

Lambda Calculus – λ^\rightarrow , System F, and System F $_\omega$

November 13, 2021

1 Simply Typed Lambda Calculus (λ^\rightarrow)

Simply typed lambda calculus [3] is also traditionally called λ^\rightarrow , where the arrow \rightarrow indicates the centrality of function types $A \rightarrow B$. The elements of lambda calculus are divided into three “sorts”:

- **terms** ranged over by metavariables M, N .
- **types** ranged over by metavariables A, B . We write $M : A$ to say type M has type A .
- **kinds** ranged over by metavariable K . We write $T :: K$ to say type T has kind K .

The grammar of λ^\rightarrow is given by:

$$\begin{array}{ll} \text{Kinds} & K ::= * \\ \text{Types} & A, B ::= \iota \mid A \rightarrow B \\ \text{Raw terms} & M, N ::= c \mid x \mid \lambda x^A. M \mid M N \end{array}$$

Kinds Kinds play little part in λ^\rightarrow , so their structure trivially consists just of $*$, the kind of standard types.

Types Types consist of base types ι such as integers and booleans, and functions where $A \rightarrow B$ represents a function taking a type A to a type B .

Terms We let x, y, z range over a set of term variables. Constants are represented by terms c . The term $\lambda x^A. M$ (also written $\lambda x : A. M$) says that we can take a variable x of type A a parameter of an expression to get a lambda abstraction. Hence the term application of a term to a term, $M N$, is also a term.

$$\Delta \vdash A : K$$

$$\begin{array}{c} \text{constant} \\ \hline \Delta \vdash \iota : * \end{array} \qquad \begin{array}{c} \text{function} \\ \Delta \vdash A : * \quad \Delta \vdash B : * \\ \hline \Delta \vdash A \rightarrow B : * \end{array}$$

Figure 1: Kinding Rules (λ^\rightarrow)

$$\Gamma \vdash M : A$$

$$\begin{array}{c} \text{constant} \\ \hline \Gamma \vdash c : \iota \end{array} \quad \begin{array}{c} \text{var} \\ \hline \Gamma \vdash x : A \quad x : A \in \Gamma \end{array} \quad \begin{array}{c} \text{lambda} \\ \Gamma \cdot (x : A) \vdash M : B \\ \hline \Gamma \vdash \lambda x^A. M : A \rightarrow B \end{array} \quad \begin{array}{c} \text{application} \\ \Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A \\ \hline \Gamma \vdash M N : B \end{array}$$

Figure 2: Typing Rules (λ^\rightarrow)

2 Polymorphic Typed Lambda Calculus (System F)

System F [2, 3], also known as polymorphic lambda calculus or second-order lambda calculus, is a typed lambda calculus that extends simply-typed lambda calculus. It extends this by adding support for “type-to-term” abstraction, allowing polymorphism through the introduction of a mechanism of universal quantification over types. It therefore formalizes the notion of parametric polymorphism

in programming languages. It is known as second-order lambda calculus because from a logical perspective, it can describe all functions that are provably total in second-order logic.

The grammar of System F is given by:

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

Kinds Kinds remain the same, and all types have kind $*$.

Types We extend types A, B with (polymorphic) type variables α and universally quantified types $\forall \alpha^K. A$ in which the bound type variable α of kind K may appear in A (we note that the only kind K in System F is $*$). A important point to make is that by introducing the type variable α , it is also necessary to also introduce $\forall \alpha^K. A$. This is because α can only exist within the scope of which it is quantified by $\forall \alpha$. We note that in a polymorphic lambda calculus without a type scheme, it is possible for type variables α to appear on their own without being bound to an in-scope quantifier $\forall \alpha$ – therefore the grammar on its own does not ensure well-formed types.

Terms The lambda abstraction term $\lambda x^A. M$ can now take variables x which have universally quantified types $\forall \alpha. A$. We extend terms with type abstraction $\Lambda \alpha^K. M$ (also written $\Lambda \alpha :: K. M$) whose parameter α is a type of kind K and returns a term M , and with type application $M[A]$ whose argument is a type A .

