

## Comp 411 Notes - Wed, Apr 27, 2005

Type operators and “Types” of Types ( $\lambda_\omega$ )

Scribe: Moez A. Abdel-Gawad  
(moez@cs.rice.edu)

This lecture presents *kinds*, or types of types. Kinds allow typing lists, for example, regardless of their “element” type. A novel concept for kinds is *type constructors* (also called *type operators*), where new types are constructed by *applying* type constructors to other types.

### 1 Starting Quiz!

In the beginning of this lecture we had a quiz about de Bruijn levels. We were questioned about how to define substitution for lambda terms that are expressed using the ‘de Bruijn levels’ notation.

### 2 What We Know So Far

System F:

$$t ::= X \mid t \rightarrow t \mid \forall X.t \mid \exists X.t$$

The syntax of types in our type systems, expressed as a BNF, is indicative of how “advanced” our type system is, and what its capabilities are. We have been, basically, adding new constructs and constructors that allow us to express new forms of types (like  $\rightarrow$ ,  $\forall$ ,  $\exists$ ). Another dimension/direction for extending our type system can be reached by observing that we used the ‘macro’ types:

$\mathbf{CNat} \equiv \forall X.X \rightarrow (X \rightarrow X) \rightarrow X$  (the first parameter for ‘zero’, and the second parameter for ‘successor’), and

$\mathbf{CList} \ A \equiv \forall X.X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$  (the first parameter for ‘nil’, and the second parameter for ‘cons’)

The second example,  $\mathbf{CList} \ A$ , would motivate our new type system.

### 3 Functions Over Types (Type Operators)

A natural, and very crucial, question to ask for  $\mathbf{CList} \ A$  is whether we should quantify, in some way or another (e.g., like we did for System F), over  $A$ ? In some sense, quantifying over  $A$ , using a  $\forall$  sounds reasonable, yet we should notice that we have been using  $\mathbf{CList} \ A$  actually as a ‘macro’, in which  $A$  (whenever  $\mathbf{CList} \ A$  was applied/used) was *substituted* by the actual element type of the list.

This suggests that, to match our usage, `CList A` could be better viewed as a *function* over types, that abstracts over the element type `A`, and whenever *applied* to a type, returns the type that includes lists with elements of that type. It, thus, is conceptually different from viewing `CList` as a type of *all* lists that have the same type for their elements (which could be expressed using the  $\forall$  quantifier). In the new type system,  $\lambda_\omega$ , `CList` is viewed as a function (over types), that takes a type as its parameter, and returns a type as its output (abstraction and application, not at the level of terms, but at the level of types).

## 4 $\lambda_\omega$

Given the strong intuition above, we realize that our earlier type systems have two levels. Earlier we had:

Types	Nat	String	CList[Nat]
Values/terms	5 $(\lambda x.4)5$	“Rabbit” $(\lambda x.“R”)5$	[1, 2, 3]

To make the transition to  $\lambda_\omega$ , we add an extra level, where we do for types what we did earlier for values and terms, i.e., we add type abstractions (functions) over types, and type applications (application of functions over types to types).

$$\begin{aligned} fx &\Rightarrow f = \lambda x.e && \text{(what we needed/done here...)} \\ FX &\Rightarrow F = \lambda X.t && \text{(...we repeat here for types)} \end{aligned}$$

We now, thus, have  $\lambda_\omega$ :

$$t ::= X \mid t \rightarrow t \mid \lambda X.t \mid t t$$

Kinds	Type (*)			Type $\Rightarrow$ Type (* $\Rightarrow$ *)
Types	Nat	String	CList[Nat]	List

## 5 Type Checking in $\lambda_\omega$

After we’ve added kinds, and for sake of consistency, the syntax for our new type functions would look like:

$$\begin{aligned} fx &\Rightarrow f = \lambda x : t.e \\ FX &\Rightarrow F = \lambda X :: K.t \end{aligned}$$

(Note the `::` notation. `K` stands for kinds.)

Given that we now have abstraction and application (defined in terms of substitution), we can see that we can actually “evaluate” at the type level. We

also see that the type checker, because of recursive definitions, can diverge!

To protect against senseless types (e.g., applying `CNat`, a ‘zeroary function over types’ which cannot be applied) we need to “type-check” the types in programs that use the  $\lambda_\omega$  type system. Typechecking at the level of types is called *kind-checking* (we check the kinds of the types). We thus need a syntax for kinds.

$K ::= * \mid K \Rightarrow K$  (note the simplicity, and the similarity with  $\lambda_{\rightarrow}$ )

This, in effect, becomes a kind system (equivalent to our earlier type-systems) to kind-check (earlier, type-check) our type system.

## 6 Homework

All problems in Ch. 29, that is 29.1.1, and 29.1.2.