# Lecture Notes on Elaboration

15-814: Types and Programming Languages
Frank Pfenning

Lecture 10
Thursday, October 3, 2019

## 1 Introduction

We have spent a lot of time analyzing and designing the essence of a programming language, starting from first principles. The focus has been on the *statics* (the type system), the *dynamics* (the rules for how to evaluate programs), and understanding the relationship between them in a mathematically rigorous way.

There is, of course, a lot more to a real programming language. At the "front end" there is the *concrete syntax* according to which the program text is parsed. The result of parsing is either some *abstract syntax* or an error message if the program is not well-formed according to the grammar defining its syntax. At the "back end" there are concerns about how a language might be executed efficiently, or *compiled* to machine language so it can run even faster. In this course we will say little about issues of grammar, concrete syntax, parsers or parser generators, because we want to focus on the deeper semantic issues where we have accumulated a lot of knowledge about language design.

In today's lecture we will look at *elaboration*, which is a translation mediating between specific forms of concrete syntax and internal representation in abstract syntax. Elaborating the program allows us to provide some conveniences that make it easy to write and read concise programs without giving up the sound underlying principles we have learned about in this course so far.

## 2 "Syntactic Sugar"

A simple form of elaboration is to eliminate some simple forms of "syntactic sugar" and translate them into an internal form to simplify downstream processing. A good example are the following definitions:

$$\begin{array}{lcl}
\mathsf{bool} & \triangleq & (\mathtt{true} : 1) + (\mathtt{false} : 1) \\
\mathsf{true} & \triangleq & \mathtt{true} \cdot \langle\,\rangle \\
\mathsf{false} & \triangleq & \mathtt{false} \cdot \langle\,\rangle \\
\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \triangleq & \mathsf{case}\ e_1\ (\mathtt{true} \cdot \_ \Rightarrow e_2 \mid \mathtt{false} \cdot \_ \Rightarrow e_3)
\end{array}$$

Here, we used another common convention, name we use an underscore (_) in place of a variable name if that variable does not occur in its scope (here, this scope would be $e_2$ for the first underscore and $e_3$ for the second. Such a syntactic transformation could take place before or after type checking.

## 3 Data Types

Consider for example, the definition of the natural numbers:

$$\mathit{nat} = \rho\alpha.\,(\mathtt{zero} : 1) + (\mathtt{succ} : \alpha)$$

This is unnecessarily difficult to read because we have to remember that $\alpha$ really is supposed to stands for *nat* on the right hand. Easier to read is

$$\mathit{nat} \cong (\mathtt{zero} : 1) + (\mathtt{succ} : \mathit{nat})$$

Moreover, the labels may sometimes be a bit awkward to use, so perhaps we could "automatically" define

$$\begin{array}{lcl}
\mathit{zero} & : & 1 \to \mathit{nat} \\
\mathit{zero} & = & \lambda u.\,\mathtt{zero} \cdot u \\
\mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{succ} & = & \lambda n.\,\mathtt{succ} \cdot n
\end{array}$$

Notice there the difference between the *function succ* (in italics) and the *label* `succ` (in typewriter font). Maybe we could even go further and eliminate the $1 \to \mathit{nat}$ because we already know that $1 \to \tau \cong \tau$, in which case we would obtain

$$\begin{array}{lcl}
\mathit{zero} & : & \mathit{nat} \\
\mathit{zero} & = & \mathtt{zero} \cdot \langle\,\rangle
\end{array}$$

Finally, it would be nice if we could simplify pattern matching as well. Instead of, for example,

$$\begin{array}{lcl} \mathit{pred} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{pred} & = & \lambda n.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathtt{zero} \cdot \_ \Rightarrow \mathit{zero} \mid \mathtt{succ} \cdot n' \Rightarrow n') \end{array}$$

it would be easier to read and understand if we could write

$$\begin{array}{lcl} \mathit{pred} & : & \mathit{nat} \to \mathit{nat} \\ \\ \mathit{pred}\ \mathit{zero} & = & \mathit{zero} \\ \mathit{pred}\ (\mathit{succ}\ n') & = & n' \end{array}$$

This would somehow only make sense if "*zero*" was understood not only as a constant of type *nat*, but also that it corresponded to a label `zero` with the same name so we can elaborate it into the case of the internal definition of predecessor shown just before. And similarly for *succ* and `succ`.

