This lecture covers:

- Dependent types
- Parameterized types
- Barendregt cube
- Bounded type parameters

## 1 More on dependent types

In the last lecture, we introduced the concept of a dependent type. For example, we introduced types of the form $e.T$, whose type depends on the type of an expression term $e$. We also saw that introducing a dependent typing feature can be problematic, because in general it renders compile-time type checking impossible. To avoid this problem, languages often restrict the type of term $e$ that can be used in this context. For example, we might require that $e$ is just a variable.

Consider Pascal as an example. In Pascal you can write a type like `array[1..5] of integer`, which means an integer array with array indices from 1 to 5. Some versions of Pascal allow types like `array[1..n] of integer`, where `n` is a variable. This is an example of a dependent type, since the type of the array depends on the value of a variable in the language. The following snippet of Pascal psuedocode shows how this feature can be useful:

```
sort(a : array[1..n] of integer, n : int) : array[1..n] of integer
    ...
    for i=1 to n do
       ... a[i] ...
    end
    ...
end
```

In this case, using dependent types allows the compiler to detect out-of-bounds errors at compile-time, without requiring runtime safety checks.

## 2 Parameterized types

We saw in the last section that a dependent typing feature takes a term and a type and gives a type:

$$\texttt{term} \times \texttt{type} \rightarrow \texttt{type} \quad \text{(dependent types)}$$

We have also seen polymorphism, which takes a term (a $\Lambda$-expression) and a type and generates a term (of the form $e[\tau]$):

$$\texttt{term} \times \texttt{type} \rightarrow \texttt{term} \quad \text{(polymorphic types)}$$

Finally, we saw $\lambda$ evaluation, which takes a $\lambda$-term and an argument term and returns another term:

$$\texttt{term} \times \texttt{term} \rightarrow \texttt{term} \quad (\lambda \text{ evaluation})$$

If we look at the pattern so far, we might wonder if there is another typing feature that maps pairs of types to types. In fact, this is called *parameterized typing* (or *type operators*):

$$\texttt{type} \times \texttt{type} \rightarrow \texttt{type} \quad \text{(parameterized types)}$$

To illustrate the idea behind parameterized types, we look at how this feature appears in a few sample languages.

- **SML :** In SML, we can write:

$$\texttt{datatype}\ \alpha\ \texttt{list}\ =\ \texttt{NIL}\ |\ \texttt{Cons}\ \alpha * \alpha\ \texttt{list}$$

In the above, **list** itself is not a type; **list** is a type constructor that maps a type into another type. If we give a type as a parameter to **list**, it generates a type, where the type operator **list** is a postfix operator.

$$\texttt{list} : \texttt{type} \rightarrow \texttt{type}$$
$$\texttt{int list} : \texttt{type}$$

We could write **list** as follows:

$$\texttt{list} = \lambda\alpha :: \texttt{type}.\ 1 + \alpha * \alpha\ \texttt{list}$$

where we further need to take a fixed point as well.

- **C++ :** C++ has class templates; for example:

```
template ⟨class T⟩
class List{
        ⋮
        T x;    x.foo();
        ⋮
}
```

Then we can declare an object of type `List⟨int⟩`, for example, to create an integer list. However, C++'s template class mechanism is not a very good implementations of parametrized types. For example, in the above code, we see that the implementation of class template **List** requires that the type **T** be a class with a method foo. If we provide an incompatible type as a parameter, the compilation will fail with a compile error inside the class. This would be alarming to a programmer trying to use a List class written by someone else. In other words, the declaration of class template **List** is not enough to guarantee the compilation. Also, the source code for the templated class must be provided to any programmers using the class, since the compiler must statically instantiate a new class for each templated type.

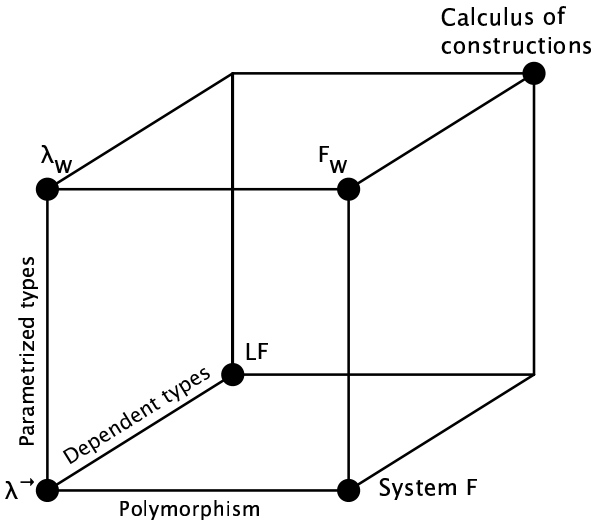
**Java 1.5 :** Java 1.5 adds parameterized polymorphism to Java. Java 1.5 overcomes the above problems in C++, using a bounded type mechanism.

```
class List ⟨T extends I⟩ {
        ⋮
}
```

`T extends I` means that `T` must be a subtype of `I`. This provides a bound on `T`: only types that satisfy the bound can be used to create a list. By ensuring that **List** is instantiated with a proper type `T`, there is no need to look at the source code to check the type. For example, the problematic situation that we observed with C++ could not occur, since $T$ would not be allowed as a parameter if it did not have a `foo()` method.

