Lecture 04.2: Polymorphic and existential types

So far, we have discussed extensions to the lambda calculus that enable us to describe relationships between data (algebraic data types) as well as self-relations within code (fixpoints). In this lecture, we introduced two new type extensions that focused on *abstraction*: how can we write code that is generic over a particular data type? How can we define abstractions that work not over concrete types like `int` or `bool` but *any* type?

1. Polymorphic types

One restriction of the lambda calculus formulated thus far is that it is difficult to enable code reuse across types. In the simplest example, consider the identity function that takes an argument and returns it:

$$\lambda\ (x : \texttt{int})\ .\ x$$

This is a function that should work for any type τ that x could be, not just `int`. However, the lambda calculus forces us to assign a concrete type to the argument, so creating a generic identity function is impossible. We have to define a new function for each type:

$$\begin{aligned}
&\lambda\ (x : \texttt{int})\ .\ x \\
&\lambda\ (x : \texttt{int} \to \texttt{int})\ .\ x \\
&\lambda\ (x : \texttt{int} \to \texttt{int} \to \texttt{int})\ .\ x \\
&\ldots
\end{aligned}$$

And since there are infinitely many types, we would have to define an infinite number of identity functions to be exhaustive. That won't fly.

1.1. OCaml examples

We've already seen that OCaml has a solution to this. For example, in OCaml, if I write the function:

```
let id = fun x -> x
```

If you inspect the type of `id`, then Merlin will tell you `'a -> 'a`. The `'a` (read as "alpha", i.e. α) is a type variable—it means you can replace α with any type and the function will still be valid. Here, we can call `(id 3)` and `(id "hello")`.

Polymorphism occurs frequently in data structures, e.g. lists, stacks, heaps, trees, and so on can all be defined irrespective of what type of element they contain. This idea is represented as type parameters in OCaml, for example:

```
type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf

let x : int list = [1; 2] in
let y : string list = ["a"; "b"] in
let z : int tree = Node (Leaf, 3, Node(Leaf, 2, Leaf))
```

Here, `type 'a tree` creates a new polymorphic type of a binary tree with any possible type of data at the nodes. These data structures, and polymorphism more generally, work well in OCaml due to heavy machinery for inferring polymorphic types and automatically generating code for using polymorphic functions. In order to understand what's actually going on under the hood, we need a more fundamental theory of polymorphic types.

1.2. Theory basics

What we want is to define a single function that is generic with respect to the input type, i.e. it could take any possible input type. This idea is called a *type function*—a piece of code that takes in a type as input. Here's an example in an extended version of our lambda calculus of the polymorphic identity function:

$$\Lambda X \,.\, \lambda\ (y : X) \,.\, y$$

Here, the Λ (capital λ) means a type function that has a parameter X. This X is an example of a *type variable*, in contrast to a term variable like the ones we're used to. We will use the convention that upper-case variables refer to type variables, while lower-case variables are term variables. Also new in this example is the usage of type variables in type expressions, e.g. the type of y in the inner function. To use a type function, we use *type application* to substitute type variables.

$$\begin{aligned} &(\Lambda X \,.\, \lambda\ (y : X) \,.\, y)\ [\texttt{int}] \\ \mapsto\ &[X \to \texttt{int}]\ (\lambda\ (y : X) \,.\, y) \\ =\ &\lambda\ (y : \texttt{int}) \,.\, y \end{aligned}$$

Here, the type function is like a function generator—when we provide it a concrete type, we get back an instance of the identity function for the requested type. This is the core idea of *polymorphism*[1], that a function can operate on terms of many different types. Also, while we originally developed the theory of variables and substitution for use with terms, observe the same ideas apply equally to types. Neat!

The last issue to address is: what type should our polymorphic identity function have? We need to introduce a new type written as $t : \forall X \,.\, \tau$ which means "for any possible type X, the term t has type τ." For example, this is the type of our identity function:

$$(\Lambda X \,.\, \lambda\ (y : X) \,.\, y) : \forall X \,.\, (X \to X)$$

The type reads as "for all types X, this term is a function that takes an X and returns an X."

[1]The etymology here is that "poly" means "many" and "morph" means "form", so the property of dealing with or having many forms.