In fact, modern functional languages such as Haskell, OCaml, or Standard ML provide syntax for data type definitions that provide essentially the above functionality, and more. In ML we would write:

```
datatype nat = Zero | Succ of nat
fun pred Zero = Zero
  | pred (Succ n') = n'
```

In OCaml it might be

```
type nat = Zero | Succ of nat;;
let pred n = match n with
  | Zero -> Zero
  | Succ n' -> n';;
```

And Haskell:

```
data Nat = Zero | Succ Nat

pred :: Nat -> Nat
pred Zero = Zero
pred (Succ n') = n'
```

The type we gave here for `pred` is optional, but it is often helpful to explicitly state the type of a function. We should also keep in mind that the dynamics of `Zero` and `Succ` is different in Haskell because it is a call-by-need ("lazy") language.

We refer to `Zero` and `Succ` as *data constructors*, which means they are simultaneously functions (or constants in the case of `Zero`) to constructs values of a sum, and labels so we can pattern-match against them.

## 4 Generalizing Sums

Let's recall our language so far:

$$\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \mid \rho\alpha.\,\tau \\
\text{Expressions} & e & ::= & x & \text{(variables)} \\
& & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
& & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\
& & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\
& & \mid & \ell \cdot e \mid r \cdot e \mid \mathsf{case}\ e\ (\ell \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2) & (+) \\
& & \mid & \mathsf{case}\ e\ (\,) & (0) \\
& & \mid & \mathsf{fold}\ e \mid \mathsf{unfold}\ e & (\rho) \\
& & \mid & f \mid \mathsf{fix}\ f.\, e & \text{(recursion)}
\end{array}$$

Except for functions and recursive types, the destructors are of the form $\mathsf{case}\ e\ (\ldots)$. We will now unify these constructs even more, replacing the primitive $\mathsf{unfold}\ e$ by a new one, $\mathsf{case}\ e\ (\mathsf{fold}\ x \Rightarrow e')$. We can then define *Unfold* as a function

$$\begin{array}{lcl}
\mathit{Unfold} & : & \rho\alpha.\,\tau \to [\rho\alpha.\,\tau/\alpha]\tau \\
\mathit{Unfold} & \triangleq & \lambda x.\, \mathsf{case}\ x\ (\mathsf{fold}\ x \Rightarrow x)
\end{array}$$

See Exercise 3 for more on this restructuring of the language.

Streamlining our language a little bit further, we now officially generalize the sum from binary to $n$-ary, allowing labels $i$ to be drawn from a finite index set $I$. The case construct for the sums then has a branch for each $i \in I$. Our previous constructs are a special case, with $\tau_1 + \tau_2 \triangleq \sum_{i \in \{\mathtt{l},\mathtt{r}\}} (i : \tau_i) = (\mathtt{l} : \tau_1) + (\mathtt{r} : \tau_2)$ and $0 \triangleq \sum_{i \in \emptyset} (i : \tau_i)$.

$$\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \sum_{i \in I} (i : \tau_i) \mid \rho\alpha.\,\tau \\
\text{Expressions} & e & ::= & x & \text{(variables)} \\
& & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
& & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\
& & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\
& & \mid & i \cdot e \mid \mathsf{case}\ e\ (i \cdot x \Rightarrow e')_{i \in I} & (\sum) \\
& & \mid & \mathsf{fold}\ e \mid \mathsf{case}\ e\ (\mathsf{fold}\ x \Rightarrow e') & (\rho) \\
& & \mid & f \mid \mathsf{fix}\ f.\, e & \text{(recursion)}
\end{array}$$

Except for functions, all destructors are now case-expressions. Functions are different because values are of the form $\lambda x.\, e$ that we cannot match against because we assumed that they are not observable outcomes of computation.