# 3 Barendregt cube

We have now seen three extensions to the basic typing system: parametrized types, dependent types, and polymorphism. A given language may have any combination of these three features. For example, the simply-typed lambda calculus $\lambda^{\rightarrow}$ has none of them. By adding different combinations of these features to the $\lambda^{\rightarrow}$-calculus, we obtain a family of eight different languages. This idea is represented graphically by the *Barendregt cube*:




Each dimension of the cube represents a typing feature, and each vertex represents a different language. (The diagram above has only a few of the vertices labeled, even though the full cube has a language at each vertex.) At the origin of the cube is $\lambda^{\rightarrow}$, which has none of these three features. Adding parametrized types to $\lambda^{\rightarrow}$ produces a new language called $\lambda_w$. Similarly, adding dependent types produces a language called $LF$ and adding polymorphism produces System F. If we add both parametrized types and polymorphism, we get a language called $F_w$, and so on. If we add all three language features to $\lambda^{\rightarrow}$, we get a language called the *calculus of constructions*.

All languages in the Barendregt cube are strongly normalizing, but we did not prove this in class.

# 4 Parameterized types

## 4.1 Type operators and kindings

As we saw in the last section, $\lambda_\omega$ is an extension to the simply typed lambda-calculus with type operators (parameterized types). Using type operators lets us introduce basic mechanisms of abstraction and application at the level of types. Abstraction and application, in turn, let us think of what it means for two expressions to be equivalent and the notion of well-formedness of types that prevents us from writing nonsensical type expressions.

$\lambda_\omega$ has kindings that can be thought of as "types of types" that prevents nonsensical typing expressions. For example, the type operator **List** has kind `type` $\rightarrow$ `type`, where `type` represents the proper types that we dealt with before the introduction of type operators. This prevents us from writing, for example, the nonsensical type expression `List` ⟨`List` ⟩.

Kinding has the following syntax.

$$K \in \texttt{Kind} := \texttt{type} \mid K \rightarrow K \mid K \times K$$

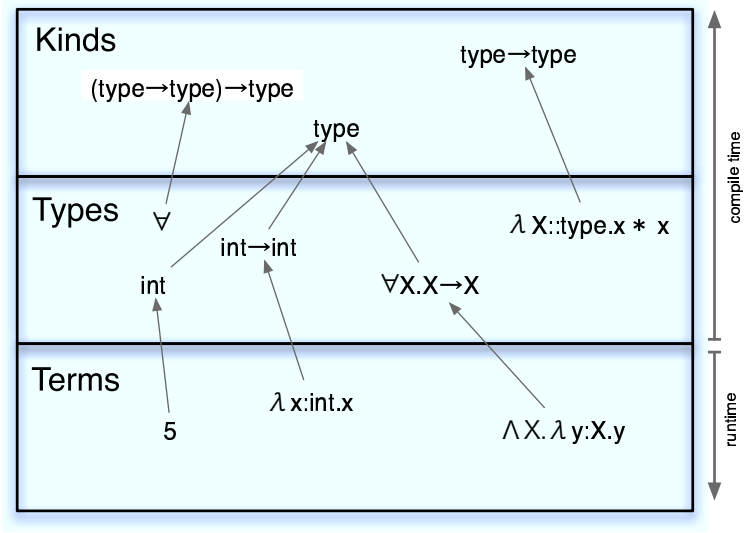
$K \times K$ is not technically in $\lambda_\omega$, but it can be simulated with $K \rightarrow K$. With the introduction of type operators, types are now:

$$\tau \in \texttt{Type} := B \mid X \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \tau_2 \mid \lambda X :: K.\, \tau$$

We can think of familar notations such as $\rightarrow$ and $+$ as base type operators.

$$\rightarrow, + :: (\ \texttt{type} \times \texttt{type}\ ) \rightarrow \texttt{type}$$

We have now talked about three different sorts of entities: terms, types, and kinds. Any valid term has a type, and any valid type has a kind. We can think of these as three different levels of abstraction. The following figure illustrates this point:




In the figure, arrows connect the related expressions across levels; for example, the term **5** has type **int**, which has kind **type**.

