Comp Sci 704 | Lecture 28: | Ethan Cecchetti
Fall 2024 | Polymorphic λ-Calculus and System F

Last time we saw how to use recursive types to define the type of a list of integers. We then used that to implement basic list values and operations like empty, cons, isempty, head, and tail. However, the types of those operations were specifically working with intList. What if we want them to work with a list of any type? How can we represent cons that works with a list τ for every type τ?

1 Polymorphic λ-Calculus

We do not even need lists and recursive types to see the problem. Consider the type inference algorithm we previously covered applied to something as simple as the identity function $\mathsf{id} = \lambda x.\, x$. The most general type for id will be $\alpha \to \alpha$ for some free type variable α. Intuitively, that seems to mean that we should be able to apply id to a value of any type, and we can, but *only if we only ever apply it to values of one type*.

As an example, consider the following program.

$$\mathsf{let}\ f = \lambda x.\, x\ \mathsf{in}\ \mathsf{if}\ (f\ \mathsf{true})\ \mathsf{then}\ (f\ 3)\ \mathsf{else}\ (f\ 4)$$

The type inference algorithm will correctly infer the type $\alpha \to \alpha$ for f, but then as soon as it sees the application f true in the condition, it will unify $\alpha = \mathsf{bool}$. Then when it sees the applications in the branches, it will attempt to unify $\mathsf{bool} = \mathsf{int}$, at which point it will fail; those are disparate types that cannot be unified.

The problem is that we do not have as much *polymorphism*[1] as we would like. The solution is to add a new constructor the *universally quantifies over types*. That is,

$$\tau \quad ::= \quad \cdots \mid \alpha \mid \forall \alpha.\, \tau.$$

The type $\forall \alpha.\, \tau$ is a *polymorphic type*, or a *type schema*, which is a pattern with type variables that can be instantiated to obtain actual types. For example, the polymorphic type of id would be

$$\mathsf{id} : \forall \alpha.\, \alpha \to \alpha.$$

Note that, much like the recursive type constructor μ, this $\forall$ constructor binds a type variable with the same notions of scope, free and bound variables, renaming, and safe substitution as both μ for types and λ for terms. Unlike μ, however, these universally quantified type variables are not being defined by this constructor, merely bound. More like a variable in a λ-abstraction, they must be instantiated later when they are used.

The language that results from adding these types is known as the *polymorphic λ-calculus*. It has the same terms and evaluation rules as STLC, but with these extra polymorphic types. All terms that were well-typed before will still be well-typed, but now more terms will be typable as well.

1.1 Typing Rules

In addition to the old typing rules, polymorphic λ-calculus adds two new rules, called the *generalization* and *instantiation* rules, that introduce and eliminate these quantifiers, respectively. The full type system is then as follows.

$$[\textsc{Unit}]\ \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad [\textsc{Var}]\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad [\textsc{Abs}]\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad [\textsc{App}]\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$[\textsc{Generalize}]\ \frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\, \tau} \qquad [\textsc{Instantiate}]\ \frac{\Gamma \vdash e : \forall \alpha.\, \tau}{\Gamma \vdash e : \tau[\alpha \mapsto \tau']}$$

[1] Greek for "many forms"

One notable change here is that the types themselves are *open*—that is, they may contain free variables. The GENERALIZE rule—which is only interesting when $\alpha \in \mathrm{FV}(\tau)$—says that, if $\Gamma \vdash e : \tau$ holds and $\alpha \notin \mathrm{FV}(\Gamma)$, then there are no assumptions involving α, meaning we can safely put in any type for it and the typing proof will still work. We can then conclude that e has type τ for all values of α, or, succinctly, $\Gamma \vdash e : \forall\alpha.\,\tau$. The INSTANTIATE rule is the other side of this concept. It says that, if e has a quantified type, then anything we put in for that type variable will work.

1.1.1 Examples

To see how this works, we can look at a couple of proof trees.