For sums, we have the following generalized statics and dynamics. Key is that we have to check all branches of a case expressions, and all of them have the same type $\tau'$.

$$\frac{k \in I \quad \Gamma \vdash e : \tau_k}{\Gamma \vdash k \cdot e : \sum_{i \in I}(i : \tau_i)}\ \mathsf{sum}$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e'_i : \tau' \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} : \tau'}\ \mathsf{sum/case}$$

$$\frac{e\ \mathit{val}}{i \cdot e\ \mathit{val}}\ \mathsf{val/sum}$$

$$\frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'}\ \mathsf{step/sum}$$

$$\frac{e_0 \mapsto e'_0}{\mathsf{case}\ e_0\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto \mathsf{case}\ e'_0\ (i \cdot x_i \Rightarrow e'_i)_{i \in I}}\ \mathsf{step/sum/case}_0$$

$$\frac{k \in I \quad v_k\ \mathit{val}}{\mathsf{case}\ (k \cdot v_k)\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto [v_k/x_k]e'_k}\ \mathsf{step/sum/case}$$

## 5 Nesting Case Expressions

As another example, let's consider a function *half* on natural numbers that is supposed to round down. We write it down in a pattern-matching style.

$$\begin{array}{lcl} \mathit{half} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{half}\ \mathit{zero} & = & \mathit{zero} \\ \mathit{half}\ (\mathit{succ}\ \mathit{zero}) & = & \mathit{zero} \\ \mathit{half}\ (\mathit{succ}\ (\mathit{succ}\ n'')) & = & \mathit{succ}\ (\mathit{half}\ n'') \end{array}$$

This could be elaborated into two nested case expressions and a use of recursion. To avoid an even deeper nesting of cases, we use *Unfold* as defined in the previous section.

$$\begin{array}{l} \mathit{half} = \mathsf{fix}\ h.\ \lambda n.\ \mathsf{case}\ (\mathit{Unfold}\ n)\ (\mathtt{zero} \cdot \_ \Rightarrow \mathit{zero} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathtt{succ} \cdot n' \Rightarrow \mathsf{case}\ (\mathit{Unfold}\ n')\ (\mathtt{zero} \cdot \_ \Rightarrow \mathit{zero} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathtt{succ} \cdot n'' \Rightarrow \mathit{succ}\ (h\ n''))) \end{array}$$

## 6 Example: Binary Numbers

This kind of elaboration hinted at in the previous section becomes quite tedious and difficult to imagine when the patterns over which a functions is defined become more complex. As an example, we introduce *binary numbers*. As previously suggested, they could be implemented as lists of booleans (*list* $(1+1)$), but we'd like to avoid using lists for now because they depend on a type parameter. Instead, we define directly:

$$\begin{array}{lcl}
\mathit{bin} & \cong & (\mathtt{E} : 1) + (\mathtt{B0} : \mathit{bin}) + (\mathtt{B1} : \mathit{bin}) \\[1ex]
E & : & \mathit{bin} \\
E & = & \mathsf{fold}\ (\mathtt{E} \cdot \langle\,\rangle) \\[1ex]
B0 & : & \mathit{bin} \to \mathit{bin} \\
B0 & = & \lambda x.\, \mathsf{fold}\ (\mathtt{B0} \cdot x) \\[1ex]
B1 & : & \mathit{bin} \to \mathit{bin} \\
B1 & = & \lambda x.\, \mathsf{fold}\ (\mathtt{B1} \cdot x)
\end{array}$$