### 4.2 Typing and kinding rules

For typing judgements, besides a typing context $\Gamma$ which maps variables into types, we need a typing context $\Delta$ that maps type variables to their kinds.

$$\begin{array}{lll} \Delta & ::= & X_1 :: K_1, ..., X_n :: K_n \\ \Gamma & ::= & x_1 : \tau_1, ..., x_n : \tau_n \end{array}$$

A typing judgement, then, has the form $\Delta; \Gamma \vdash e : \tau$.

With type operators and kinds, the typing rules are similar, except that in lambda abstraction, the argument type must be a `type` kind. The typing rules are:

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Delta; \Gamma, x : \tau \vdash e : \tau' \qquad \Delta \vdash \tau :: \texttt{type}}{\Delta; \Gamma \vdash \lambda x : \tau.\, e : \tau \rightarrow \tau'}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau' \rightarrow \tau \qquad \Delta; \Gamma \vdash e_2 : \tau'}{\Delta; \Gamma \vdash e_1 e_2 : \tau}$$

In addition to typing rules, we now also have to introduce "kinding rules":

$$\frac{}{\Delta, X :: K \vdash X :: K}$$

$$\frac{\Delta, X :: K \vdash \tau :: K'}{\Delta \vdash \lambda X :: K.\, \tau :: K \to K'}$$

$$\frac{\Delta \vdash \tau_0 :: K \to K' \qquad \Delta \vdash \tau_1 :: K}{\Delta \vdash \tau_0 \tau_1 :: K'}$$

The evaluation rule for typing expressions, then, is

$$(\lambda X :: K.\, \tau)\tau' = \tau\{\tau'/X\}$$

With abstraction and application mechanisms and an evaluation rule at the type expression level, we can see that there is more than one way to write the same type. For example, `int` $\to$ `int` and $(\lambda X ::$ `type`$.\, X \to X)$ `int` are equivalent types. We can say two types are equivalent by reducing them to normal forms and comparing them modulo $\alpha$ equivalence: $\vdash \tau_1 \cong \tau_2$.

### 4.3 Bounded type parameters

As we saw with the parameterized typing feature of Java, it is important to be able to restrict the types that can be passed to a type operator. For example, a parameterized abstract data structure like Hashmap might require its parameter type to be a class that implements an equality method. We can do this by bounding the type parameter. For example, to enform a bound of $\tau$, we would write $\lambda X \leq \tau :: K.$ and $\forall X \leq \tau.\tau'$.

Consider the following Java-like definition of a class Hashmap:

$$\texttt{Hashmap}\langle \texttt{K implements Comparable}\langle \texttt{K} \rangle \ \rangle$$

meaning that to be made into a Hashmap, a type has to be Comparable. Note that in this case, K appears in its own bound. Formally, this can be expressed as:

$$\lambda X \leq \text{ Comparable } X.\, \{ \text{ ... class definition ... } \}$$

Some languages, like C++, do not have a bounded type parameter mechanism. A workaround is to add more type parameters to supply methods that are needed by the parameterized class. For the above example, we could define Hashmap as

$$\texttt{Hashmap} \ \langle \ \texttt{K, equality\_operator} \ \rangle$$

### 4.4 Type operators and subtyping

The language $F^{\omega}_{\langle}$ (presented in Pierce's book) combines parameterized types with subtyping, as the superscript and subscript of its name suggest. However, introducing both type operators and subtyping can make it easy to write down rules that are wrong. For example, we might try writing a rule like:

$$\frac{\tau_1 \leq \tau_2}{(\lambda X.\, \tau)\tau_1 \leq (\lambda X.\, \tau)\tau_2} \qquad \text{wrong!}$$

But this implies that if $\tau_1 \leq \tau_2$, it is the case that $\dfrac{\tau_1 \leq \tau_2}{(\lambda X.\, X \to \texttt{int})\ \tau_1 \leq (\lambda X.\, X \to \texttt{int})\ \tau_2}$, which cannot be since $\tau_1 \to \texttt{int} \not\leq \tau_2 \to \texttt{int}$.

One approach to deal with this problem is to introduce a polarity notation, where the superscript + means the type variable is used as covariant, while - means contravariant.

$$\begin{array}{ll} \lambda X^{+}.\, X \to \texttt{int} & : \texttt{not allowed} \\ \lambda X^{+}.\, \texttt{int} \to X & : \texttt{allowed} \end{array}$$

Then we can safely write a rule like:

$$\frac{\tau_1 \leq \tau_2}{(\lambda X^{+}.\, \tau)\tau_1 \leq (\lambda X^{+}.\, \tau)\tau_2}$$