## 1 2nd-order lambda calculus

Today we introduce a new language that allows expressions to have first class polymorphic types. This language is the 2nd-order lambda calculus. Its flavor of polymorphism is known as *impredicative polymorphism*, as opposed to the *predicative polymorphism* of ML. The grammar of this language is as follows :

$$\sigma ::= \forall \alpha_1, \alpha_2, .....\alpha_n.\tau$$

$$\tau ::= B \mid \tau_0 \rightarrow \tau_1 \mid \alpha \mid \sigma$$

$$e ::= n \mid x \mid \lambda x : \tau.e \mid e_0 e_1 \mid \Lambda\alpha_1, \alpha_2....\alpha_n.e \mid e[\tau_1, \tau_2, ...\tau_n]$$

In the above grammar[1], the $\alpha$ 's in the expression $\Lambda\alpha_1, \alpha_2, ...\alpha_n.e$ stands for the type variables we want to abstract $e$ over. In the next expression the $\tau$'s in $e[\tau_1, \tau_2, ...\tau_n]$ stand for the types for which we instantiate $e$. We see an example in the next section.

*It is interesting to note that we can express predicative polymorphism (without the implicit instantiation that ML provides) by restricting this language somewhat. This is accomplished by disallowing the $\Lambda$ expression as a general expression; we would need to introduce a special* plet *construct to allow variables to be bound to polymorphic values:* (plet $(x : \sigma\ (\Lambda\alpha_1, \alpha_2....\alpha_n\ .\ e_0))\ e_1$)*. In ML, the type variables $\alpha_i$ are introduced automatically and implicitly, and we can use an ordinary* let *or* letrec *to define variables in this manner.*

## 2 Example

An example of an expression in the above language with a desugared **let** expression is :
$(let\ \ (f : (\forall\alpha.\alpha \rightarrow \alpha) \rightarrow int\ \ (\lambda i : (\forall\alpha.\alpha \rightarrow \alpha).\ \ i[int]\ \ 0))$
$\quad(let\ \ ident : \forall\alpha.\alpha \rightarrow \alpha\ \ (\Lambda\alpha'.\ \ (\lambda x : \alpha'.\ \ x)))$
$\quad\quad(f\ \ ident))$
In this expression $f$ takes *ident* as its argument, and in its body instantiates $\alpha'$ with int. So in effect after substitution of *ident* as the argument to $f$, what we get is an identity function applied to zero, that is $((\lambda x : int\ \ x)\ \ 0)$. So the expression evaluates to 0.

We can use the new features in our language to express some features of polymorphism found in some other languages. For example what **Templates** do in $C++$ is that they bind the type to a class when it is called. As for the example given in class, i.e.
$template\langle class\ \ T\rangle$
$sort(T\ [\ \ ]\ \ a)$
Although this is not syntactically correct it does represent what template actually does. In the case above it sorts an array of class $T$ instantiated with type $a$. In our language we could write the type for this construct as $\Lambda\alpha.\ \ \alpha[\ \ ] \rightarrow \alpha[\ \ ]$ . That is, we abstract our expression over the type $\alpha$ and instantiate it later with the type $a$ as in the above example.

[1] Note: As usual B represents the base types