First, we see how to show that $\vdash \lambda x.\,x : \forall\alpha.\,\alpha \to \alpha$.

$$\text{[Generalize]}\ \dfrac{\text{[Abs]}\ \dfrac{\text{[Var]}\ \dfrac{}{x{:}\,\alpha \vdash x : \alpha}}{\vdash \lambda x.\,x : \alpha \to \alpha}}{\vdash \lambda x.\,x : \forall\alpha.\,\alpha \to \alpha}$$

Notably, some terms are typable that were not in STLC. For example, $\omega = \lambda x.\,x\;x$ can now have a type!

$$\text{[Generalize]}\ \dfrac{\text{[Abs]}\ \dfrac{\text{[App]}\ \dfrac{\text{[Instantiate]}\ \dfrac{\text{[Var]}\ \dfrac{}{x{:}\,(\forall\alpha.\,\alpha) \vdash x : \forall\alpha.\,\alpha}}{x{:}\,(\forall\alpha.\,\alpha) \vdash x : \gamma \to \beta} \qquad \dfrac{\dfrac{}{x{:}\,(\forall\alpha.\,\alpha) \vdash x : \forall\alpha.\,\alpha}\ \text{[Var]}}{x{:}\,(\forall\alpha.\,\alpha) \vdash x : \gamma}\ \text{[Instantiate]}}{x{:}\,(\forall\alpha.\,\alpha) \vdash x\;x : \beta}}{\vdash \lambda x.\,x\;x : (\forall\alpha.\,\alpha) \to \beta}}{\vdash \lambda x.\,x\;x : \forall\beta.\,(\forall\alpha.\,\alpha) \to \beta}$$

Unfortunately, this type is not particularly meaningful because *nothing* has the type $\forall\alpha.\,\alpha$. It is said to be *uninhabited*, and we can give it the name void. By a nearly identical argument, however, we can say that ω has type $\forall\beta.\,(\forall\alpha.\,\alpha \to \alpha) \to \beta \to \beta$, which is a meaningful type.

Interestingly, while ω is typable, polymorphism is not enough by itself to allow us to type $\Omega = \omega\;\omega$. We still need recursive types for that. Indeed, polymorphic λ-calculus on its own (without recursive types or something similar) still has the same property of STLC that every well-typed program terminates. Proving that is considerably more complicated and is beyond the scope of this course.

2 Using Polymorphism

While polymorphic λ-calculus allows for reuse of programs in very convenient ways, it has a major downside: type inference is now undecidable. A compiler could try its best and ask programmers to insert type annotations when it fails, but we can also restrict the use of polymorphism to regain decidability.

2.1 Let-Polymorphism

A simple approach taken by languages like Haskell and OCaml is two restrict polymorphism in two ways.

1. Type quantification can only appear at the top level of a type. That is, we only allow polymorphic expressions of the form $\forall\alpha_1, \ldots, \forall\alpha_n.\,\tau$ where τ is quantifier-free.
2. Polymorphism can only be introduced in the context of a let expression, not arbitrary variable bindings.

Together, these restrictions result in the following modifications to the language.

$$\begin{array}{rrcl} \text{Quantifier-Free Terms} & \tau & ::= & \mathsf{unit} \mid \alpha \mid \tau_1 \to \tau_2 \\ \text{Primitive } \mathsf{let} & e & ::= & \cdots \mid \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \end{array}$$

$$[\textsc{PolyLet}]\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\forall\alpha_1, \ldots, \forall\alpha_n.\, \tau_1 \vdash e_2 : \tau_2 \\ \{\alpha_1, \ldots, \alpha_n\} = \mathrm{FV}(\tau_1) - \mathrm{FV}(\Gamma)}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2}$$

Note that we retain the INSTANTIATE rule from Section 1.1, but we remove that GENERALIZE rule.

We previously considered the term $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ to be syntactic sugar for $(\lambda x.\, e_2)\ e_1$. While the two still have the same semantic rules, they are no longer equivalent in the type system. Some terms are well-typed using let, but not using function application, including the example from the beginning of this lecture.

The fact that polymorphism can only be introduced with let expressions is why this is known as *let-polymorphism*. Both Haskell and OCaml use let-polymorphism. In theory, this could cause the type checker to require exponential time, but in practice it is not a problem.

