Scribe: Sunny Gleason and Kamen Yotov Lecturer: Andrew Myers

1 Introduction

Last time we explored a versatile type inference algorithm, which is actually built into the ML type inference engine. Inference helps a lot in writing type-safe programs without explicitly specifying some of the types, in a way that we get higher-order functions without tons of obscure (type annotation) code. Further, we noticed that the mentioned algorithm works fine around issues like polymorphic code (actually a limited form of *parametric* polymorphism). This connection between type inference and polymorphism may seem important, but it should be noted that these are two distinct phenomena and should be treated separately.

2 Types of polymorphism

We can distinguish several different types of polymorphism. Some of them are **parametric** polymorphism, **ad-hoc** polymorphism and **subtype** polymorphism.

- Parametric polymorphism (which is the only one we have seen till now) is the primary topic of this lecture. In this case expressions are allowed to have multiple types, all in the form $\forall X_1, X_2, \ldots, X_n.\tau$. Types actually appear as "parameters" to expressions, where also the name comes from. Parametric polymorphism is useful when an expression can be written the same way for many different types. Then the polymorphic value is instantiated on some concrete types.

  In some languages (like Java and C) we don't have parametric polymorphism. In Java we can somehow resolve the problem using subtype polymorphism, but in C it's almost hopeless. The only way to write something slightly polymorphic is to use "**void***", i.e. pointer to anything (in reality nothing) and keep your fingers crossed and your brain working. On the other hand in languages like C++ and Modula-3 we can write generic code by using a language feature, called *templates*.

- Ad-hoc polymorphism is a kind of cheating in the sense that it is not a true polymorphism, because there are no polymorphic values. It primarily means using one and the same name to denote values of different types. An illustrative example is that in most programming languages "+" denotes one operator on integers, another on floats and yet another on strings. In some languages like C++ and Java the programmer is allowed to extend the notion by overloading. Ad-hoc polymorphism can be easily implemented by allowing overloading in the type context Γ.

- Subtype polymorphism is much more interesting, because it is an integral part of almost all object-based and all object-oriented languages. The definition is that one type S is a *subtype* of another type T ($S \leq T$) if all values of S are values of T, i.e. $\mathcal{T}[\![S]\!] \subseteq \mathcal{T}[\![T]\!]$. A value here can be polymorphic, because it is a member of all types that are *supertypes* of its type.

  For example, Java code in the form "class C extends D $\{\ldots\}$" implies $C \leq D$. We will discuss subtype polymorphism in greater detail later, in lecture 35.

3 First-class polymorphic values

Polymorphic values in ML are only bound by "let" statements and are instantiated immediately on use. Let's extend our language, so we can use polymorphic values as first-class values:

$$\begin{array}{rcl} \tau & ::= & B \mid X \mid \tau_1 \rightarrow \tau_2 \\ \sigma & ::= & \forall X.\sigma \mid \tau \mid \sigma_1 \rightarrow \sigma_2 \\ e & ::= & \lambda x:\sigma.\, e \mid e_1\ e_2 \mid \Lambda X.\, e \mid e[\tau] \end{array}$$

Here B is any base type, X is type variable. We have added two new kinds of expressions: type abstraction $\Lambda X.\, e$ and type projection $e[\tau]$. The idea behind is that any polymorphic value e can be *explicitly* instantiated on type τ, using $e[\tau]$. In the Turbak & Gifford text these new expressions are called "plambda" and "proj" respectively.

Note that $\forall X.\sigma$ can be used inside function types to produce for example $(\forall X.X) \to \mathsf{bool}$. This type of capturing parametric polymorphism, where σ and τ are separated and one can instantiate only on τ is called predicative polymorphism.

4 Operational semantics

As one can expect, it is straightforward to provide operational semantics for our new polymorphic λ calculus. What is important to realize here, is that type abstraction and application (instantiation) are purely compile-time phenomena. Here are the allowed small steps:

$$\begin{array}{rcl} (\lambda x : \sigma.\, e_1)\ e_2 & \to & e_1\{e_2/x\} \\ (\Lambda X.\, e)\ [\tau] & \to & e\{\tau/X\} \end{array}$$

Here are some examples to illustrate these rules and techniques:

$$\begin{array}{rcl} IF & \equiv & \Lambda V.\, (\lambda f : \forall X.X \to X \to X.\, (\lambda x : V.\, (\lambda y : V.\, (f[V]\ x\ y)))) \\ TRUE & \equiv & \Lambda V.\, (\lambda x, y : V.\, x) \\ FALSE & \equiv & \Lambda V.\, (\lambda x, y : V.\, y) \\ \textbf{if}\ e_0\ e_1\ e_2 : \tau & \Rightarrow & IF[\tau]\ e_0\ e_1\ e_2 : \tau \end{array}$$

