\ e & \text{(expressions)} \\
 & \mid & \mathsf{None}^{\tau} \mid \mathsf{Some}(e) \mid [\,]^{\tau} \mid \mathsf{true} \mid \mathsf{false} \mid x \mid n & \\
o & ::= & + \mid - \mid \times \mid \leq \mid\ :: & \text{(operators)} \\
\tau & ::= & \tau \to \tau \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{Option}(\tau) \mid \mathsf{List}(\tau) & \text{(types)} \\
v & ::= & \mathsf{true} \mid \mathsf{false} \mid n \mid (\!|\, \mathcal{E}\ ,\ \lambda x.e \,|\!) \mid (\!|\, \mathcal{E}\ ,\ \lambda x.e\ ,\ x \,|\!) & \text{(values)} \\
 & \mid & \mathsf{None} \mid \mathsf{Some}(v) \mid [\,] \mid v :: v & \\
x & & & \text{(variables)} \\
n & & & \text{(integers)} \\
\mathcal{E} & & & \text{(environments)}
\end{array}
$$

Figure 1: Abstract syntax of our base PL

$$
\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathsf{int}}\ \textsc{int} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \textsc{true}
$$

$$
\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \textsc{false} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ \textsc{add}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 - e_2 : \mathsf{int}}\ \textsc{sub} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \times e_2 : \mathsf{int}}\ \textsc{mul}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \leq e_2 : \mathsf{bool}}\ \textsc{leq} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x^{\tau_1}.e : \tau_1 \to \tau_2}\ \textsc{fun}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ \textsc{app} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textsc{let}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}\ \textsc{if} \qquad \frac{}{\Gamma \vdash [\,]^{\tau} : \mathsf{List}(\tau)}\ \textsc{nil}
$$

$$
\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathsf{List}(\tau)}{\Gamma \vdash e_1 :: e_2 : \mathsf{List}(\tau)}\ \textsc{cons} \qquad \frac{}{\Gamma \vdash \mathsf{None}^{\tau} : \mathsf{Option}(\tau)}\ \textsc{none}
$$

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Some}(e) : \mathsf{Option}(\tau)}\ \textsc{some}
$$

Figure 2: Type system of our base PL

$$\frac{}{\langle\ \mathcal{E}\ ,\ x\ \rangle \Downarrow \mathcal{E}(x)}\ \textsc{varE} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ n\ \rangle \Downarrow n}\ \textsc{intE} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{true}\ \rangle \Downarrow \mathsf{true}}\ \textsc{trueE}$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{false}\ \rangle \Downarrow \mathsf{false}}\ \textsc{falseE} \qquad \frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1 + e_2\ \rangle \Downarrow v_1 + v_2}\ \textsc{addE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1 - e_2\ \rangle \Downarrow v_1 - v_2}\ \textsc{subE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1 \times e_2\ \rangle \Downarrow v_1 \times v_2}\ \textsc{mulE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 \leq v_2}{\langle\ \mathcal{E}\ ,\ e_1 \leq e_2\ \rangle \Downarrow \mathsf{true}}\ \textsc{leqTrue}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad v_1 > v_2}{\langle\ \mathcal{E}\ ,\ e_1 \leq e_2\ \rangle \Downarrow \mathsf{false}}\ \textsc{leqFalse}$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \lambda x^{\tau}.e\ \rangle \Downarrow (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!)}\ \textsc{funE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad \langle\ \mathcal{E}'[x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ e_1 e_2\ \rangle \Downarrow v}\ \textsc{appE}$$

$$\frac{\begin{array}{c}\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ ,\ x'\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \\ \langle\ \mathcal{E}'[x' \mapsto (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ ,\ x'\ |\!)][x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v\end{array}}{\langle\ \mathcal{E}\ ,\ e_1 e_2\ \rangle \Downarrow v}\ \textsc{appRecE}$$

$$\frac{e_1 \neq \lambda x^{\tau}.e \qquad \langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}[x \mapsto v_1]\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \rangle \Downarrow v_2}\ \textsc{letE}$$

$$\frac{\langle\ \mathcal{E}[x \mapsto (\!|\ \mathcal{E}\ ,\ \lambda x_1.e_1\ ,\ x\ |\!)]\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{let}\ x = \lambda x_1^{\tau}.e_1\ \mathsf{in}\ e_2\ \rangle \Downarrow v}\ \textsc{letRecE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathsf{true} \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \rangle \Downarrow v_2}\ \textsc{ifTrue}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathsf{false} \qquad \langle\ \mathcal{E}\ ,\ e_3\ \rangle \Downarrow v_3}{\langle\ \mathcal{E}\ ,\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \rangle \Downarrow v_3}\ \textsc{ifFalse} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{None}^{\tau}\ \rangle \Downarrow \mathsf{None}}\ \textsc{noneE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{Some}(e)\ \rangle \Downarrow \mathsf{Some}(v)}\ \textsc{someE} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ [\,]^{\tau}\ \rangle \Downarrow [\,]}\ \textsc{nilE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1 :: e_2\ \rangle \Downarrow v_1 :: v_2}\ \textsc{consE}$$