$\Delta \vdash T : K$

$$\begin{array}{c} \text{constant} \\ \hline \Delta \vdash \iota : * \end{array} \quad \begin{array}{c} \text{function} \\ \hline \frac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \rightarrow B : *} \end{array} \quad \begin{array}{c} \text{forall} \\ \hline \frac{\Delta \cdot (\alpha : K) \vdash A : *}{\Delta \vdash \forall \alpha^K. A : *} \end{array} \quad \begin{array}{c} \text{type variable} \\ \hline \frac{\alpha : K \in \Delta}{\Delta \vdash \alpha : K} \end{array}$$

Figure 3: Kinding Rules (System F)

$\Gamma \vdash M : A$

$$\begin{array}{c} \text{var} \\ \hline \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \end{array} \quad \begin{array}{c} \text{lambda abstraction} \\ \hline \frac{\Gamma \cdot (x : A) \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \rightarrow B} \end{array} \quad \begin{array}{c} \text{application} \\ \hline \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \end{array} \quad \begin{array}{c} \text{type abstraction} \\ \hline \frac{\Delta \cdot (\alpha : K) \vdash M : A}{\Gamma \vdash \Lambda \alpha^K. M : \forall \alpha^K. A} \end{array}$$

$$\begin{array}{c} \text{type application} \\ \hline \frac{\Gamma \vdash M : \forall \alpha^K. A \quad \Delta \vdash B : K}{\Gamma \vdash M[B] : A[\alpha \mapsto B]} \end{array}$$

Figure 4: Typing Rules (System F)

3 Higher-Order Polymorphic Typed Lambda Calculus (System F_ω)

System F_ω [3, 1], also known as higher-order polymorphic lambda calculus, extends System F with richer kinds and adds type-level lambda-abstraction and application.

3.0.1 System F_ω

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

Kinds In System F, the structure of kinds has been trivial, limited to a single kind $*$ to which all type expressions belonged. In System F_ω , we enrich the set of kinds with an operator \rightarrow such

that if K_1 and K_2 are kinds, then $K_1 \rightarrow K_2$ is a kind. This allows us to construct kinds which contain *type operators/constructors* and higher-order forms of these, such as product \times . This lets us add other arbitrary custom kind constants to this calculus.

Types The set of types in System F_ω now additionally includes type constructors i.e. type-level lambda-abstraction ($\lambda\alpha^K. A : K \rightarrow K'$), and type constructor application ($AB : K_2$ when $A : K_1 \rightarrow K_2$ and $B : K_1$) as we are able to apply higher-kinded types $K_1 \rightarrow K_1$ to other types.

Additionally, universal quantification ($\forall\alpha^K. A : *$) now requires the bound type variable α to be annotated by a kind K , meaning types can be parameterised by polymorphic type variables of any kind K as long as the overall type returned is of kind $*$.

Terms Although the terms in System F_ω remain the same as System F, the term for type abstraction ($\Lambda\alpha^K. M$) can now take types with kinds other than $*$, as long as there exists a type variable for that specific kind.

The introduction of richer kinds means that it becomes more necessary to add *kinding rules* to dictate what are well-formed types.

$\Delta \vdash T : K$			
constant $\frac{}{\Delta \vdash \iota : *}$	function $\frac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \rightarrow B : *}$	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 $\frac{\Delta \cdot (\alpha : K_1) \vdash A : K_2}{\Delta \vdash \lambda\alpha^{K_1}. A : K_1 \rightarrow K_2}$		type constructor application $\frac{\Delta \vdash A : K_1 \rightarrow K_2 \quad \Delta \vdash B : K_1}{\Delta \vdash AB : K_2}$	

Figure 5: Kinding Rules (System F_ω)

$\Gamma \vdash M : A$			
var $\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$	lambda abstraction $\frac{\Gamma \cdot (x : A) \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \rightarrow B}$	application $\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$	type abstraction $\frac{\Delta \cdot (\alpha : K) \vdash M : A}{\Gamma \vdash \Lambda\alpha^K. M : \forall\alpha^K. A}$
type application $\frac{\Gamma \vdash M : \forall\alpha^K. A \quad \Delta \vdash B : K}{\Gamma \vdash M[B] : A[\alpha \mapsto B]}$			

Figure 6: Typing Rules (System F_ω)

References

- [1] 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.
- [2] Peter Selinger. *Lecture Notes on the Lambda Calculus*. <https://www.irif.fr/~mellies/mpri/mpri-ens/biblio/Selinger-Lambda-Calculus-Notes.pdf>. Accessed: 2021-08-18.
- [3] Cambridge University. *Lambda Calculus Lecture Notes*. <https://www.cl.cam.ac.uk/teaching/1415/L28/lambda.pdf>. Accessed: 2021-08-18.