2.2 System F

When we first introduced STLC, we used Church-style terms with types explicitly annotated on function arguments. The corresponding version of polymorphic λ-calculus is called *System F*. In System F, we explicitly abstract terms with respect to types and explicitly instantiate those types before using the term. We therefore augment the syntax of STLC with new types and terms as follows.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \alpha \mid \forall\alpha.\, \tau \\ e & ::= & \cdots \mid \Lambda\alpha.\, e \mid e\ \tau \end{array}$$

These new terms are known as *type abstraction* and *type application*, respectively. Operationally, we can add the following semantic rule

$$\frac{}{(\Lambda\alpha.\, e)\ \tau \longrightarrow e[\alpha \mapsto \tau]}.$$

This just gives a rule for instantiating a polymorphic type. Since these reductions only involve types, they can be performed at compile time.

Unlike in polymorphic λ-calculus, in System F we only want to introduce new type variables if they are going to be bound by a Λ-expression. To check this, the type system needs to keep track of which type variables are bound, and make sure that any type that appears in a type application is *well-formed*, meaning all of its free variables have been bound in the surrounding context. To do this, we introduce a new context Δ to keep track of these variables. In more complicated type systems, Δ would be a partial map from variables to *kinds*, that tell us the space a type lives in (kinds are to types as types are to terms), but right now we only have one kind, so we will simply consider Δ to be a set.

The type system then has two classes of judgements:

$$\Delta \vdash \tau \qquad\qquad \Delta; \Gamma \vdash e : \tau$$

The first says that τ is well-formed in type context Δ, while the second says that we can prove e has type τ in type context Δ and variable context Γ. The rules for the well-formed type judgment are as follows.

$$\frac{}{\Delta \vdash \mathsf{unit}} \qquad \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall\alpha.\, \tau}$$

Right now, since these rules do nothing more than keep track of the type variables bound in the surrounding context, one can show that $\Delta \vdash \tau$ if and only if $\mathrm{FV}(\tau) \subseteq \Delta$.

The typing rules for terms are as follows.

$$\frac{}{\Delta; \Gamma \vdash () : \mathsf{unit}} \qquad \frac{\Gamma(x) = \tau \qquad \Delta \vdash \tau}{\Delta; \Gamma \vdash x : \tau} \qquad \frac{\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \\ \Delta; \Gamma \vdash e_2 : \tau_1\end{array}}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Delta; \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\, \tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e\ \tau' : \tau[\alpha \mapsto \tau']}$$

The first four rules are the same as the rules for STLC, but with Δ included and a requirement that any types that appear be well-formed. The last two rules specify the types for type abstractions and applications, also requiring that types be well-formed. In fact, one can show that if $\Delta; \Gamma \vdash e : \tau$, then τ and all type annotations occurring in e must be well-formed in Δ. In particular $\vdash e : \tau$ is only possible when e and τ are both closed.

To see how to use this, we can look at the polymorphic identity function: $\Lambda\alpha.\, \lambda x{:}\alpha.\, x$ which has type $\forall\alpha.\, \alpha \to \alpha$ according to the following proof.

$$\cfrac{\cfrac{\cfrac{\alpha \vdash \alpha}{\alpha; x{:}\alpha \vdash x : \alpha} \qquad \alpha \vdash \alpha}{\alpha; \cdot \vdash \lambda x{:}\alpha.\, x : \alpha \to \alpha}}{\vdash \Lambda\alpha.\, \lambda x{:}\alpha.\, x : \forall\alpha.\, \alpha \to \alpha}$$

To apply this function, we must specify what type we are applying it to. So, the correct version of our original example written in System F (with booleans and integers) would be:

$$\begin{array}{l}\mathsf{let}\ f = \Lambda\alpha.\, \lambda x{:}\alpha.\, x \\ \mathsf{in\ if}\ (f\ \mathsf{bool\ true}) \\ \quad \mathsf{then}\ (f\ \mathsf{int}\ 3) \\ \quad \mathsf{else}\ (f\ \mathsf{int}\ 3)\end{array}$$