Figure 3: Big step CBV environment semantics for our base PL

$$e \ ::= \ \mathsf{match}\ e\ \mathsf{with}\ [\,]^{\tau} \to e \ \restriction \ x :: x \to e$$

$$\frac{\Gamma \vdash e : \mathsf{List}(\tau') \qquad \Gamma \vdash e_1 : \tau \qquad x_1 \neq x_2 \qquad \Gamma, x_1 : \tau', x_2 : \mathsf{List}(\tau') \vdash e_2 : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ [\,] \to e_1 \ \restriction \ x_1 :: x_2 \to e_2 : \tau}\ \textsc{matchList}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow [\,] \qquad \langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{match}\ e\ \mathsf{with}\ [\,] \to e_1 \ \restriction \ x_1 :: x_2 \to e_2\ \rangle \Downarrow v}\ \textsc{matchListNil}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v_1 :: v_2 \qquad \langle\ \mathcal{E}[x_1 \mapsto v_1][x_2 \mapsto v_2]\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{match}\ e\ \mathsf{with}\ [\,] \to e_1 \ \restriction \ x_1 :: x_2 \to e_2\ \rangle \Downarrow v}\ \textsc{matchListCons}$$

Figure 4: Shallow pattern matching for lists

$$e \ ::= \ \mathsf{elimList}(e, e, e)$$

$$\frac{\Gamma \vdash e_1 : \mathsf{List}(\tau') \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau' \to \mathsf{List}(\tau') \to \tau}{\Gamma \vdash \mathsf{elimList}(e_1, e_2, e_3) : \tau}\ \textsc{elimList}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow [\,] \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{elimList}(e_1, e_2, e_3)\ \rangle \Downarrow v}\ \textsc{elimListNil}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 :: v_2 \qquad \langle\ \mathcal{E}\ ,\ e_3\ v_1\ v_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{elimList}(e_1, e_2, e_3)\ \rangle \Downarrow v}\ \textsc{elimListCons}$$

Figure 5: Eliminator for lists

Exercise 4. *For each of the following expressions e, determine a type τ such that $\varnothing \vdash e : \tau$ and a value v such that $\langle\, \varnothing\ ,\ e\, \rangle \Downarrow v$. Also provide the derivations.*

1. $\mathsf{elimList}(1 :: 2 :: [\,]^{\mathsf{int}}, 0, \lambda x^{\mathsf{int}}.\lambda l^{\mathsf{List(int)}}.x + x)$
2. $\lambda l^{\mathsf{List(int)}}.\lambda r^{\mathsf{List(int)}}.\mathsf{elimList}(l, \mathsf{false}, \lambda x_1.\lambda x_2.\mathsf{elimList}(r, \mathsf{false}, \lambda y_1.\lambda y_2.x_1 \leq y_1))$

Exercise 5. *Give the equivalent big-step substitution semantics for both shallow list match expressions and list eliminators.*

Exercise 6. *Give a translation between shallow list match expressions and list eliminators. That is, given an arbitrary shallow list match expression, express it as an expression with a list eliminator that is semantically equivalent (and vice versa).*

Exercise 7. *Give the syntax, typing, and semantic rules for shallow option match expressions and option eliminators.*

Exercise 8. *Should we introduce a shallow Boolean match expression or a Boolean eliminator? Explain why and/or why not.*

The issue with shallow match expressions (and eliminators) is that, well, they're shallow. This does not affect our ability to implement certain algorithms, but it does affect the concision with which we can do so. Suppose, for example, we want to write the function that determines the third element of a list:

```
let third l =
  match l with
  | _ :: _ :: x :: _-> Some x
  | _ -> None
```

Easy. But if we were forced to use the "shallow" form of this match expression, the implementation is tedious.

```
let third l =
  match l with
  | [] -> None
  | _ :: l ->
    match l with
    | [] -> None
    | _ :: l ->
      match l with
      | [] -> None
      | x :: l -> Some x
```

