Higher-Order Polymorphic Typed Lambda Calculus: Approaches to Formalising Grammars

November 13, 2021

System F_ω [3, 2], also known as higher-order polymorphic lambda calculus, extends System F with richer kinds K, namely the kind $K_1 \to K_2$ of type constructors, and adds type-level lambda-abstraction $\lambda \alpha^K.\, A$ and application $A\, B$.

$$\begin{aligned}
\text{Kinds} \quad & K ::= * \mid K_1 \to K_2 \\
\text{Types} \quad & A, B ::= \iota \mid A \to B \mid \forall \alpha^K.\, A \mid \alpha \mid \lambda \alpha^K.\, A \mid A\, B \\
\text{Terms} \quad & M, N ::= x \mid \lambda x^A.\, M \mid M\, N \mid \Lambda \alpha^K.\, M \mid M\, [A]
\end{aligned}$$

We are also free to extend the set of kinds K with other arbitrary kind constants, which means our set of types we range over with A, B includes non-value types of kind other than $*$. We must therefore consider how to organize the type calculus, and what the kinding rules for our types are. Importantly, we must also consider for what syntactic classes of type metavariables (and hence kinds) do there exist type variables α for, as this determines what kind of types we can quantify over and be polymorphic in. There are a number of options in how to formalize a grammar with rich kinds.

Let's consider adding row types R to our grammar, which is an unordered collection of labels ℓ. We say that rows have kind Row and labels have kind Label. From rows, we can then form variants (sums) $\langle R \rangle$ and records (products) $\{R\}$ which are value types of kind $*$.

$\boxed{\Delta \vdash T : K}$

constant
$$\frac{}{\Delta \vdash \iota : *}$$

function
$$\frac{\Delta \vdash A : * \qquad \Delta \vdash B : *}{\Delta \vdash A \to B : *}$$

forall
$$\frac{\Delta \cdot (\alpha : K) \vdash A : *}{\Delta \vdash \forall \alpha^K.\, A : *}$$

type variable
$$\frac{\alpha : K \in \Delta}{\Delta \vdash \alpha : K}$$

type constructor
$$\frac{\Delta \cdot (\alpha : K_1) \vdash A : K_2}{\Delta \vdash \lambda \alpha^{K_1}.\, A : K_1 \to K_2}$$

type constructor application
$$\frac{\Delta \vdash A : K_1 \to K_2 \qquad \Delta \vdash B : K_1}{\Delta \vdash A\, B : K_2}$$

Figure 1: Kinding Rules (System F_ω)

$\boxed{\Gamma \vdash M : A}$

var
$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

lambda abstraction
$$\frac{\Gamma \cdot (x : A) \vdash M : B}{\Gamma \vdash \lambda x^A.\, M : A \to B}$$

application
$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

type abstraction
$$\frac{\Delta \cdot (\alpha : K) \vdash M : A}{\Gamma \vdash \Lambda \alpha^K.\, M : \forall \alpha^K.\, A}$$

type application
$$\frac{\Gamma \vdash M : \forall \alpha^K. A \qquad \Delta \vdash B : K}{\Gamma \vdash M\, [B] : A[\alpha \mapsto B]}$$

Figure 2: Typing Rules (System F_ω)

0.0.1 Type Indiscriminative Method

The most general way is to use a single type metavariable T which captures all types of different kinds K. We note that this relies on the kinding rules to delineate which types are well-formed. For example:

$$
\begin{array}{ll}
\text{Kinds} & K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \mid \mathsf{Label} \\
\text{Types} & T ::= c \mid T_1 \to T_2 \mid \forall \alpha^K . T \mid \alpha \mid \lambda \alpha^K . T \mid T_1\ T_2 \\
& \quad \mid \quad l \mid l; T \mid \ \cdot \ \mid \{T\} \mid \langle T \rangle
\end{array}
$$

Here we have type constants c which capture value types such as `Bool`. Functions are written $T_1 \to T_2$. Universal quantification $\forall \alpha^K . T$ can quantify over types of any kind K to produce some type T of kind K'. Type variables α can be inhabited by types of any kind which T ranges over. Type abstraction $\lambda \alpha^K . T$ has higher-kind $K_1 \to K_2$. Type application $T_1\, T_2$ can then allow types of kinds other than $*$ and $* \to *$ to be applied to each other. We then have labels l, rows extended with labels $l; T$, empty rows $\cdot$, records $\{T\}$, and variants $\langle T \rangle$.

This grammar therefore permits functions, type application, type abstraction, and universal quantification to work over types of any kind. We note that this grammar on its own cannot dictate what types are well-formed or ill-formed, therefore it is important to have kinding rules to express what is allowed. For example, this type syntax says that a function $l \to l$ between two types of kind Label is possible, however this isn't well-formed as we cannot pass or return labels as values.