The interpretation of bit strings as numbers in binary form is given by

$$\begin{array}{lcl}
\llcorner E \lrcorner & = & 0 \\
\llcorner B0\ x \lrcorner & = & 2\,\llcorner x \lrcorner + 0 \\
\llcorner B1\ x \lrcorner & = & 2\,\llcorner x \lrcorner + 1
\end{array}$$

which means that the least significant bit of the number in binary form comes first in the representation. For example,

$$6 = (110)_2 = \llcorner B0\ (B1\ (B1\ E)) \lrcorner$$

It also means that the representation of a number is not unique, because leading zeros do not change its value. For example, $\llcorner B0\ E \lrcorner = \llcorner E \lrcorner = 0$. We will return to this issue in a later lecture.

As a warm-up exercise, let's define an increment function on this representation, using pattern matching. Our specification is $\llcorner \mathit{inc}\ v \lrcorner = \llcorner v \lrcorner + 1$ for any value $v : \mathit{bin}$.

$$\begin{array}{lcl}
\mathit{inc} & : & \mathit{bin} \to \mathit{bin} \\
\mathit{inc}\ E & = & B1\ E \\
\mathit{inc}\ (B0\ x) & = & B1\ x \\
\mathit{inc}\ (B1\ x) & = & B0\ (\mathit{inc}\ x)
\end{array}$$

The recursive call to *inc* in the last case represents the carry bit. It is straightforward to imagine how this function could be elaborated into the current language primitives.

Next, let's define equality on binary numbers. It would be tempting to write

$$\begin{array}{lcl} eq & : & (bin \times bin) \to bool \\ eq\ \langle x, x\rangle & = & true \\ eq\ \_ & = & false \end{array}$$

However, pattern matching does not allow repeated variables in patterns. The principal reason is that this means the complexity of matching even against a single pattern is $O(n)$, where $n$ is the size of the data structure matching against. A second reason is that the type of the variable we are matching against is, say, a function type, we cannot even determine equality. And finally, even types for which their structural equality could be defined, it is often the wrong kind of equality. This is actually the case here: because leading zeros do not change the represented value, our equality function is trickier than we might expect. Here is one way to write it:

$$\begin{array}{lcl} eq & : & (bin \times bin) \to bool \\ eq\ \langle E, E\rangle & = & true \\ eq\ \langle B0\ x, B0\ y\rangle & = & eq\ \langle x, y\rangle \\ eq\ \langle B1\ x, B1\ y\rangle & = & eq\ \langle x, y\rangle \\ eq\ \langle B0\ x, E\rangle & = & eq\ \langle x, E\rangle \\ eq\ \langle E, B0\ y\rangle & = & eq\ \langle E, y\rangle \\ eq\ \_ & = & false \end{array}$$

Elaborating this kind of pattern match into our explicit internal forms is quite tricky. For example, the last catch-all pattern _ (which stands for any variable $p$) must indeed come last. If we had put it first, the pattern match would have been incorrect because it would always return false. So even before we could write complicated elaboration rules, general pattern matching is complicated enough that we should design a formal statics and dynamics.

## 7 General Pattern Matching

We now unify all the different case expressions into a single one. For this, we need two new categories of syntax: *branches* $B$ and patterns $p$. Patterns are either variables, or value constructors for one of types (omitting only functions).

| | | | |
|---|---|---|---|
| Expressions | $e$ | $::=$ | $x \mid \langle e_1, e_2\rangle \mid \langle\,\rangle \mid i \cdot e \mid \mathsf{fold}\ e \mid f \mid \mathsf{fix}\ f.\, e \mid \mathsf{case}\ e\ B$ |
| Patterns | $p$ | $::=$ | $x \mid \langle p_1, p_2\rangle \mid \langle\,\rangle \mid i \cdot p \mid \mathsf{fold}\ p$ |
| Branches | $B$ | $::=$ | $\cdot \mid (p \Rightarrow e \mid B)$ |

