Lambda Calculus – $\lambda^{\rightarrow}$, System F, and System F_ω

November 14, 2021

1 Simply Typed Lambda Calculus ($\lambda^{\rightarrow}$)

Simply typed lambda calculus [3] is also traditionally called $\lambda^{\rightarrow}$, 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 $\lambda^{\rightarrow}$ is given by:

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

Kinds Kinds play little part in $\lambda^{\rightarrow}$, so their structure trivially consists just of $*$ i.e. the kind of value 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 Term variables are written x. Constants are represented by terms c. The term $\lambda x^A.\, M$ (also written $\lambda x : A.\, M$) is a function which when given some term of type A, binds it to the variable x and returns the term M. Lastly we have application $M\ N$ which applies a term M to a term N.

Below we give the typing and kinding rules for simply typed lambda calculus, where Δ is a kinding context (environment) and Γ is a typing context (environment). We note that one can also choose not to distinguish between kinds and types, and use a single typing context Γ for both.

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

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

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

Figure 1: Kinding Rules ($\lambda^{\rightarrow}$)

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

constant
$$\frac{}{\Delta; \Gamma \vdash c : \iota}$$

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

lambda
$$\frac{\Delta; \Gamma \cdot (x : A) \vdash M : B}{\Delta; \Gamma \vdash \lambda x^A.\, M : A \rightarrow B}$$

application
$$\frac{\Delta; \Gamma \vdash M : A \rightarrow B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\, N : B}$$

Figure 2: Typing Rules ($\lambda^{\rightarrow}$)

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:

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

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 $\forall \alpha^\kappa.\, A$ 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 $*$). An important point to note is that type variables α are only well-formed if they exist within the scope of which they are quantified by $\forall \alpha$. We note that in a polymorphic lambda calculus without a type scheme, such as this one, it is possible for type variables α to appear on their own without being bound to an inscope quantifier $\forall \alpha$ – therefore this grammar on its own does not ensure well-formed types.
Terms Lambda abstraction $\lambda x^A.\, M$ can now take variables x which have universally quantified types, e.g. $\forall \alpha.\, \alpha$. 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. We can then apply types A to type lambda abstractions M using type application $M\, [A]$.

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

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

Figure 3: Kinding Rules (System F)

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

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

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

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

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

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

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

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

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 $\to$ such that if K_1 and K_2 are kinds, then $K_1 \to K_2$ is a kind. This allows us to construct kinds which contain *type operators/constructors* and higher-order forms of these, such as product $\times$. We are then free to extend this calculus with arbitrary custom kind constants.
Types The set of types in System F_ω additionally includes type constructors i.e. type-level lambda-abstraction $\lambda\alpha^K.\, A$, which when provided a type of kind K, binds this to the type variable α and returns the type A. Type constructors A can be applied to a type B to form a new type $A\, B$. 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.
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 $*$.

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

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 5: Kinding Rules (System F_ω)

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

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

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

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

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

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

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.