Last time we saw Simply Typed λ-Calculus (STLC), but noted that the types not stopped us from writing loops, but also hindered our ability to encode much simpler constructs like numbers, and pairs. In this lecture we will recover those by adding them to the language as primitives, and investigate the semantics and typing rules of these new constructs.

1 Recap: STLC

Recall that STLC has not only expressions and values, but also *types*. The three are defined by the following BNF grammar

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \mathsf{unit} \mid \tau_1 \to \tau_2 \\ \text{Terms} & e & ::= & () \mid x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \\ \text{Values} & v & ::= & () \mid \lambda x{:}\tau.\, e \text{ closed} \end{array}$$

We also saw typing rules that allowed us to prove the judgment $\Gamma \vdash e : \tau$, meaning that we can prove e has type τ in context Γ. The rules were as follows.

[UNIT] $\dfrac{}{\Gamma \vdash () : \mathsf{unit}}$ [VAR] $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$ [ABS] $\dfrac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}$ [APP] $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$

This type system guaranteed *type soundness*, the requirement that well-typed programs not get stuck, but it restricted us substantially. For instance, our encoding of numbers as Church numerals no longer worked the way we wanted. Instead, we will see how to add various other types to the language as primitives.

2 Integers

We have seen these before as a primitive language feature in FL, but that language did not have types. To add integers to our language, we extend the syntax of both types, terms, and values. As previously, we will use $\otimes$ to denote arithmetic operations (+, −, ∗, ^).

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \mathsf{int} \\ e & ::= & \cdots \mid n \mid e_1 \otimes e_2 \\ v & ::= & \cdots \mid n \end{array}$$

Like in FL, we need to define the semantics for these new operations. For integers, we have seen how to do this before and we can just do it again.

$$E ::= \cdots \mid E \otimes e \mid v \otimes E \qquad \frac{m = n_1 \otimes n_2 \text{ (mathematically)}}{n_1 \otimes n_2 \longrightarrow m}$$

Unlike in FL, we also need to extend the type system with rules for these new terms. We need an axiom saying a literal n has type int in every environment, and an inductive rule to handle arithmetic expressions.

[INT] $\dfrac{}{\Gamma \vdash n : \mathsf{int}}$ [ARITH] $\dfrac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \otimes e_2 : \mathsf{int}}$

While we no longer have Church numerals that behave the way we want, we now have regular arithmetic expressions and can use them accordingly. We do not have the full power of Church numerals, which allowed us to define bounded for loops, but we have gotten a substantial feature back.

3 Pairs

Another useful data structure that we previously encoded as *pairs*. Again, we need to extend the grammars of types, terms, and values. The type of a pair where the first element has type τ_1 and the second has τ_2 is $\tau_1 \times \tau_2$. These are often called *product types*. For terms, we include a pairing operator and a projection operator, and only pairs of values are themselves values.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \tau_1 \times \tau_2 \\ e & ::= & \cdots \mid (e_1, e_2) \mid \mathsf{proj}_1\ e \mid \mathsf{proj}_2\ e \\ v & ::= & \cdots \mid (v_1, v_2) \end{array}$$

Notice that for every added syntactic form, we have have an expression to *introduce* the form that we will write in blue (here (e_1, e_2)), and a separate expression to *eliminate* the form that we will write in red (here $\mathsf{proj}_1\ e$ and $\mathsf{proj}_2\ e$). This will be a common theme.

The semantics of pairs mirrors precisely the equational requirements we used to structure an encoding of them in untyped λ-calculus.

$$E ::= \cdots \mid (E, e) \mid (v, E) \mid \mathsf{proj}_i\ E \qquad \frac{i \in \{1,2\}}{\mathsf{proj}_i\ (v_1, v_2) \longrightarrow v_i}$$

Note that the elimination form only proceeds when operating on the introduction form, which it *eliminates*.

Lastly, we again need typing rules for our new pairing expressions.

[Pair]
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

[Proj]
$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Gamma \vdash \mathsf{proj}_i\ e : \tau_i}$$

4 Records

A record type is like an arbitrary-sized tuple, but with names, similar to a C-style struct. Each entry is *labeled* with a name from a countable list of labels **Lab**. The syntax for records is as follows.

$$\begin{array}{rcl} \ell & \in & \mathbf{Lab} \\ \tau & ::= & \cdots \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \\ e & ::= & \cdots \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid e.\ell \\ v & ::= & \cdots \mid \{\ell_1 = v_1, \ldots, \ell_n = v_n\} \end{array}$$

The semantics for the introduction form evaluates the expression in a record left-to-right, and the elimination form simply accesses the relevant value.

$$E ::= \cdots \mid \{\ell_1 = v_1, \ldots, \ell_{i-1} = v_{i-1}, \ell_i = E, \ell_{i+1} = e_{i+1}, \ldots, \ell_n = e_n\} \mid E.\ell$$

$$\frac{1 \le i \le n}{\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.\ell_i \longrightarrow v_i}$$

The typing rule for creating a record require that all expressions match the type specified, and the typing rule for accessing simply pulls out the appropriate type.