5 Static semantics

In order to define static semantics for our new polymorphic language, we need to extend the grammar with rules for type contexts and type variable contexts:

$$\begin{array}{rcl} \tau & ::= & B \mid X \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \forall X.\sigma \mid \tau \mid \sigma_1 \to \sigma_2 \\ e & ::= & \lambda x : \sigma.\, e \mid e_1\ e_2 \mid \Lambda X.\, e \mid e[\tau] \\ \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma \\ \Delta & ::= & \emptyset \mid \Delta, x \end{array}$$

Now we want to derive type judgements in the form $\Delta; \Gamma \vdash e : \sigma$ when $\Delta \vdash \sigma$. Further, we want to have something like α-reduction for type variables, i.e. we want $\forall X.\sigma \cong \forall Y.\sigma\{Y/X\}$. Here are the new type inference rules:

$$\frac{X \notin \Delta \quad \Delta, X; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \Lambda X.\, e : \forall X.\sigma} \qquad \frac{\Delta; \Gamma \vdash e : \forall X.\sigma \quad \Delta \vdash \tau}{\Delta; \Gamma \vdash e[\tau] : \sigma\{\tau/X\}}$$

We also keep all the type inference rules we had in the non-polymorphic λ calculus language, but:

1. stick an additional Δ in front of each Γ and
2. change all τs to σs,

so that we have $\Delta; \Gamma \vdash \ldots \sigma \ldots$ instead of $\Gamma \vdash \ldots \tau \ldots$.

That is:

$$\frac{}{\Delta;\Gamma, x:\sigma \vdash x:\sigma} \qquad \frac{\Delta;\Gamma, x:\sigma \vdash e:\sigma'}{\Delta;\Gamma \vdash (\lambda x:\sigma.\, e):\sigma \to \sigma'} \qquad \frac{\Delta;\Gamma \vdash e_1:\sigma \to \sigma' \quad \Delta;\Gamma \vdash e_2:\sigma}{\Delta;\Gamma \vdash e_1\ e_2:\sigma'}$$

The invariant that is maintained by these rules is that Γ is always well-formed with respect to Δ, written $\Delta \vdash \Gamma$, which means that all the types in Γ are well-formed:

$$\Delta \vdash \Gamma \iff \forall_{i \in 1..n} \Delta \vdash \tau_i \quad \textit{where}\ \ \Gamma = x_1:\tau_1, \ldots, x_n:\tau_n$$

6 Modeling Predicative Polymorphism

To create a model for this type system, we will need a universe $\mathcal{U}$ of all ordinary type interpretations:

$$\begin{aligned} \mathcal{D}_0 &= \{\mathbb{Z},\ \mathbb{U}\} \\ \mathcal{D}_n &= \{X \to Y \mid X, Y \in \mathcal{D}_{n-1}\} \cup \mathcal{D}_{n-1} \\ \mathcal{U} &= \bigcup_{n \in \omega} \mathcal{D}_n \end{aligned}$$

Thus each element of $\mathcal{U}$ is the meaning of some type. We can define the semantics inductively from the meanings of the base types, *unit* and *int*.

$$\begin{aligned} \mathcal{T}[\![int]\!]\chi &= \mathbb{Z} \\ \mathcal{T}[\![unit]\!]\chi &= \mathbb{U} \\ \mathcal{T}[\![\tau_1 \to \tau_2]\!]\chi &= \mathcal{T}[\![\tau_1]\!]\chi \to \mathcal{T}[\![\tau_2]\!]\chi \\ \mathcal{T}[\![\sigma_1 \to \sigma_2]\!]\chi &= \mathcal{T}[\![\sigma_1]\!]\chi \to \mathcal{T}[\![\sigma_2]\!]\chi \\ \mathcal{T}[\![X]\!]\chi &= \chi(X) \\ \mathcal{T}[\![\forall X.\sigma]\!]\chi &= \prod_{D \in \mathcal{U}} \mathcal{T}[\![\sigma]\!]\chi[X \mapsto D] \end{aligned}$$

The meaning of a type variable X is found by "looking it up" in the type environment χ. The meaning of a polymorphic type $\forall X.\sigma$ is the product of translations of σ over type environments where X is mapped to elements D in $\mathcal{U}$. The product above is a **dependent product**.