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.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \rhd B : \sigma}{\Gamma \vdash \mathsf{case}\ e\ B : \sigma}\ \mathsf{case}$$

The new judgment here is

$$\Gamma \vdash \tau \rhd B : \sigma$$

We read this as

> *Match a case subject of type $\tau$ against the branches $B$, each of which must have type $\sigma$.*

The reason all branches must have the same type is the same as for the conditional or branching over a sum: we don't know which branch will be taken when the programs runs. Each pattern in $B$ should match the type $\tau$. Because there are two alternatives for branches in the syntax, we have two typing rules for branches.

$$\frac{}{\Gamma \vdash \tau \rhd \cdot : \sigma}\ \mathsf{branch/none} \qquad \frac{\Gamma\ ;\ (p : \tau) \vdash e : \sigma \quad \Gamma \vdash \tau \rhd B : \sigma}{\Gamma \vdash \tau \rhd (p \Rightarrow e \mid B) : \sigma}\ \mathsf{branch/alt}$$

We see that to check a single branch $p \Rightarrow e$ we also need a new judgment. We want to check that $p : \tau$, but $p$ contains variables that may occur in $e$ so we also want to create these assumptions. We do this with a new judgment

$$\Gamma\ ;\ \Phi \vdash e : \sigma$$

where $\Phi$ consists of a sequence of assumptions about patterns

$$\Phi ::= \cdot \mid (p : \tau)\ \Phi$$

In the rule above, we always start it as a singleton, but it becomes more complicated as we build its derivation. The simplest case is that for variables, which we just move to $\Gamma$. And if the pattern context is empty, we revert back to the usual typing judgment because we have successfully extracted a type for all the variables in the original pattern $p$.

$$\frac{x : \_ \notin \Gamma \quad \Gamma, x : \tau \; ; \Phi \vdash e : \sigma}{\Gamma \; ; (x : \tau) \; \Phi \vdash e : \sigma}\,\textsf{pat/var} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \; ; \cdot \vdash e : \sigma}\,\textsf{pat/none}$$

In the other cases we do two things: we check that the type matches the pattern at the outer level, and also decompose it to check all component patterns.

$$\frac{\Gamma \; ; (p_1 : \tau_1) \; (p_2 : \tau_2) \; \Phi \vdash e : \sigma}{\Gamma \; ; (\langle p_1, p_2 \rangle : \tau_1 \times \tau_2) \; \Phi \vdash e : \sigma}\,\textsf{pat/pair} \qquad \frac{\Gamma \; ; \Phi \vdash e : \sigma}{\Gamma \; ; (\langle\,\rangle : 1) \; \Phi \vdash e : \sigma}\,\textsf{pat/unit}$$

$$\frac{k \in I \quad \Gamma \; ; (p : \tau_k) \; \Phi \vdash e : \sigma}{\Gamma \; ; (k \cdot p : \sum_{i \in I}(i : \tau_i)) \; \Phi \vdash e : \sigma}\,\textsf{pat/sum} \qquad \frac{\Gamma \; ; (p : [\rho\alpha.\, \tau/\alpha]\tau) \; \Phi \vdash e : \sigma}{\Gamma \; ; (\textsf{fold}\; p : \rho\alpha.\, \tau) \; \Phi \vdash e : \sigma}\,\textsf{pat/fold}$$

We have written $\Phi$ as an ordered sequence rather than a set so that the next step in breaking down $\Phi$ in the judgment $\Gamma \; ; \; \Phi \vdash e : \sigma$ is uniquely determined.

Are these new rules *syntax-directed*? They are! There is one rule for case expression, then one for the empty branches and one to check the first branch. In the $\Gamma \; ; \Phi \vdash e : \sigma$ judgment there is exactly one rule for each form of $\Phi$ for each nonempty $\Phi$ exactly one rule for each kind of pattern.

