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

November 13, 2021

System F_ω [4, 3], 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 $A\, B$.

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

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 various details when organizing the type calculus, such as what syntactic classes of type metavariables are there, what are the kinding rules for our types, which kinds of types can be polymorphic over, and what kinds of types can we use in type application. 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 $*$.

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:

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

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. Lastly, we have labels l, rows extended with labels $l; T$, empty rows $\cdot$, records $\{T\}$, and variants $\langle T \rangle$.

This grammar 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.

$\Delta \vdash T : K$

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

function
$$\frac{\Delta \vdash T_1 : * \qquad \Delta \vdash T_2 : *}{\Delta \vdash T_1 \to T_2 : *}$$

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

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

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

type constructor application
$$\frac{\Delta \vdash T_1 : K_1 \to K_2 \qquad \Delta \vdash T_2 : K_1}{\Delta \vdash T_1\, T_2 : K_2}$$

label
$$\frac{}{\Delta \vdash l : \mathsf{Label}}$$

row-extend
$$\frac{\Delta \vdash l : \mathsf{Label} \qquad \Delta \vdash T : \mathsf{Row}}{\Delta \vdash l ; T : \mathsf{Row}}$$

row-empty
$$\frac{}{\Delta \vdash \cdot : \mathsf{Row}}$$

record
$$\frac{\Delta \vdash T : \mathsf{Row}}{\Delta \vdash \{T\} : *}$$

variant
$$\frac{\Delta \vdash T : \mathsf{Row}}{\Delta \vdash \langle T \rangle : *}$$

Figure 1: Kinding Rules

0.0.2 Type Categorization Method [1]

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

Kinds	$K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \mid \mathsf{Label}$
Value Types	$A, B ::= c \mid A \to B \mid \forall \alpha^K . A \mid \alpha \mid \lambda \alpha^K . A \mid A\, B \mid \{R\} \mid \langle R \rangle$
Row Types	$R ::= l ; R \mid \cdot \mid \rho$
Label Types	$l ::= l_1 \mid l_2 \mid \ldots$

Value types A, B are any types which produce a kind $*$, except for type variables α. This includes type constants c and functions $A \to B$. Universally quantified types, $\forall \alpha^K . A$, are able to quantify over types with any kind K in $\forall \alpha^K . A$ but the type variable α^K must always be used to return a type A of kind $*$. We also consider type constructors $\lambda \alpha^K . A$ of kind $K_1 \to K_2$ as value types, which take as input a type of rich kind K_1 but must eventually produce a type of kind $*$ in K_2. Type application $A\, B$ applies a type constructor A to a value type B. Lastly, 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 $*$.

Note that 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 approach more clearly delineates types of different kinds, and restricts type application, type abstraction, and universal quantification to types which produce a kind $*$. This is generally desirable 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 $*$.

0.0.3 • Explicitly Kinded Type Indiscriminative Method [2]

When we have a system with rich kinds, we can refine the notion of using a single metavariable T by annotating it with a kind K, written T^K. This lets us capture types with kinds other than $*$ whilst being explicit about which types are well-formed.

For each kind K we have a collection of types T^K; this includes type constants c^K, polymorphic types $\forall \alpha^K . T^{K'}$, type variables α^K, and type application $T_1^{K_2 \to K}\ T_2^{K_2}$. The kind signatures for type constants c^K are given explicitly in the grammar, where we use wildcards "_" to represent arguments of type constants.

Kinds	$K ::= * \mid K_1 \to K_2 \mid \mathsf{Row} \mid \mathsf{Label}$	
Types	$T^K ::= c^K \mid \forall \alpha^{K_1}.\, T^{K_2} \mid \alpha^K \mid \lambda \alpha^{K_1}.\, T^{K_2} \mid T_1^{K_2 \to K}\ T_2^{K_2}$	
Type constants	$c^K ::= \texttt{()}, \texttt{bool}, \texttt{int}$	$:: *$
	$\mid \ \_ \to \_$	$:: * \to * \to *$
	$\mid \ l$	$:: \mathsf{Label}$
	$\mid \ \_;\_$	$:: \mathsf{Label} \to \mathsf{Row} \to \mathsf{Row}$
	$\mid \ \cdot$	$:: \mathsf{Row}$
	$\mid \ \{\_\}$	$:: \mathsf{Row} \to \mathsf{Type}$
	$\mid \ \langle \_ \rangle$	$:: \mathsf{Row} \to \mathsf{Type}$

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

References

[1] Daniel Hillerström and Sam Lindley. "Liberating effects with rows and handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development*. 2016, pp. 15–27.

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

[3] 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.

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