Lecture 15: Polymorphism and System F

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 What is Polymorphism?

- It is useful to write functions that work for a variety of types. For example, the identity function $\lambda x.x$ should work for any type of x.
- Type systems that allow a single piece of code to be used with multiple types are broadly referred to as **polymorphic type systems**, or sometimes simply **polymorphism**.[1]
- There are many kinds of polymorphism, and we will explore some of them in this class. A particularly common form of polymorphism permits writing functions that are *parametric in a type*, meaning that the function itself takes a **type argument** (also called a *type parameter*) as part of its definition. This form of polymorphism is called **parametric polymorphism**, and it is what we will study first.
- For example, Java supports a form of parametric polymorphism via the following syntax:

  ```
  public class Main {
    public static <T> T identity(T x) {
      return x;
    }

    public static void main(String[] args) {
      System.out.println(identity("Hello World!"));
    }
  }
  ```

  The syntax `<T>` declares a type argument `T`. The identity function is not allowed to know what `T` is; it must work over all possible instantiations of `T`. The `main` functions calls `identity` with an argument of type `string`; this replaces `T` with `string` in `identity`.
- Parametric polymorphism is essential for *code reuse*: it allows the programmer to avoid unnecessary duplication of code that should work regardless of the argument type.

[1]The word "polymorphism" comes from the Greek words poly , meaning "many", and morph, meaning "shape"

2 Another Form of Polymorphism: Subtyping

- There are other common kinds of polymorphism besides parametric polymorphism. Another form of polymorphism is **subtype polymorphism**, which uses inheritance to achieve polymorphic behavior.
- Here is an example of subtype polymorphism in Java:

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void playSound(Animal a) {
    a.animalSound();
  }

  public static void main(String[] args) {
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    playSound(myPig);
    playSound(myDog);
  }
}
```

- In the above code, there is a root class `Animal` and 2 subclasses (`Pig` and `Dog`). All of these classes share a common method (`animalSound`). The method `playSound` takes an animal as an argument and calls its `playSound` method.
- The `playSound` method exhibits polymorphic behavior since it can take as argument any object of type `Animal`.

3 A Tiny Language for Parametric Polymorphism: System F

- As usual in this class, we will design, implement, and study a small language to better understand the principles of parametric polymorphism. This language will be called **System F**.[2]
- The idea of System F is to extend the simply-typed λ-calculus with the following new ideas:
  - **Type abstraction**. We need a way of designating a function that takes a type as an argument. This is called a type abstraction, and we use the surface syntax $\Lambda X.e$ to denote it (the symbol Λ is a capital lambda). Using this syntax, we can write a polymorphic identity function:

    $$\Lambda X.(\lambda x : X.x) \tag{1}$$

    The identifier X above is a type parameter.
  - **Type application**. Now we need a way to instantiate type parameters. We do this using something that looks a lot like normal function application:

    $$(\Lambda X.(\lambda x : X.x))\ [\mathtt{num}] \tag{2}$$

    We call the above syntax a *type application*: its semantics is to substitute the type parameter X for the argument `num` in the body of the type abstraction. So, 1 evaluates to $(\lambda x : \mathtt{num}.x)$. We wrap type application in square brackets to syntactically distinguish it from term application.
  - **Universal types**. The term 1 is a program waiting to receive a type as an argument. Such a program should itself have a type to make sure we don't misuse it (for instance, by calling it with an argument that is not a type). We will write this type $\forall X.t$ (read "for all X"), where X is a type identifier and t is a type that can refer to X. Now we can write a type for the identity function above:

    $$\Lambda X.(\lambda x : X.x) : \forall X.X \to X \tag{3}$$

    Think of "$\forall X.X \to X$" like a lambda: the argument X is free in the body $X \to X$. Once application occurs, X will be substituted for a concrete type.
- The complete surface syntax for System F with numbers is:

  $$\begin{aligned}\mathtt{t} &::= \mathtt{num} \mid \mathtt{t} \to \mathtt{t} \mid \forall X.t \mid X\\ \mathtt{e} &::= \lambda x : \mathtt{t}.\mathtt{e} \mid \Lambda X.\mathtt{e} \mid (\mathtt{e}\ \mathtt{e}) \mid (\mathtt{e}\ [\mathtt{t}]) \mid \mathtt{number} \mid \mathtt{ident}\end{aligned}$$

- Here is the abstract syntax of System F (using OCaml syntax):