As an example, let's return to the equality function on binary numbers.

$$\begin{array}{lcl}
\mathit{eq} & : & (\mathit{bin} \times \mathit{bin}) \to \mathit{bool} \\[1ex]
\mathit{eq}\; \langle E, E \rangle & = & \mathit{true} \\
\mathit{eq}\; \langle B0\; x, B0\; y \rangle & = & \mathit{eq}\; \langle x, y \rangle \\
\mathit{eq}\; \langle B1\; x, B1\; y \rangle & = & \mathit{eq}\; \langle x, y \rangle \\
\mathit{eq}\; \langle B0\; x, E \rangle & = & \mathit{eq}\; \langle x, E \rangle \\
\mathit{eq}\; \langle E, B0\; y \rangle & = & \mathit{eq}\; \langle E, y \rangle \\
\mathit{eq}\; \_ & = & \mathit{false}
\end{array}$$

This is elaborated into something like

$$\begin{array}{ll}
\mathit{eq} = \textsf{fix} f.\, \lambda p.\, \textsf{case}\; p\; ( & \langle \textsf{fold}\; (\texttt{E} \cdot \langle\,\rangle), \textsf{fold}\; (\texttt{E} \cdot \langle\,\rangle) \rangle \Rightarrow \texttt{true} \cdot \langle\,\rangle \\
& \mid \langle \textsf{fold}\; (\texttt{B0} \cdot x), \textsf{fold}\; (\texttt{B0} \cdot y) \rangle \Rightarrow f\; \langle x, y \rangle \\
& \mid \ldots)
\end{array}$$

We typecheck the case with

$$\frac{\Gamma_0 \vdash p : \mathit{bin} \times \mathit{bin} \quad \Gamma_0 \vdash \mathit{bin} \times \mathit{bin} \rhd B_0 : \mathit{bool}}{\Gamma_0 \vdash \textsf{case}\; p\; B_0 : \tau'}\,\textsf{case}$$

where

$$\begin{array}{rcl} \Gamma_0 & = & (f : (\mathit{bin} \times \mathit{bin}) \to \mathit{bool}, p : \mathit{bin} \times \mathit{bin}) \\ B_0 & = & (\,\langle \mathsf{fold}\ (\mathtt{E} \cdot \langle\,\rangle), \mathsf{fold}\ (\mathtt{E} \cdot \langle\,\rangle)\rangle \Rightarrow \mathtt{true} \cdot \langle\,\rangle \\ & & \mid \langle \mathsf{fold}\ (\mathtt{B0} \cdot x), \mathsf{fold}\ (\mathtt{B0} \cdot y)\rangle \Rightarrow f\ \langle x, y\rangle \\ & & \mid \ldots) \end{array}$$

Let's look at how to type-check the second branch of $B_0$. We have to check

$$\Gamma_0\ ;\ (\langle \mathsf{fold}\ (\mathtt{B0} \cdot x), \mathsf{fold}\ (\mathtt{B0} \cdot y)\rangle : \mathit{bin} \times \mathit{bin}) \overset{?}{\vdash} f\ \langle x, y\rangle : \mathit{bool}$$

If we abbreviate

$$\sigma_0 \quad = \quad (\mathtt{E} : 1) + (\mathtt{B0} : \mathit{bin}) + (\mathtt{B1} : \mathit{bin})$$

