CS4400 Lecture 13: Polymorphism (System F)

Feburary 6, 2026

1 Lecture outline

- Polymorphism: motivating examples
- System F

2 Polymorphism: Motivating Examples

We've seen by now a number of situations where we want to describe the types of expressions in a *generic* way, with variables that can range over types. Examples include:

- Mapping a function over a list: $\mathsf{map} : \mathsf{List}\ \alpha \to (\alpha \to \beta) \to \mathsf{List}\ \beta$
- Currying: $\mathsf{curry} : (\alpha \times \beta \to \gamma) \to (\alpha \to \beta \to \gamma)$
- The identity function: $\mathsf{id} : \alpha \to \alpha$

We can write down typing derivations for expressions that leave these types as metavariables, but so far we have not had a way to *internalize* this kind of genericity-over-types into our language. Doing so will introduce *polymorphism*: literally meaning "many forms", we will be able to explicitly characterize values that can be instantiated with multiple types.

Polymorphism shows up in most modern programming languages, sometimes under the term *generics*. It is an important tool for code reuse and library design, helping us reveal the essential structure of our programs without overly specializing them to specific tasks.

3 Main idea: type abstraction

We've previously characterized functions $\lambda x.e$ as *abstractions*: instead of defining a concrete expression e (where its free variable is instantiated with a specific concrete expression), we *abstract over* some parameter. In this setting we say we have "expression abstraction": the ability to abstract over expressions by binding variables that refer to arbitrary expressions.

We will now do the same for *types* by implementing *type abstraction*. An abstract type is written:

$$\forall \alpha : \mathsf{Type}.\tau$$

In practice, all type abstractions will be over type variables, so we will omit the $: \mathsf{Type}$ annotation.

The values of this type can be thought of as *functions that take a type as an argument*, so we write them, by analogy with the λ calculus, as:

$$\Lambda \alpha.e$$

This gives us a way to explicitly represent the type of, e.g., curry in a generic way:

$$\mathsf{curry} : \forall \alpha.\forall \beta.\forall \gamma.(\alpha \times \beta \to \gamma) \to (\alpha \to \beta \to \gamma)$$

When we implement the curry function, we need to explicitly bind the type variables with type-level lambda. We can use the following concrete syntax:

```
curry = TLam a. TLam b. TLam c.
            lam (f : a * b -> c). lam (x : a). lam (y : b). f (x,y)
```

...to represent the following abstract (formal) syntax:

$$\Lambda a.\Lambda b.\Lambda c.\lambda f.\lambda x.\lambda y.\ f\ (x, y)$$

Now let's say we have a specific function we want to curry, such as this function that takes a pair of a number n and a string s, then concatenates s to itself n times:

```
repeat : Number * String -> String
repeat 0 s = ""
repeat n s = s ++ (repeat (n-1) s)
```

We will need to *instantiate* the curry function with the types `Number`, `String`, and `String` in place of its variables α, β, and γ. To insantiate variables, we will "apply" the type-level function:

```
curry [Number] [String] [String]
    : (Number * String -> String) -> (Number -> String -> String)
```

...and now it has the correct type to be applied to the `repeat` function.

4 System F

Starting with STLC, we can add polymorphic types to our language of types, and type-level abstraction and application to our language of expressions.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \ldots \mid \forall\alpha.\tau \\ e & ::= & x \mid \lambda x:\tau.e \mid e_1\ e_2 \mid \ldots \mid \Lambda\alpha.e \mid e[\tau] \end{array}$$

4.1 Dynamics

The dynamics of System F are a straightforward adaptation of the usual lambda calculus rules, so we give them first:

Values:

$$\frac{}{\Lambda\alpha.e\ \mathsf{value}}\ \mathsf{val/tylam}$$

Evaluating type application:

$$\frac{e \Downarrow (\Lambda\alpha.e) \quad [\tau/\alpha]e \Downarrow v}{e[\tau] \Downarrow v}\ \mathsf{eval/tyapp}$$

Note the use of type-level substitution during evaluation.

4.2 Typing Rules

In addition to defining well-typed expressions, we also need to say what it means for a type to be well-formed. Since types can have variables in them we need a notion of scoping for variables.

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha\ \mathsf{type}$$

Well-formed types: $\boxed{\Gamma \vdash \tau\ \mathsf{type}}$

$$\frac{(\alpha\ \mathsf{type}) \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \mathsf{wf/var}_\alpha \qquad \frac{\Gamma, \alpha\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Gamma \vdash \forall\alpha.\tau\ \mathsf{type}}\ \mathsf{wf/all}$$

$$\frac{\Gamma \vdash \tau_1\ \mathsf{type} \quad \Gamma \vdash \tau_2\ \mathsf{type}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathsf{type}}\ \mathsf{wf/arr}$$

Well-formed contexts: $\boxed{\Gamma\ \mathsf{ctx}}$

$$\frac{}{(\cdot)\ \mathsf{ctx}}\ \mathsf{ctx/emp} \qquad \frac{\Gamma\ \mathsf{ctx}}{(\Gamma, \alpha\ \mathsf{type})\ \mathsf{ctx}}\ \mathsf{ctx/tvar} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \tau\ \mathsf{type}}{(\Gamma, x:\tau)\ \mathsf{ctx}}\ \mathsf{ctx/var}$$

Finally we can present the typing rules for expressions: $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma, \alpha\ \mathsf{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \mathsf{ty/tylam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau \quad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash e[\sigma] : [\sigma/\alpha]\tau}\ \mathsf{ty/tyapp}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{ty/var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\ \tau_1.e : \tau_1 \to \tau_2}\ \mathsf{ty/lam} \qquad \frac{\Gamma \vdash f : \tau \to \tau' \quad \Gamma \vdash e : \tau}{\Gamma \vdash f\ e : \tau'}\ \mathsf{ty/app}$$

4.2.1 Examples

We will do the following typing derivations in class.

1. $\Lambda\alpha.\lambda x.x : \forall\alpha.\alpha \to \alpha$
2. $\Lambda\alpha.\Lambda\beta.\lambda x : \alpha \times \beta.\ x.1 : \forall\alpha.\forall\beta.\ \alpha \times \beta \to \alpha$
3. $(\Lambda\alpha.\Lambda\beta.\lambda x : \alpha \times \beta.\ x.1)\ [\mathsf{Unit}] : \forall\beta.\ \mathsf{Unit} \times \beta \to \mathsf{Unit}$

Just to keep things concrete, here is a Java-like concrete syntax for the above examples:

```
1. function id<T>(x : T) : T = { return x; }
2. function proj1<T,S>(x : {fst : T, snd : S}) = { return x.fst; }
3. function proj1U<S> = proj1<T=Unit, S=S>
```

5 Exercises

Fill in the blanks in the following typing judgments so the resulting judgment holds, or indicate there is no way to do so.

1. $\boxed{} \vdash \forall\alpha.\ \alpha \to \beta\ \mathsf{type}$
2. $\boxed{} \vdash f[\alpha] : \alpha \to \beta$
3. $\alpha\ \mathsf{type} \vdash \boxed{} : \forall\beta.\ \alpha \to \beta$
4. $x : \forall\alpha.\ (\forall\beta.\ \beta \to \beta) \to \alpha, \gamma\ \mathsf{type} \vdash \boxed{} : (\gamma \to \gamma) \to \gamma$