Lecture 9: Product and Sum Types

Chris Martens

January 30, 2026

1 Lecture outline

- Continuing: STLC
  - Review Typing Rules
  - Checking, Synthesis, and Type Annotations
- Product and Sum Types
  - Example programs
  - Typing rules
  - Dynamic semantics

2 Continuing: STLC

$$\frac{\Gamma, x{:}\tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\ e : \tau \to \sigma}\ \mathsf{Ty}/\lambda \qquad \frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash (e_1\ e_2) : \sigma}\ \mathsf{Ty}/\mathsf{app} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{Ty}/\mathsf{var}$$

Exercises:

1. $\lambda x.x : \mathsf{Num} \to [????]$
2. $(\lambda x.\lambda y.x)\ 10 : [????] \to \mathsf{Bool} \to [????]$
3. $(\lambda f.\lambda x.\ f\ x)\ (\lambda x.10) : [????]$

Note that some expressions like $\lambda x.x$ don't have a unique type. So we can't strictly do synthesis with this system. However, if we try to do checking, notice what happens with the app rule.

We will resolve this issue by adding an *annotation* on the argument of lambdas:

$$\frac{\Gamma, x{:}\tau \vdash e : \sigma}{\Gamma \vdash \lambda x : \tau.\ e : \tau \to \sigma}\ \mathsf{Ty}/\lambda$$

This annotation will allow us to synthesize unique types for expressions. (There is another approach called *bidirectional typing* that we may cover later in class.)

3 Product and Sum Types: Motivation

In this lecture, we'll introduce types that allow us to describe the *shape of data*. For example, let's say we're implementing a card game, and we want to describe the data represented by a card in a standard deck: it has a suit ($\heartsuit, \clubsuit, \diamondsuit, \spadesuit$) and a rank (1-13). At its essence, we can think of this as a *pair of alternatives*, the first of which has four alternatives and the second of which has thirteen (although for now we will use our built in Number type).

In a pragmatic concrete syntax, we'd have labels for things, like this:

```
type Suit = Heart | Club | Diamond | Spade
type Card = {suit : Suit, rank : Number}
```

And we could write a function like this to get the color of a card:

```
type Color = Red | Black

get_color : Card -> Color
= lambda c .
    case (c.suit) of
      Heart => Red
    | Diamond => Red
    | Spade => Black
    | Club => Black
```

In our abstract syntax, which we can think of as a "compilation target" for this kind of data format, it will look something like this:

$$\mathsf{Suit} \triangleq \mathsf{Unit} + \mathsf{Unit} + \mathsf{Unit} + \mathsf{Unit}$$

$$\mathsf{Color} \triangleq \mathsf{Unit} + \mathsf{Unit}$$

$$\mathsf{Card} \triangleq \mathsf{Suit} \times \mathsf{Number}$$

$$\mathsf{get\_color} = \lambda c.\ \mathsf{case}(\mathsf{proj}_1 c, x.\mathsf{tag}_1(), [\text{to fill in}])$$

3.1 Syntax

Let's consider the new class of values we're introducing with these data formers. We have *pairs* of values, and *tagged alternatives*. We thus add two new *type formers* (also known as *connectives*) to our language, the product $\times$ for the type corresponding to pair values, and sums $+$ corresponding to alternative values, also known as a *disjoint union* of two sets of values.

3.2 Typing Rules

We now *extend* STLC with product and sum types. From now on, we will move from thinking of expressions first and types secondary to thinking of types as coming first, with expressions representing their *introduction* and *elimination* forms: in other words, how do we *construct* an element of the type, and how do we *use* one?

Products:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \times I \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1} \times E_1 \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2} \times E_2$$

Sums:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{tag}_1(e) : \tau_1 + \tau_2} + I_1 \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{tag}_2(e) : \tau_1 + \tau_2} + I_2$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \sigma \quad \Gamma, x : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{case}(e, x.e_1, y.e_2) : \sigma} + E$$

One more thing: the unit type!

$$\frac{}{() : \mathsf{Unit}} \ \mathsf{Unit}/I \qquad \text{(no Unit E)}$$

Q: Where do we need to add type annotations in order to be able to synthesize types for this extended language?

4 Dynamic Semantics

Let's collect our syntax of expressions:

$e ::= () \mid (e_1, e_2) \mid e.1 \mid e.2 \mid \mathsf{tag}_1\ e \mid \mathsf{tag}_2\ e \mid \mathsf{case}(e, x.e_1, y.e_2) \mid \lambda x : \tau.e \mid (e_1\ e_2)$

Values: $()$, pairs of values (v_1, v_2) (assuming eager pairs), and tagged values $\mathsf{tag}_i\ v$.

For pairs we have two choices, eager and lazy. Here's the eager version:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \Downarrow \mathsf{pair} \qquad \frac{e \Downarrow (v_1, v_2)}{e.1 \Downarrow v_1} \Downarrow \mathsf{proj}_1 \qquad \frac{e \Downarrow (v_1, v_2)}{e.2 \Downarrow v_2} \Downarrow \mathsf{proj}_2$$

Sums:

$$\frac{e \Downarrow v}{\mathsf{tag}_1\ e \Downarrow \mathsf{tag}_1\ v} \Downarrow \mathsf{tag}_1 \qquad \frac{e \Downarrow v}{\mathsf{tag}_2\ e \Downarrow \mathsf{tag}_1\ v} \Downarrow \mathsf{tag}_2$$

$$\frac{e \Downarrow \mathsf{tag}_i\ v \quad [v/x]e_i \Downarrow v'}{\mathsf{case}(e, x.e_1, x.e_2) \Downarrow v'} \Downarrow \mathsf{case}$$

5 Compilation

5.1 Type encodings

- From binary to multi-ary sums and products
- Bool and if
- Colors, suits, and cards