we go through the following inferences

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_0, x : \mathit{bin}, y : \mathit{bin} \vdash f\ \langle x, y\rangle : \mathit{bool}}{\Gamma_0, x : \mathit{bin}, y : \mathit{bin}\ ;\ \color{red}{\cdot} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/none}}{\Gamma_0, x : \mathit{bin}\ ;\ \color{red}{(y : \mathit{bin})} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/var}}{\Gamma_0, x : \mathit{bin}\ ;\ \color{red}{(\mathtt{B0} \cdot y : \sigma_0)} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/sum}}{\Gamma_0, x : \mathit{bin}\ ;\ \color{red}{(\mathsf{fold}\ (\mathtt{B0} \cdot y) : \mathit{bin})} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/fold}}{\Gamma_0\ ;\ \color{red}{(x : \mathit{bin})}\ \color{blue}{(\mathsf{fold}\ (\mathtt{B0} \cdot y) : \mathit{bin})} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/var}}{\Gamma_0\ ;\ \color{red}{(\mathtt{B0} \cdot x : \sigma_0)}\ \color{blue}{(\mathsf{fold}\ (\mathtt{B0} \cdot y) : \mathit{bin})} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/sum}}{\Gamma_0\ ;\ \color{red}{(\mathsf{fold}\ (\mathtt{B0} \cdot x) : \mathit{bin})}\ \color{blue}{(\mathsf{fold}\ (\mathtt{B0} \cdot y) : \mathit{bin})} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/fold}}{\Gamma_0\ ;\ \color{red}{(\langle \mathsf{fold}\ (\mathtt{B0} \cdot x), \mathsf{fold}\ (\mathtt{B0} \cdot y)\rangle : \mathit{bin} \times \mathit{bin})} \vdash f\ \langle x, y\rangle : \mathit{bool}}\ \mathsf{pat/pair}$$

We have highlighted the active part of the pattern context $\Phi$ (where the inference rule is applied) in red and the rest of the pattern context in blue. At the top we arrive back at the usual typing judgment. You should convince yourself that this correctly extracts the assumptions $x : \mathit{bin}$ and $y : \mathit{bin}$ from the pattern.

## 8 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 actually match a value $v$ against a pattern $p$, this match either has to fail or return to us a substitution

$\eta$ for all the variables in $p$. We write this as either $v = [\eta]p$ or "*there is no $\eta$ with $v = [\eta]p$*". This $\eta$ 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 MatchException, which is a new possible outcome of a computation.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ B \mapsto \mathsf{case}\ e_0'\ B}\ \mathsf{step/case_0}$$

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

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

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

If we allow MatchException to have every possible type, then the preservation theorem still goes through. But since MatchException cannot be a value, 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 "_ $\Rightarrow$ 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** It is often intuitive to define types in a mutually recursive way. As a simple example, consider how to define binary numbers in *standard* *form*, that is, not allowing leading zeros. We define binary numbers in standard form (*std*) mutually recursively with strictly positive binary numbers (*pos*).

$$\begin{array}{lcl} \mathit{std} & \cong & (\mathtt{E} : 1) + (\mathtt{B0} : \mathit{pos}) + (\mathtt{B1} : \mathit{std}) \\ \mathit{pos} & \cong & \phantom{(\mathtt{E} : 1) + {}} (\mathtt{B0} : \mathit{pos}) + (\mathtt{B1} : \mathit{std}) \end{array}$$

1. Using only *std*, *pos*, and function types formed from them, give all types of *E*, *B0*, and *B1* defined as follows:

$$\begin{array}{lcl} E & = & \mathsf{fold}\ (\mathtt{E} \cdot \langle\,\rangle) \\ B0 & = & \lambda x.\, \mathsf{fold}\ (\mathtt{B0} \cdot x) \\ B1 & = & \lambda x.\, \mathsf{fold}\ (\mathtt{B1} \cdot x) \end{array}$$

2. Define the types *std* and *pos* explicitly in our language using the $\rho$ type former so that the isomorphisms stated above hold.

3. Does the function *inc* from Section 6 have type $\mathit{std} \to \mathit{pos}$? Rewrite it in the syntax from Section 4, where you may use the function *Unfold* (defined in that section and also in Exercise 3). Then either explain where the typing fails or indicate that it has that type. You do not need to write out a typing derivation.

4. Write a function $\mathit{pred} : \mathit{pos} \to \mathit{std}$ that returns the predecessor of a strictly positive binary number. You may use pattern matching to define your function, but you must make sure it is correctly typed.