$\boxed{\Delta \vdash T : K}$

$$
\textsf{constant}\ \frac{}{\Delta \vdash c : *}
\qquad
\textsf{function}\ \frac{\Delta \vdash T_1 : * \qquad \Delta \vdash T_2 : *}{\Delta \vdash T_1 \to T_2 : *}
\qquad
\textsf{forall}\ \frac{\Delta \cdot (\alpha : K) \vdash T : *}{\Delta \vdash \forall \alpha^K . T : *}
\qquad
\textsf{type variable}\ \frac{\alpha : K \in \Delta}{\Delta \vdash \alpha : K}
$$

$$
\textsf{type constructor}\ \frac{\Delta \cdot (\alpha : K_1) \vdash T : K_2}{\Delta \vdash \lambda \alpha^{K_1} . T : K_1 \to K_2}
\qquad
\textsf{type constructor application}\ \frac{\Delta \vdash T_1 : K_1 \to K_2 \qquad \Delta \vdash T_2 : K_1}{\Delta \vdash T_1\, T_2 : K_2}
\qquad
\textsf{label}\ \frac{}{\Delta \vdash l : \mathsf{Label}}
$$

$$
\textsf{row-extend}\ \frac{\Delta \vdash l : \mathsf{Label} \qquad \Delta \vdash T : \mathsf{Row}}{\Delta \vdash l; T : \mathsf{Row}}
\qquad
\textsf{row-empty}\ \frac{}{\Delta \vdash \cdot : \mathsf{Row}}
\qquad
\textsf{record}\ \frac{\Delta \vdash T : \mathsf{Row}}{\Delta \vdash \{T\} : *}
\qquad
\textsf{variant}\ \frac{\Delta \vdash T : \mathsf{Row}}{\Delta \vdash \langle T \rangle : *}
$$

Figure 3: Kinding Rules

0.0.2 Type Categorization Method

Another method is to distinguish between type metavariables which produce types of different kinds. For example:

$$
\begin{array}{ll}
\text{Kinds} & K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \mid \mathsf{Label} \\
\text{Value Types} & A, B ::= c \mid A \to B \mid \forall \alpha^K .\, A \mid \alpha \mid \lambda \alpha^K .\, A \mid A\, B \mid \{R\} \\
\text{Type Constants} & c ::= \texttt{bool} \mid \texttt{int} \\
\text{Row Types} & R ::= (\!|\ |\!) \mid (\!|\ L \mid R |\!) \mid \rho \\
\text{Label Types} & L ::= l_1 \mid l_2 \mid \ldots
\end{array}
$$

Value types A, B are any types which produce a kind $*$, except for type variables α. This includes type constants `bool`, `int`, record types $\langle R \rangle$, functions $A \to B$, universally quantified types $\forall \alpha^K .\, A$, and type application $A\ B$, which are all of kind $*$. This also includes type constructors $\lambda \alpha^K .\, A$ of kind $K_1 \to K_2$ which takes as input a type of rich kind K_1 but must eventually produce a type of kind $*$ in K_2. Note that although we can universally quantify over types with any kind K in $\forall \alpha^K .\, A$, the type variable α^K must always be used to return a type A of kind $*$. Additionally, although we can abstract over types of kind K in type constructors $\lambda \alpha^K .\, A$, type constructor application $A\, B$ does not allow us to apply type constructors to types of kind other than $*$; this means types such as record constructors $\{\_\}$ cannot exist on their own, only records $\{R\}$ which are already applied to a row type R to yield a kind $*$.

Non-value types are types which produce kinds other than $*$. This includes row types R and label types L. We note that row types R also have their own row type variable ρ, which allows ρ to be used in place of where R can occur, and hence row types can be defined polymorphically. The fact that universal quantification is only defined in value types A, B means that row types must be used in the context of a value type where ρ is quantified over by $\forall\alpha^{\mathsf{Row}}.\, A$ where we can unify ρ and α.

This method restricts type application, type abstraction, and universal quantification to types which produce a kind $*$. This is desirable whenever we want to enforce a stronger well-formed type system within the grammar itself, e.g. that values can only possibly have types A, B with output kinds $*$ at the term-level, and that types which use type constructors to take a richly-kinded type as input must already be fully applied to have a kind $*$.

A disadvantage of this is that type constructor application $A\,B$ does not allow us to apply type constructors to types of kind other than $*$.