7 Interpreting Terms

Thus, given type variable environment χ, ordinary variable environment ρ, such that $\chi \models \Delta$ and $\rho \models \Gamma$, we want:

$$\mathcal{C}[\![\Delta;\Gamma \vdash e:\sigma]\!]\chi\rho \in \mathcal{T}[\![\sigma]\!]\chi$$

The meaning of a variable is simply the value of the variable in ρ. The rules for ordinary function applications and abstractions are straightforward.

$$\begin{aligned} \mathcal{C}[\![\Delta;\Gamma \vdash x:\sigma]\!]\chi\rho &= \rho(x) \\ \mathcal{C}[\![\Delta;\Gamma \vdash e_1\ e_2:\sigma']\!]\chi\rho &= (\mathcal{C}[\![\Delta;\Gamma \vdash e_1:\sigma \to \sigma']\!]\chi\rho)(\mathcal{C}[\![\Delta;\Gamma \vdash e_2:\sigma]\!]\chi\rho) \\ \mathcal{C}[\![\Delta;\Gamma \vdash \lambda x:\sigma.\, e\ :\sigma \to \sigma']\!]\chi\rho &= \lambda v \in \mathcal{T}[\![\sigma]\!]\chi.\, \mathcal{C}[\![\Delta;\Gamma, x:\sigma \vdash e:\sigma']\!]\chi\rho[x \mapsto v] \end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![\Delta;\Gamma \vdash \Lambda X.\, e : \forall T.\sigma]\!]\chi\rho &= \lambda D \in \mathcal{U}.\, \mathcal{C}[\![\Delta, X;\Gamma \vdash e : \sigma]\!]\chi[T \mapsto D]\rho \\ \mathcal{C}[\![\Delta;\Gamma \vdash e[\tau]\ : \sigma\{\tau/T\}]\!]\chi\rho &= (\mathcal{C}[\![\Delta;\Gamma \vdash e : \forall T.\sigma]\!]\chi\rho)\ (\mathcal{T}[\![\tau]\!]\chi)\end{aligned}$$

Polymorphic expressions over a type variable X evaluate to lambda expressions which take a type $D \in \mathcal{U}$ as argument, and return an instantiation where X is mapped to D. Instantiating a polymorphic expression e on type τ is analogous to applying the lambda expression just mentioned on type τ. Of course, we will need an analogous substitution lemma for type schemes: $\mathcal{T}[\![\sigma\{\tau/T\}]\!]\chi = \mathcal{T}[\![\sigma]\!]\chi[T \mapsto \tau]$

8 System F

System F, a.k.a "The polymorphic lambda calculus", merges type schemes and types:

$$\boxed{\begin{array}{c}\tau ::= B \mid X \mid \tau_1 \to \tau_2 \\ \sigma ::= \forall X.\sigma \mid \tau \mid \sigma_1 \to \sigma_2\end{array}} \Rightarrow \boxed{\sigma, \tau ::= B \mid X \mid \tau_1 \to \tau_2 \mid \forall X.\tau}$$

In System F, type schemes are first-class, and may be instantiated. With this *impredicative polymorphism*, type inference becomes undecidable. In addition, we can show that there exists no set-based model for types in System F.

As an example, in System F, we can write a type for the term $(\lambda x.\,(x\ x))$ without using recursion:

$$\mathit{SELF\text{-}APP} \equiv (\lambda x : \forall T.T \to T.\,(x[\forall T.T \to T]\ x)) : (x : \forall T.T \to T) \to (x : \forall T.T \to T)$$

In addition, we find that under System F, all standard λ-calculus encodings can be given types too (Church numerals, etc.)

9 No Set Model

Why can't we represent System F types using a set model? Under the predicative model, we have:

$$\mathcal{T}[\![\forall X.\sigma]\!]\chi = \prod_{D \in \mathcal{U}} \mathcal{T}[\![\sigma]\!]\chi[X \mapsto D]$$

$\forall X.\sigma$ is the set of all functions from type interpretations D (in $\mathcal{U}$) to corresponding sets $\mathcal{T}[\![\sigma]\!]\chi[X \mapsto D]$. We need to extend $\mathcal{U}$ to include σ's. $\forall X.X$ is a function mapping all $\mathcal{D} \in \mathcal{U}$ to $\mathcal{D}$. However, the extension of $\mathcal{T}[\![\forall X.X]\!]$ is $\{\langle \mathcal{D}, \mathcal{D}\rangle \mid \mathcal{D} \in \mathcal{U}\}$. Since $\mathcal{T}[\![\forall X.X]\!] \in \mathcal{U}$, it must be one of the $\mathcal{D}$'s in $\mathcal{T}[\![\forall X.X]\!]$. This means $\mathcal{T}[\![\forall X.X]\!]$ is a set that contains itself, which is not allowed. We require that all sets be well-founded, otherwise we allow *Russell's paradox*: $\{x \mid x \notin x\} \in \{x \mid x \notin x\}$? (Bertrand Russell, b.1872-d.1970).