1.3. Formal semantics

To formalize these semantics, first we will update our grammar:

$$\begin{array}{rll} \mathsf{Type}\ \tau ::= & \dots & \\ & X & \text{type variable} \\ & \forall X\ .\ \tau & \text{polymorphic type} \\ \\ \mathsf{Term}\ t ::= & \dots & \\ & \Lambda X\ .\ t & \text{type function} \\ & t\ [\tau] & \text{type application} \end{array}$$

Now, types are permitted to reference type variables, and we can introduce/eliminate terms with polymorphic types. The statics are as follows:

$$\frac{\Gamma, X \vdash t : \tau}{\Gamma \vdash \Lambda X\ .\ t : \forall X\ .\ \tau}\ \text{(T-tfn)} \qquad \frac{\Gamma \vdash t : \forall X\ .\ \tau_1}{\Gamma \vdash t\ [\tau_2] : [X \to \tau_2]\ \tau_1}\ \text{(T-tapp)}$$

Just like typechecking functions in the simply typed lambda calculus required us to introduce the typing context to map term variables to types, so do type functions require us to again extend the semantics of the type context. Now, our type context can hold both mappings from term variables to types as well as a set of live type variables, notated by Γ, X which says: "remember that X is a valid type variable."[2]

The (T-tfn) rule says that if a term t has a type τ knowing that X is a type variable, then that term under a type function Λ has type $\forall X\ .\ \tau$, which says "for any possible X, t has type τ." Then (T-tapp) says if t is a type function (i.e. it has a polymorphic type), then applying a type τ_2 to the type function means substituting the type variable X with the type argument τ_2 in the body type τ_1.

The dynamics are uninteresting, as all of the legwork happens at compile time (typechecking), not runtime (interpretation). Nonetheless, we still need them in the language:[3]

$$\frac{}{\Lambda X\ .\ t\ \mathsf{val}}\ \text{(D-tfn)} \qquad \frac{t \mapsto t'}{t\ [\tau] \mapsto t'\ [\tau]}\ \text{(D-tapp}_1\text{)} \qquad \frac{}{(\Lambda X\ .\ t)\ [\tau] \mapsto [X \to \tau]\ t}\ \text{(D-tapp}_2\text{)}$$

Like normal functions, type functions are values and when applied, cause a substitution to occur—however here, the substitution is on a type variable, not a term variable. Note that performing these substitutions should never actually affect the runtime—there are no dynamic rules that change depending on the type of a value. However, it is important still to perform the substitution, because in order to prove progress and preservation, our terms need to be well-typed at every step of evaluation.

[2]Our presentation of the statics here does not, in fact, rely on X existing the type context. The primary reason for knowing X exists is to check for unbound type variables—for example, the term $(\lambda\ (x : Y)\ .\ x)$ with no corresponding type function should not typecheck, as Y is not bound. We could capture this with an auxiliary judgment τ type that says "τ is a valid type" and would check for unbound type variables. This judgment would be applied any time the user introduces a new type by hand, e.g. in function declaration.

[3]The dynamics presented here differ from the ones provided in the assignment—the ones for assignment 4 are simpler, but also technically violate preservation.

2. Existential types

While polymorphism is a useful programming pattern to enable, there's only but so many functions that can be defined over every possible type. More frequently in software development, we want to be abstract over some type but also have some knowledge about what we can do with that type. Most statically-typed language have some notion of an *interface* that captures this idea: we specify what things a type should be able to do, and then we have *implementations* that concretize which types actually can implement the given interface.

2.1. OCaml examples

In OCaml this idea is presented in the form of *modules*. A module is basically like a struct (or a product type)—it groups together a bunch of terms under a single name. For example, we can create a module that implements a counter:

```
module IntCounter = struct
  type t = int

  let make (n : int) : t = n
  let incr (ctr : t) (n : int) : t = ctr + n
  let get (ctr : t) : int = ctr
end
```

This module follows the convention of many OCaml modules where the type *t* represents the main type of the module, in this case the counter. We can then use the counter module as follows:

```
let ctr : IntCounter.t = IntCounter.make 3 in
let ctr : IntCounter.t = IntCounter.incr ctr 5 in
assert((IntCounter.get ctr) = 8);
assert(ctr = 8)
```

This works as we expect, although there's one unsightly detail: we're allowed to directly inspect the value of the `ctr` variable instead of just going through the `IntCounter` implementation, which stinks of bad design. More generally, the question is: how do we write `Counter` code that is generic with respect to the implementation of the counter? For example, I could create another counter implementation using records (product types with labels, or basically a C struct):

```
module RecordCounter = struct
  type t = { x : int }

  let make (n : int) : t = {x = n}
  let incr (ctr : t) (n : int) : t = {x = ctr.x + n}
  let get (ctr : t) : int = ctr.x
end
```

Essentially, we need some way of specifying an interface for these modules, which can we do with the `module type` keyword:

```
module type Counter = sig
```

```
  type t

  val make : int -> t
  val incr : t -> int -> t
  val get  : t -> int
end
```