```
type typ =
  | TNum
  | TFun of typ * typ
  | TIdent of string
  | Tforall of string * typ

type expr =
  | Num of int
  | Ident of string
  | Lam of string * typ * expr
  | App of expr * expr
  | Tlam of string * expr
  | Tapp of expr * typ
```

[2]System F was introduced independently by Girard [1972] in the context of logic and Reynolds [1974] in the context of programming languages. Our development in this section is based on Pierce [2002, Chapter 23].

4 Programming in System F

- The semantics of System F can be given in terms of substitution, just like STLC. Let's run some example programs to get a feel for how programming in System F looks.
- Let's use the **step arrow** "$\rightarrow$" to denote one step of simplifying System F programs, and run a few examples.[3]

  Example 4.1. The identity function evaluated on the number 10:

  $$((\Lambda X.\lambda x : X.x)\ [\mathtt{num}])\ 10 \xrightarrow{\text{Substitute } \mathtt{num}} (\lambda x : \mathtt{num}.x)\ 10 \xrightarrow{\text{Substitute } 10} 10$$

  Example 4.2. Call twice:

  $$\begin{aligned}
  &(((\Lambda X.\lambda f : X \rightarrow X.\lambda a : X.f\ (f\ a))\ [\mathtt{num}])\ (\lambda x : \mathtt{num}.x))\ 10\\
  &\xrightarrow{\text{substitute } \mathtt{num}} ((\lambda f : \mathtt{num} \rightarrow \mathtt{num}.\lambda a : \mathtt{num}.f\ (f\ a))\ (\lambda x : \mathtt{num}.x))\ 10\\
  &\xrightarrow{\text{substitute } (\lambda x:\mathtt{num}.x)} ((\lambda a : \mathtt{num}.(\lambda x : \mathtt{num}.x)\ ((\lambda x : \mathtt{num}.x)\ a)))\ 10\\
  &\xrightarrow{\text{substitute } 10} ((\lambda x : \mathtt{num}.x)\ ((\lambda x : \mathtt{num}.x)\ 10))\\
  &\rightarrow (\lambda x : \mathtt{num}.x)\ 10\\
  &\rightarrow 10
  \end{aligned}$$

- Pause and reflect: how is the semantics of System F reflecting how parametric polymorphism works in Java?
- Let's take a moment to examine the consequences of parametric polymorphism. Write down as many programs as you can of type $\forall X.X \rightarrow X$. What do you notice?
- You should notice that *they all behave like the identity function*!
- Why is this the case? Intuitively, all programs of type $\forall X.X \rightarrow X$ must (1) consume some unknown type X, and (2) produce an **inhabitant** of some unknown type X.[4] Since this function can't know anything at all about X – including how to construct values of type X – all it can do is return the argument to the caller.
- Can you write down a program of type $\Lambda X.\Lambda Y.X \rightarrow Y$?

[3]The step arrow is sometimes also called a "small step semantics", to contrast it with our usual big-step semantics we have seen so far in class.

[4]A inhabitant of a type is a value of that type.

5 Typechecking System F

- The rules for typechecking System F are fairly simple extensions of typechecking STLC.
- We need to extend our type context Γ to account for the introduction of type variables. Now we will have two ways of extending Γ: (1) $\Gamma \cup \{x : \texttt{t}\}$ which extends Γ by associating x with type `t`, and (2) $\Gamma \cup X$ which extends Γ with a new type variable X.
- A type is **well-formed in context** Γ if it never refers to a type variable that is not in Γ. We will need to check that all types are well-formed in our typechecker.
- Here are the rules We include them in here in full:

$$\frac{}{\Gamma \vdash \texttt{(enum n)} : \texttt{num}}\ \text{(T-NUM)} \qquad \frac{x \in \Gamma \qquad \Gamma(x) = t}{\Gamma \vdash x : t}\ \text{(T-IDENT)}$$

$$\frac{\Gamma \cup \{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1 . e) : t_1 \to t_2}\ \text{(T-LAM)} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}\ \text{(T-APP)}$$