[Record]
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

[Access]
$$\frac{\Gamma \vdash e : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \qquad 1 \le i \le n}{\Gamma \vdash e.\ell_i : \tau_i}$$

5 Sums

Sum types are a little more interesting. They are useful when we the collection of data a type represents to include data with multiple forms. For instance, a list can be either empty or non-empty.

The standard way to represent these datatypes is with a *sum type*, denoted $\tau_1 + \tau_2$, meaning data of this type is either a τ_1 or a τ_2. To construct something of type $\tau_1 + \tau_2$, we can take something of type τ_1 (or τ_2) and *inject* it into the space $\tau_1 + \tau_2$ using an appropriate injection operator. The introduction forms for sum types are therefore inl and inr, the *left injection* and *right injection*, respectively.

To eliminate a $\tau_1 + \tau_2$, we need to be able to handle both the case where we have a τ_1 and the case where we have a τ_2, though not necessarily in the same way. To do that, we use a match expression, that takes a value of type $\tau_1 + \tau_2$ and the code to execute in each case, examines whether it has a left (τ_1) value or a right (τ_2) value, and calls the appropriate branch.

The syntax for these operations is as follows. Note that we annotate the introduction forms with types because otherwise we cannot determine what τ_2 should be with inl or τ_1 with inr.

$$
\begin{array}{rcl}
\tau & ::= & \cdots \mid \tau_1 + \tau_2 \\
e & ::= & \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ e \mid \mathsf{inr}_{\tau_1+\tau_2}\ e \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x).\ e_1 \mid \mathsf{inr}(y).\ e_2 \\
v & ::= & \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v
\end{array}
$$

Again, we must provide a semantics for these new terms, which we do by following the intuition described above. Note that, as with pairs, the elimination for (match) proceeds by destructing the introduction forms.

$$
E ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ E \mid \mathsf{inr}_{\tau_1+\tau_2}\ E \mid \mathsf{match}\ E\ \mathsf{with}\ \mathsf{inl}(x).\ e_1 \mid \mathsf{inr}(y).\ e_2
$$

$$
\frac{}{\mathsf{match}\ (\mathsf{inl}_{\tau_1+\tau_2}\ v)\ \mathsf{with}\ \mathsf{inl}(x).\ e_1 \mid \mathsf{inr}(y).\ e_2 \longrightarrow e_1[x \mapsto v]}
$$

$$
\frac{}{\mathsf{match}\ (\mathsf{inr}_{\tau_1+\tau_2}\ v)\ \mathsf{with}\ \mathsf{inl}(x).\ e_1 \mid \mathsf{inr}(y).\ e_2 \longrightarrow e_2[y \mapsto v]}
$$

Note that the evaluation of the branches is lazy; only one will evaluate, and only after we destruct a sum value to determine which one.

Adding typing rules to these terms also requires some thought. The typing rules for inl and inr are fairly simple, as they specify both types and which one must be provided. For match, however, things are slightly more complicated. First we note that the match expression itself binds variable x inside e_1 and y inside e_2. That means that, when type checking those branches, we need to add the appropriate variable to the context. Second, we note that the return type of the whole match cannot depend on whether we took the left or right branch. That means that e_1 and e_2 must both of the same return type as each other, even though their inputs types may be different! We formalize this intuition with the following typing rules.

[Inl]
$$
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2}
$$

[Inr]
$$
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2}
$$

[Match]
$$
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x).\ e_1 \mid \mathsf{inr}(y).\ e_2 : \tau}
$$

To see an example of a sum type, we look to a datatype that has been conspicuously absent to this point: booleans. A boolean is a type with two values, true and false. We already have unit, a type with exactly one value, so we can use that along with sums to construct the type unit + unit, that has exactly two values: inl () and inr (). We can take these as our definitions of true and false and build if statements using match.

In particular, if b then e_1 else e_2 should evaluate e_1 when $b \longrightarrow^* \mathsf{true}$ and e_2 when $b \longrightarrow^* \mathsf{false}$. The match statement above can produce the same behavior with following definitions.

$$\begin{aligned} \mathsf{true} &\triangleq \mathsf{inl}_{\mathsf{unit+unit}}\ () \\ \mathsf{false} &\triangleq \mathsf{inr}_{\mathsf{unit+unit}}\ () \\ \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &\triangleq \mathsf{match}\ b\ \mathsf{with}\ \mathsf{inl}(\_).\, e_1 \mid \mathsf{inr}(\_).\, e_2 \end{aligned}$$

Here _ is used to indicate a dummy variable that is not free in its scope.