This declares what's called a module *signature*, or a specification of what values/types a module should contain. This signature says that a `Counter` module should contain some type `t` as well as three functions of various types. We can express that a module adheres to a signature like this:

```
module IntCounter : Counter = struct ... end
```

After specifying that `IntCounter` adheres to the `Counter` interface, our previous example usage is broken now, specifically when we run `assert(ctr = 8)`. We will get the error:

```
Error: This expression has type IntCounter.t
       but an expression was expected of type int
```

Excellent! This says that now, when we use the IntCounter implementation of the Counter interface, we can't assume that the type `t` is an integer, which means we can no longer bypass the interface. The next step would be to erase any mention of the implementation—we should be able to write our code without knowing that `IntCounter` was the implementation at all. This relies on another OCaml mechanism called *functors* that we won't discuss in this class, but you can read about it in Real World OCaml if you're interested.

2.2. Theory basics

As with polymorphism, OCaml modules have a lot of language machinery in place to make them work smoothly. We want to distill down to its essence the kind of abstraction occurring here with separating interfaces from implementation. On a high level, the process we want to capture is that of *type erasure*—we want to take a concrete implementation of something like a counter and then erase the type of the counter such that anything using a given counter implementation cannot violate the abstraction boundary.

First, we need to define a process for creating (introducing) an implementation of some interface, and second, we need a way to use that interface. Here's how we're going to define the same `IntCounter` implementation in our lambda calculus:

$$\begin{aligned}&\{\texttt{int}, (((\lambda\ (n : \texttt{int})\ .\ n), (\lambda\ (c : \texttt{int})\ .\ \lambda\ (n : \texttt{int})\ .\ c + n)), (\lambda\ (c : \texttt{int})\ .\ c))\}\\ &\texttt{as}\ \exists X\ .\ (((\texttt{int} \to X) \times (X \to \texttt{int} \to X)) \times (X \to \texttt{int}))\end{aligned}$$

This syntax describes the packing of a "package", or a term with one of its types erased. Specifically, a package has three components: the implementation, the interface, and the abstracted type.

1. Here, the implementation is the second term in the curly braces, concretely a set of functions to create and manipulate a counter.
2. The interface is the type after the as keyword that says what the implementation's type ought to be after abstracting over its types, which in this case is the implementation's expected type

except with a few `int` replaced with a variable X. Note that here the interface is prefixed with $\exists X$ which we'll explain in a moment.

3. Lastly, the type being abstracted, the thing replaced with X, is written as the first element in the curly braces.

The dual to pack is unpack, which enables a client to open up a package and use its methods. Its syntax looks like this (assuming we have let bindings):

$$\begin{aligned}
&\texttt{unpack}\ \{X, p\} = (\{\texttt{int}, (\ldots)\}\ \texttt{as}\ \exists X\ .\ \ldots)\ \texttt{in}\\
&\texttt{let}\ c : X = p.L.L\ 3\ \texttt{in}\\
&\texttt{let}\ c : X = p.L.R\ c\ 5\ \texttt{in}\\
&p.R\ c
\end{aligned}$$

Here, when we open up a package, we get access to two things: a type variable X representing the abstracted type and a term p representing the packaged term. Recall that $t.L$ means get the left element of the pair t, and R for right. So $p.L.L$ means get the "make" function, $p.L.R$ is "incr", and $p.R$ is "get".

In this example, even though the counter is concretely implemented as an `int`, our packing/unpacking semantics enable us to treat that counter type as a black box type variable X, which forces the client to only use the methods that "understand" the type variable, i.e. those in the package. For example, we could not compute $c + 1$ as that would not typecheck (since $c : X$, not $c : \texttt{int}$). Together, these two constructs enable our language to separate complex interfaces from their implementations, and also allows clients to write code that is abstract with respect to the choice of implementation.

2.3. Formal semantics

Again, we will update our grammar:

$$\begin{array}{rll}
\mathsf{Type}\ \tau ::= & \ldots & \\
& \exists X\ .\ \tau & \text{existential type} \\
\\
\mathsf{Term}\ t ::= & \ldots & \\
& \{\tau_1, t\}\ \texttt{as}\ \exists X\ .\ \tau_2 & \text{type pack} \\
& \texttt{unpack}\ \{X, x\} = t_1\ \texttt{in}\ t_2 & \text{type unpack}
\end{array}$$