If we instead wanted type constructors to be applied to types of any kind K, and allow universal quantification and type abstraction to produce types of rich kinds e.g. Row to be produced and exist on their own, we would need to change some things to make the calculus closer to the design of the type indiscriminative system (in 0.0.1):

1. The cleanest way would be to introduce a metavariable T which ranges types of all kinds, and add universal quantification, type abstraction, and type application to T. Types which can be defined polymorphically still need their own type variables included in their grammar, e.g. value types have type variable α which unifies with $\forall\alpha^K.\, T$ when $K = *$, and row types have type variable ρ which unifies with $\forall\alpha^K.\, T$ when $K = \mathsf{Row}$.

Kinds	$K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \mid \mathsf{Label}$
Types	$T ::= A \mid R \mid L \mid \forall\alpha^K.\, T \mid \lambda\alpha^K.\, T \mid T_1\, T_2$
Value Types	$A, B ::= c \mid A \to B \mid \alpha \mid \{R\}$
Type Constants	$c ::= \texttt{bool} \mid \texttt{int}$
Row Types	$R ::= (\!\mid\,\mid\!) \mid (\!\mid L \mid R \mid\!) \mid \rho$
Label Types	$L ::= l_1 \mid l_2 \mid \ldots$

   When doing this however, we need to be careful about considering the subtype/supertype relation between T and more specific types A, R, L. A consequence of placing type forms $\forall\alpha^K.\, T$, $\lambda\alpha^K.\, T$, and $T_1\, T_2$ under T is that we can no longer use these in place of value types A, B, because T does not occur anywhere else except within its own syntactic category. This means that the type forms such as α and $A \to B$ which A, B can expand out, are more restricted in what they can expand out into. For example, although we can express $\forall\alpha^K.\, (A \to B)$ (by expanding out $\forall\alpha^K.\, T$), it is not possible to represent $(\forall\alpha^K.\, A) \to (\forall\alpha^K.\, B)$ unless we add a form $T_1 \to T_2$ as a member of the metavariable T.

2. Another way would be to introduce the constructs of type variables, universal quantification, and type abstraction for each different kind. However this is very verbose, and we also have to capture type application for all possible kinds we can apply a type abstraction to:

Kinds	$K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \mid \mathsf{Label}$
Value Types	$A, B ::= c \mid A \to B \mid \forall\alpha^K.\, A \mid \alpha \mid \lambda\alpha^K.\, A \mid A\, B \mid \{\, R\,\} \mid A\, R$
Type Constants	$c ::= \texttt{bool} \mid \texttt{int}$
Row Types	$R ::= (\!\mid\,\mid\!) \mid (\!\mid L \mid R \mid\!) \mid \rho \mid \forall\alpha^K.\, R \mid \lambda\alpha^K.\, R \mid R\, L \mid R\, R$
Label Types	$L ::= l_1 \mid l_2 \mid \ldots$

   Note that we can now specify universal quantification and type abstraction over row types as well as value types. Record constructors can be applied to row types using type application $A\, R$, for example, where $A := \lambda\alpha^{\mathsf{Row}}.\, \{\, \alpha\}$ and $R := (\!\mid\,\mid\!)$. Row types can also be applied to labels and other rows using $R\, L$ and $R\, R$, for example, we can apply a row to a label by letting $R := \lambda\alpha^{\mathsf{Label}}.\, (\!\mid \alpha \mid (\!\mid\,\mid\!) \mid\!)$ and $L ::= l_1$.

0.0.3 • Explicitly Kinded Type Indiscriminative Method [1]

When we have a system with rich kinds, we can refine the notion of using a single metavariable T to capture non-value types of kinds other than $*$ (as described in 0.0.1) by assigning *kinds* K to the type metavariable T as T^K to exclude ill-formed types.

Lets first consider a calculus which only includes simple kinds $*$ and higher kinds $K_1 \to K_2$. The arrow kind $K_1 \to K_2$ is used for type constructors like `Maybe` and function types $(\to)$. We assume that there is an initial set of type variables $\alpha \in A$ and type constants $c \in C$ including function types. For each kind K we have a collection of types T^K; in *Variant 1*, this includes type constants c^K, type variables α^K, and type application $T_1^{K_2 \to K}\ T_2^{K_2}$. Alternatively as seen in *Variant 2*, type application can be written as an application of a type constant to type arguments, $c^{K_0}\ T_1^{K_1} \ldots T_n^{K_n}$ where $K_0 = K_1 \to \ldots \to K_n \to K$; this also subsumes the case where c^K takes no type arguments, hence c^K on its own is omitted (in contrast to *Variant 1*).