$$\frac{\Gamma \cup X \vdash \texttt{e} : \texttt{t}}{\Gamma \vdash \Lambda X.\texttt{e} : \forall X.\texttt{t}}\ \text{(T-TLAM)} \qquad \frac{\Gamma \vdash \texttt{e}_1 : \forall X.\texttt{t}_1}{\texttt{e1}\ [\texttt{t}_2] : \texttt{t}_1[X \mapsto \texttt{e}_2]}\ \text{(T-TAPP)}$$

Figure 1: Type system for System F.

- There is a subtlety in these rules: the T-APP rule implicitly requires comparing two types for equality. However, it is not trivial to check whether or not two types are equal in System F. Consider the following two terms, annotated with their types:

$$\Lambda X.\lambda x : X.x : \forall X.X \to X \qquad \Lambda Y.\lambda x : Y.x : \forall Y.Y \to Y$$

  Intuitively, these two terms should have the same type, since they differ only by their naming of the type parameter (in other words, they are *α-equivalent types*).
- As a consequence, when implementing our typechecker, we will need to check type equality up to α-equivalence.

6 Typing Derivations for System F

- We will do a few examples in class, but here is one example for your reference:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{X, x : X\} \vdash x : X}\;(\text{T-V}\small{\text{AR}})}{\{X\} \vdash \lambda x : X.x : X \to X}\;(\text{T-L}\small{\text{AM}})}{\{\} \vdash (\Lambda X.\lambda x : X.x) : \forall X.X \to X}\;(\text{T-TL}\small{\text{AM}})}{\{\} \vdash ((\Lambda X.\lambda x : X.x)\ [\texttt{num}]) : \texttt{num} \to \texttt{num}}\;(\text{T-T}\small{\text{APP}}) \qquad \{\} \vdash 10 : \texttt{num}}{\{\} \vdash ((\Lambda X.\lambda x : X.x)\ [\texttt{num}])\ 10 : \texttt{num}}\;(\text{T-A}\small{\text{PP}})$$

7 Typing Self-Application

- A natural question you might have is: *is System F more expressive than the simply-typed λ-calculus*?
- The answer is *yes*: we can give types to programs that are not typeable in STLC.
- An example is the self-application term:

$$\lambda x.(x\ x)$$

- We saw how we couldn't give a type to this program in STLC because there was no way for the argument to a function to have the same type as the function itself.
- In System F, we can give a type to this program by making use of type application:

$$\omega = \lambda x : (\forall X.X \to X).(x\ [\forall X.X \to X])\ x$$

- Note that this program has type $(\forall X.X \to X) \to (\forall X.X \to X)$
- This seems to imply that System F is extremely expressive, perhaps even so expressive that it can write all programs that can be written in the untyped λ-calculus. But in fact, this is not the case: *just like STLC, System F is strongly normalizing* (meaning, all well-typed System F programs terminate).
- But how can this be, we just showed that we can typecheck self-application!
- Look closely at the types: we see the type of ω is:

$$(\forall X.X \to X) \to (\forall X.X \to X) \tag{4}$$

  Ask yourself: *what kinds of programs can have this type*?
- We saw earlier that there is only a single value of type $(\forall X.X \to X)$: the identity function! So, by inspecting the types, we can conclude that any well-typed ω can only have one possible behavior: it takes an identity function as input, and produces an identity function as its output.
- This restriction on the behavior of ω prevents us from writing $\Omega = \omega\ \omega$, since the second ω argument is required to be the identity function by our above discussion.

References

Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur. 1972.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

John C Reynolds. Towards a theory of type structure. In *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, pages 408–425. Springer, 1974.