Exercise 9. *Implement the* `third` *function for integer lists in our base PL using list eliminators.*

Exercise 10. *Implement in our base PL the function that determines the sum of the elements in an integer list, using shallow list match expressions.*

Deep pattern matching. Introducing the ability to match on arbitrary nested patterns of data is an incredibly powerful tool, but it comes at the cost of specification complexity. Part of the issue is that *it's no longer guaranteed that match expressions are exhaustive.* And the dual issue is that *it's possible to have redundant matches.* The former means potentially unsafe code (i.e., code that crashes at runtime) and the latter means potentially confusing code (e.g., bugs that occur because a match case is never reached). It's outside of the scope of these notes to cover exhaustiveness and redundancy, but they're important to consider when implementing deep pattern-matching.

Syntax of pattern matching. Another issue is that we have to specify a new kind of syntactic category, i.e., *what exactly is a pattern?* We should have the sense that they're values of ADTs, but that they're allowed to have identifiers and ***wildcards*** (i.e., the underscore in OCaml) interspersed within them. This is essentially the definition that we'll take.

$$p \ ::= \ \mathsf{true} \mid \mathsf{false} \mid n \mid \mathsf{None} \mid \mathsf{Some}(p) \mid [\,] \mid p :: p \mid x \mid _ \qquad \text{(patterns)}$$

Note that we don't take functions as patterns.[1] We take this to define the patterns in arms of a match expression.

$$\begin{aligned} e \ &::= \ \mathsf{match}\ e\ \mathsf{with}\ a\ \{a\} && \text{(match expressions)} \\ a \ &::= \ \lceil\ p \to e && \text{(match arms)} \end{aligned}$$

Note that a match expression must have at least one arm.

Typing of pattern matching. The typing and semantics are where things get a bit complicated. We start with typing. There are two things we need to check when type-checking a match expression.

1. All the patterns are "the same type" as e.
2. All the match arm bodies are the "same" type.

The former implies we need to introduce a new kind of judgment for typing patterns. The reason this is tricky is that identifiers in a pattern are meant to indicate parameters that can be used within the body of the match arm. This means that the context *changes* when typing the body of a match arm, depending on the pattern, which means we have to think about how variable declarations are *generated* by patterns. This suggests the following kind of judgment:

$$p : \tau \Rightarrow \Gamma$$

[1] There are a number of reasons for this, but the primary one is that, since we don't evaluate under binders, there are many function values that we should probably consider equal that aren't. But as is usually the case of any PL feature you could dream up, there's probably someone who's tried it; there are experimental PLs that allow for function pattern matching.

$$\frac{}{\mathsf{true} : \mathsf{bool} \Rightarrow \varnothing}\ \textsc{trueP} \qquad \frac{}{\mathsf{false} : \mathsf{bool} \Rightarrow \varnothing}\ \textsc{falseP} \qquad \frac{n \in \mathbb{Z}}{n : \mathsf{int} \Rightarrow \varnothing}\ \textsc{intP}$$

$$\frac{}{\mathsf{None} : \mathsf{Option}(\tau) \Rightarrow \varnothing}\ \textsc{noneP} \qquad \frac{}{[\,] : \mathsf{List}(\tau) \Rightarrow \varnothing}\ \textsc{nilP}$$

$$\frac{p : \tau \Rightarrow \Gamma}{\mathsf{Some}(p) : \mathsf{Option}(\tau) \Rightarrow \Gamma}\ \textsc{someP}$$

$$\frac{p_1 : \tau \Rightarrow \Gamma_1 \qquad p_2 : \mathsf{List}(\tau) \Rightarrow \Gamma_2 \qquad \Gamma_1 \text{ and } \Gamma_2 \text{ are disjoint}}{p_1 :: p_2 : \mathsf{List}(\tau) \Rightarrow \Gamma_1, \Gamma_2}\ \textsc{consP}$$

$$\frac{}{x : \tau \Rightarrow \{x : \tau\}}\ \textsc{varP} \qquad \frac{}{_ : \tau \Rightarrow \varnothing}\ \textsc{wildP}$$

Figure 6: Pattern typing rules

which we read as "p has the type τ and *generates* the variable declarations Γ." Note that we don't need to type patterns in contexts; the identifiers in patterns aren't free variables that need to be typed, but parameters that need to be *given* types. The rules for these pattern typing judgments are given in Figure 6. The most important of these rules is the VARP rule, which states that a variable pattern *generates* a new variable declaration. These are collected in the SOMEP and CONSP rules as they get generated by the constituent parts of the pattern. Another subtlety: we require that identifiers are distinct within a pattern. This is mostly to make our match expressions easier to understand by disallowing shadowing within a pattern, but it also coincides with the behavior of OCaml.

```
utop> match [] with x :: x -> false;;
Error: Variable x is bound several times in this matching
```