0.0.4 Variant 1

$$\begin{array}{lll}
\text{Kinds} & K ::= * \mid K_1 \to K_2 & \\
\text{Types} & T^K ::= c^K \mid \forall \alpha^K .\, T \mid \alpha^K \mid \lambda \alpha^K .\, T \mid T_1^{K_2 \to K}\ T_2^{K_2} & \\
\text{Type constants} & c^K ::= \texttt{()}, \texttt{bool}, \texttt{int} & :: * \\
 & \quad \mid\ (\to) & :: * \to * \to * \\
 & \quad \mid\ \texttt{Maybe} & :: * \to *
\end{array}$$

0.0.5 Variant 2 (With and without type scheme)

$$\begin{array}{lll}
\text{Kinds} & K ::= * \mid K_1 \to K_2 & \\
\text{Types} & T^K ::= \forall \alpha^{K_0} .\, T^K \mid \alpha^K & \\
 & \quad \mid \lambda \alpha^{K_0} .\, T^{K_1} \mid c^{K_0}\ T_1^{K_1} \ldots T_n^{K_n} & \\
\text{Type constants} & c^K ::= \texttt{()}, \texttt{bool}, \texttt{int} & :: * \\
 & \quad \mid\ (\to) & :: * \to * \to * \\
 & \quad \mid\ \texttt{Maybe} & :: * \to *
\end{array}$$

$$\begin{array}{lll}
\text{Kinds} & K ::= * \mid K_1 \to K_2 & \\
\text{Monotypes} & T^K ::= \alpha^K \mid \lambda \alpha^{K_0} .\, T^{K_1} \mid c^{K_0}\ T_1^{K_1} \ldots T_n^{K_n} & \\
\text{Polytypes} & \sigma ::= \forall \alpha^K .\, \sigma \mid T^* & \\
\text{Type constants} & c^K ::= (), \texttt{bool}, \texttt{int} & :: * \\
 & \quad \mid\ (\to) & :: * \to * \to * \\
 & \quad \mid\ \texttt{Maybe} & :: * \to *
\end{array}$$

This provides a safer way to specify a grammar for types which can range over multiple kinds including richer kinds. For example using a *row polymorphic calculus* where we have a kind Row:

1. One way of specifying a calculus is to use the *type categorization method* (0.0.2) and explicitly distinguish the metavariables which range over types for different kinds. Here, A, B range over regular types and R ranges over row types. Similarly, α ranges over regular type variables and ρ ranges over row type variables, but when universally quantifying over a bound type variable, we can let α^K represent either.

$$\begin{array}{rl}
\text{Kinds} & K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \\
\text{Regular Types} & A, B ::= \iota \mid A \to B \mid \forall \alpha^K .\, A \mid \alpha \mid \lambda \alpha^K .\, A \mid A\, B \\
\text{Row Types} & R ::= \ell\,; R \mid \cdot \mid \rho
\end{array}$$

2. Another way is to use the *type indiscriminative method* (0.0.1) to let T range over types of all kinds in our language, and α can be used as a type variable in place of these kinds.

$$\begin{array}{rl}
\text{Kinds} & k ::= * \mid k_1 \to k_2 \mid \mathsf{Row} \\
\text{Types} & T ::= \iota \mid T_1 \to T_2 \mid \forall \alpha^k .\, T \mid \alpha \mid \lambda \alpha^K .\, T \mid T_1\, T_2 \mid \ell\,; T \mid \cdot
\end{array}$$

3. Now, we can assign kinds to types in the *type indiscriminative method* to exclude ill-formed types. We let the metavariable for types be parameterised by a kind, and hence range over types of all kinds.

$$\begin{array}{rl}
\text{Kinds} & k ::= * \mid k_1 \to k_2 \mid \mathsf{Row} \\
\text{Types} & T^k ::= \iota \mid T_1^{k_1} \to T_2^{k_2} \mid \forall \alpha^{k_0} .\, T^k \mid \alpha^k \mid \lambda \alpha^{k_0} .\, T^{k_1} \mid T_1^{k_2 \to k}\, T_2^{k_2} \mid \ell\,; T^{\mathsf{Row}} \mid \cdot
\end{array}$$

   We can then let a choice of metavariables range over types over different kinds, e.g. let $\epsilon \doteq T^{\mathsf{Row}}$, and similarly with type variables, e.g. let $\varepsilon \doteq \alpha^{\mathsf{Row}}$.

References

[1] Daan Leijen. "Extensible records with scoped labels." In: *Trends in Functional Programming* 6 (2005), pp. 179–194.

[2] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002. URL: http://kevinluo.net/books/book_Types%20and%20Programming%20Languages%20-%20Benjamin%20C.%20Pierce.pdf.

[3] Cambridge University. *Lambda Calculus Lecture Notes*. https://www.cl.cam.ac.uk/teaching/1415/L28/lambda.pdf. Accessed: 2021-08-18.