**Exercise 2** It is often convenient to define functions by mutual recursion. As a simple example, consider the following two functions on bit strings determining if it has *even or odd parity*.

$$\begin{array}{lcl} \mathit{bin} & \cong & (\mathtt{E} : 1) + (\mathtt{B0} : \mathit{bin}) + (\mathtt{B1} : \mathit{bin}) \\[4pt] \mathit{even} & : & \mathit{bin} \to \mathit{bool} \\ \mathit{odd} & : & \mathit{bin} \to \mathit{bool} \\[4pt] \mathit{even}\ E & = & \mathit{true} \\ \mathit{even}\ (B0\ x) & = & \mathit{even}\ x \\ \mathit{even}\ (B1\ x) & = & \mathit{odd}\ x \\[4pt] \mathit{odd}\ E & = & \mathit{false} \\ \mathit{odd}\ (B0\ x) & = & \mathit{odd}\ x \\ \mathit{odd}\ (B1\ x) & = & \mathit{even}\ x \end{array}$$

1. Write a function *parity* with a single fixed point constructor and use it to define *even* and *odd*. You may use pattern matching, but the pattern of recursion (and the fact you only need one fixed point) should be clear. Also, state the type of your *parity* function explicitly.

2. More generally, our simple recipe for implementing a recursively specified function using the fixed point constructor in our call-by-value language goes from the specification

$$\begin{array}{lcl} f & : & \tau_1 \to \tau_2 \\ f\,x & = & h\,f\,x \end{array}$$

   to the implementation

$$f \;=\; \mathsf{fix}\,g.\,\lambda x.\,h\,g\,x$$

   It is easy to misread these, so remember that by our syntactic convention, $h\,f\,x$ stands for $(h\,f)\,x$ and similarly for $h\,g\,x$. Give the type of $h$ and show by calculation that $f$ satisfies the given specification by considering $f\,v$ for an arbitrary value $v$ of type $\tau_1$.

3. A more general, mutually recursive specification would be

$$\begin{array}{lcl} f & : & \tau_1 \to \tau_2 \\ g & : & \sigma_1 \to \sigma_2 \\ f\,x & = & h_1\,f\,g\,x \\ g\,y & = & h_2\,f\,g\,y \end{array}$$

   Give the types of $h_1$ and $h_2$.

4. Show how to explicitly define $f$ and $g$ in our language from $h_1$ and $h_2$ using the fixed point constructor and verify its correctness by calculation as in part 2. You may use any other types in the language introduced so far (pairs, unit, sums, and recursive types).

**Exercise 3** In the language where the primitive unfold has been replaced by pattern matching, we can define the following two functions:

$$\begin{array}{lcl} \mathit{Unfold} & : & \rho\alpha.\,\tau \to [\rho\alpha.\,\tau/\alpha]\tau \\ \mathit{Unfold} & = & \lambda x.\,\mathsf{case}\;x\;(\mathsf{fold}\;x \Rightarrow x) \\[1ex] \mathit{Fold} & : & [\rho\alpha.\,\tau/\alpha]\tau \to \rho\alpha.\,\tau \\ \mathit{Fold} & = & \lambda x.\,\mathsf{fold}\;x \end{array}$$

Prove that *Fold* and *Unfold* are witnessing a type isomorphism.

**Exercise 4** Design a collection of inference rules for bidirectional type-checking in the language with general pattern matching. As before, we expect *checking* to either succeed or fail and *synthesis* to either return a unique type or fail. One key question will be how to interpret the new judgments $\Gamma \vdash \tau \triangleright B : \sigma$ and $\Gamma \,;\, \Phi \vdash e : \sigma$ or, if that is not possible, how to restructure them.

For simplicity, you only need to write out the rules for functions $\tau_1 \to \tau_2$ and general sums $\sum_{i \in I}(i : \tau_i)$, omitting products, unit, and recursive types.