Exercise 11. *Give derivations for each of the following pattern typing judgments.*

1. $\mathsf{Some}(x) : \mathsf{Option}(\mathsf{int}) \Rightarrow \{x : \mathsf{int}\}$
2. $\mathsf{Some}(\mathsf{true}) :: y : \mathsf{List}(\mathsf{Option}(\mathsf{bool})) \Rightarrow \{y : \mathsf{List}(\mathsf{Option}(\mathsf{bool}))\}$
3. $x :: y :: l : \mathsf{List}(\mathsf{int}) \Rightarrow \{x : \mathsf{int}, y : \mathsf{int}, l : \mathsf{List}(\mathsf{int})\}$
4. $_ :: _ :: x :: _ : \mathsf{List}(\mathsf{int}) \Rightarrow \{x : \mathsf{int}\}$

We can now use this new form of judgment within our typing rule for deep

$$\frac{}{\mathsf{true\ matches\ true} \Rightarrow \varnothing}\ \textsc{trueM} \qquad \frac{}{\mathsf{false\ matches\ false} \Rightarrow \varnothing}\ \textsc{falseM}$$

$$\frac{n \in \mathbb{Z}}{n\ \mathsf{matches}\ n \Rightarrow \varnothing}\ \textsc{intM} \qquad \frac{}{\mathsf{None\ matches\ None} \Rightarrow \varnothing}\ \textsc{noneM}$$

$$\frac{}{[\,]\ \mathsf{matches}\ [\,] \Rightarrow \varnothing}\ \textsc{nilM} \qquad \frac{v\ \mathsf{matches}\ p \Rightarrow \mathcal{E}}{\mathsf{Some}(v)\ \mathsf{matches}\ \mathsf{Some}(p) \Rightarrow \mathcal{E}}\ \textsc{someM}$$

$$\frac{v_1\ \mathsf{matches}\ p_1 \Rightarrow \mathcal{E}_1 \qquad v_2\ \mathsf{matches}\ p_2 \Rightarrow \mathcal{E}_2}{v_1 :: v_2\ \mathsf{matches}\ p_1 :: p_2 \Rightarrow \mathcal{E}_1 \cup \mathcal{E}_2}\ \textsc{consM}$$

$$\frac{}{v\ \mathsf{matches}\ x \Rightarrow \{x \mapsto v\}}\ \textsc{varM} \qquad \frac{}{v\ \mathsf{matches}\ _ \Rightarrow \varnothing}\ \textsc{wildM}$$

Figure 7: Pattern matching rules

pattern matching.

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad p_1 : \tau_0 \Rightarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash e_1 : \tau \qquad \ldots \qquad p_k : \tau_0 \Rightarrow \Gamma_k \qquad \Gamma, \Gamma_k \vdash e_k : \tau}{\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\quad \mid\ p_1 \to e_1 \ldots\ \mid\ p_k \to e_k : \tau}\ \textsc{match}$$

Note that the generated variable declarations are used to type the bodies of the match arms.

Exercise 12. *Give a derivation for the following typing judgment.*

$$\varnothing \vdash \lambda l^{\mathsf{List(int)}}.\mathsf{match}\ l\ \mathsf{with}\quad \mid\ _ :: x :: _ \to \mathsf{Some}(x)\ \mid\ _ \to \mathsf{None}^{\mathsf{int}} : \mathsf{List(int)} \to \mathsf{Option(int)}$$

Semantics of pattern matching. Finally, we describe how to evaluate a match expression. Just as with typing, matching a pattern *generates* new stuff, this time variable bindings in the environment (or substitutions, if we're working in the substitution model). The new form of judgment we'll need is

$$v\ \mathsf{matches}\ p \Rightarrow \mathcal{E}$$

which we read as "the value v matches the pattern p and generates the variable bindings $\mathcal{E}$". The rules for pattern matching judgments are given in Figure 7. Again, any value can match with identifiers in patterns, and this creates a new binding via the varM rule. In these rules and below, the environment $\mathcal{E}_1 \cup \mathcal{E}_2$ refers to the combined bindings in both $\mathcal{E}_1$ and $\mathcal{E}_2$ with bindings in $\mathcal{E}_2$ potentially shadowing those of $\mathcal{E}_1$.