We introduce a new type $\exists X\ .\ \tau$ which reads as "there exists an X that satisfies the type τ," contrasting with the previous extension which claimed that a type $\forall X\ .\ \tau$ was valid for all possible values of X. The idea here is that the existence claim allows us to define code that says "I assume that I have some X that satisfies a particular interface. I don't know what that X is right now, but if you give me an implementation, then I will use it." As you saw above, existential types are a little bit harder to understand and work with than polymorphic types, so we will discuss the static semantics at length.

$$\frac{\Gamma \vdash t : [X \to \tau_1]\ \tau_2}{\Gamma \vdash \{\tau_1, t\}\ \texttt{as}\ \tau_2 : \exists X\ .\ \tau_2}\ \text{(T-pack)} \qquad \frac{\Gamma \vdash t_1 : \exists X\ .\ \tau_1 \qquad \Gamma, X, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{unpack}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : \tau_2}\ \text{(T-unpack)}$$

In (T-pack), when we create a package, that package explicitly specifies its interface (τ_2) as well as its abstracted type (τ_1). The typechecker then needs to verify the claim that t fulfills the interface collectively specified by τ_1 and τ_2. Let's say that t actually has the concrete type τ_3. For example, take the simple package:

$$\{\texttt{int}, \lambda\ (y : \texttt{int})\ .\ y\}\ \texttt{as}\ \exists X\ .\ (\texttt{int} \to X)$$

Here, $\tau_1 = \texttt{int}$, $\tau_2 = \texttt{int} \to X$, and $\tau_3 = \texttt{int} \to \texttt{int}$. Note that τ_3 is not explicitly specified, but deducible from the typechecking process on the packaged term $(\lambda\ (y : \texttt{int})\ .\ y)$. Then to verify that the term fulfills its specified interface, we can check this by replacing the existential type variable (X) with the abstracted type ($\tau_1 = \texttt{int}$) in the interface ($\tau_2 = \texttt{int} \to X$) and comparing against the concrete type ($\tau_3 = \texttt{int} \to \texttt{int}$), i.e. checking $[X \to \texttt{int}]\ (\texttt{int} \to X) = \texttt{int} \to \texttt{int}$. Hence, in the general case, typechecking a pack means checking if t has the type $[X \to \tau_1]\ \tau_2$. The returned type of a pack should be the provided interface if it matches the implementation.

In (T-unpack), the rule is more verbose but the logic is more straightforward. To unpack a package t_1, first we need to verify that t_1 actually is a package, i.e. that it has an existential type $\exists X\ .\ \tau_1$. Then, we need to typecheck the body of the term, i.e. the t_2 we're unpacking the package into. Specifically, the body needs access to both the type variable X representing the abstracted type and the term variable x representing the implementation of the package. The package should have type τ_1 from the existential, so we insert both X and $x : \tau_1$ into the type context and typecheck the body t_2, returning the body's type τ_2 as the type for the full expression. For example, take the term:

$$\texttt{unpack}\ \{X, x\} = \{\texttt{int}, \lambda\ (y : \texttt{int})\ .\ y\}\ \texttt{as}\ \exists X\ .\ (\texttt{int} \to X)\ \texttt{in}\ x\ 0$$

Here $t_1 = (\{\texttt{int}, \lambda\ (y : \texttt{int})\ .\ y\}\ \texttt{as}\ \exists X\ .\ (\texttt{int} \to X))$ and $t_2 = (x\ 0)$. From (T-pack) we know $t_1 : \exists X\ .\ \texttt{int} \to X$, so we then typecheck the body $t_2 = (x\ 0)$ with the typing context $(X, x : \texttt{int} \to X)$. From this, we deduce that $(x\ 0)$ has type X, which is the type of the whole term.

Lastly, we need to define dynamics, which as before are uninteresting:

$$\frac{}{\{\tau_1, t\}\ \texttt{as}\ \tau_2\ \mathsf{val}}\ \text{(D-pack)} \qquad \frac{t_1 \mapsto t_1'}{\texttt{unpack}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \mapsto \texttt{unpack}\ \{X, x\} = t_1'\ \texttt{in}\ t_2}\ \text{(D-unpack}_1\text{)}$$

$$\frac{}{\texttt{unpack}\ \{X, x\} = \{\tau_1, t_1\}\ \texttt{as}\ \tau_2\ \texttt{in}\ t_2 \mapsto [X \to \tau_1, x \to t_1]\ t_2}\ \text{(D-unpack}_2\text{)}$$

Packages are values we can pass around, and when we choose to unpack a package in (D-unpack$_2$), we substitute both the packaged type τ_1 and the packaged term t_1 into the body t_2. And that's all! Now we've defined a formal semantics for both polymorphic and existential types, successfully bringing first-order logic to our type system.