Exercise 13. *Give derivations for each of the following pattern matching judgments.*

$$
\begin{array}{rcll}
e & ::= & \mathsf{match}\ e\ \mathsf{with}\ a\ \{a\} & \text{(match expressions)} \\
a & ::= & \mid\ p \to e & \text{(match arms)} \\
p & ::= & \mathsf{true} \mid \mathsf{false} \mid n & \text{(patterns)} \\
 & \mid & \mathsf{None} \mid \mathsf{Some}(p) \mid [\,] \mid p :: p \mid x \mid _ &
\end{array}
$$

$$
\frac{\langle\ \mathcal{E}\ ,\ e_0\ \rangle \Downarrow v_0 \qquad \forall j < i.\neg(v_0\ \mathsf{matches}\ p_j) \qquad v_0\ \mathsf{matches}\ p_i \Rightarrow \mathcal{E}_i \qquad \langle\ \mathcal{E} \cup \mathcal{E}_i\ ,\ e_i\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{match}\ e_0\ \mathsf{with}\ \ \mid\ p_1 \to e_1 \ldots\ \mid\ p_k \to e_k\ \rangle \Downarrow v}\ \textsc{matchE}
$$

$$
\frac{\Gamma \vdash e_0 : \tau_0 \qquad p_1 : \tau_0 \Rightarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash e_1 : \tau \qquad \ldots \qquad p_k : \tau_0 \Rightarrow \Gamma_k \qquad \Gamma, \Gamma_k \vdash e_k : \tau}{\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \ \mid\ p_1 \to e_1 \ldots\ \mid\ p_k \to e_k : \tau}\ \textsc{match}
$$

Figure 8: Syntax, typing, and semantic rules for deep pattern matching

1. $\mathsf{Some}(2)\ \mathsf{matches}\ \mathsf{Some}(x) \Rightarrow \{x \mapsto 2\}$
2. $\mathsf{Some}(\mathsf{true}) :: [\,]\ \mathsf{matches}\ \mathsf{Some}(\mathsf{true}) :: y \Rightarrow \{y \mapsto [\,]\}$
3. $1 :: 2 :: 3 :: [\,]\ \mathsf{matches}\ x :: y :: l \Rightarrow \{x \mapsto 1, y \mapsto 2, l \mapsto 3 :: [\,]\}$
4. $1 :: 2 :: 3 :: 4 :: [\,]\ \mathsf{matches}\ _ :: _ :: x :: _ \Rightarrow \{x \mapsto 3\}$

And finally, the semantic rule. The last point to make here is that match arms are evaluated from top to bottom. This requires a clunky side-condition which is meant to simply express that the expression evaluates to the value of the body of the *first* arm whose pattern matches the value of the matched expression.

$$
\frac{\langle\ \mathcal{E}\ ,\ e_0\ \rangle \Downarrow v_0 \qquad \forall j < i.\neg(v_0\ \mathsf{matches}\ p_j) \qquad v_0\ \mathsf{matches}\ p_i \Rightarrow \mathcal{E}_i \qquad \langle\ \mathcal{E} \cup \mathcal{E}_i\ ,\ e_i\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{match}\ e_0\ \mathsf{with}\ \ \mid\ p_1 \to e_1 \ldots\ \mid\ p_k \to e_k\ \rangle \Downarrow v}\ \textsc{matchE}
$$

Again, we read "$\forall j < i.\neg(v_0\ \mathsf{matches}\ p_j)$" as "$v_0$ does *not* match any of the patterns $p_1, \ldots, p_{i-1}$." A summary of these match expression rules is given in Figure 8.

Exercise 14. *Give a derivation for the following semantic judgment.*

$$
\begin{array}{l}
\langle\ \varnothing\ ,\ \mathsf{match}\ \mathsf{Some}(3) :: \mathsf{None}^{\mathsf{int}} :: [\,]^{\mathsf{Option(int)}}\ \mathsf{with} \\
\qquad \mid\ \mathsf{None} :: _ \to 0 \\
\qquad \mid\ \mathsf{Some}(x) :: \mathsf{Some}(y) :: _ \to x \times y \\
\qquad \mid\ \mathsf{Some}(x) :: _ \to x + 1\ \rangle \Downarrow 4
\end{array}
$$

Exercise 15. *Update our base PL with pattern matching to include rules for result types.*

Exercise 16. *Update our base PL with pattern matching to include rules for binary trees.*