# Chapter 20

# Girard's System F

The languages we have considered so far are all *monomorphic* in that every expression has a unique type, given the types of its free variables, if it has a type at all. Yet it is often the case that essentially the same behavior is required, albeit at several different types. For example, in $\mathcal{L}\{\texttt{nat}\to\}$ there is a *distinct* identity function for each type $\tau$, namely $\lambda\,(x\!:\!\tau)\,x$, even though the behavior is the same for each choice of $\tau$. Similarly, there is a distinct composition operator for each triple of types, namely

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda\,(f\!:\!\tau_2\to\tau_3)\,\lambda\,(g\!:\!\tau_1\to\tau_2)\,\lambda\,(x\!:\!\tau_1)\,f(g(x)).$$

Each choice of the three types requires a *different* program, even though they all exhibit the same behavior when executed.

Obviously it would be useful to capture the general pattern once and for all, and to instantiate this pattern each time we need it. The expression patterns codify generic (type-independent) behaviors that are shared by all instances of the pattern. Such generic expressions are said to be *polymorphic*. In this chapter we will study a language introduced by Girard under the name *System F* and by Reynolds under the name *polymorphic typed $\lambda$-calculus*. Although motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts, including the concept of data abstraction (the subject of Chapter 21), and the definability of product, sum, inductive, and coinductive types considered in the preceding chapters. (Only general recursive types extend the expressive power of the language.)



## 20.1 System F

*System **F***, or the *polymorphic $\lambda$-calculus*, or $\mathcal{L}\{\to\forall\}$, is a minimal functional language that illustrates the core concepts of polymorphic typing, and permits us to examine its surprising expressive power in isolation from other language features. The syntax of System **F** is given by the following grammar:

| | | | | | |
|---|---|---|---|---|---|
| Typ | $\tau$ | ::= | $t$ | $t$ | variable |
| | | | $\texttt{arr}(\tau_1;\tau_2)$ | $\tau_1\to\tau_2$ | function |
| | | | $\texttt{all}(t.\tau)$ | $\forall(t.\tau)$ | polymorphic |
| Exp | $e$ | ::= | $x$ | $x$ | |
| | | | $\texttt{lam}[\tau](x.e)$ | $\lambda\,(x\!:\!\tau)\,e$ | abstraction |
| | | | $\texttt{ap}(e_1;e_2)$ | $e_1(e_2)$ | application |
| | | | $\texttt{Lam}(t.e)$ | $\Lambda(t.e)$ | type abstraction |
| | | | $\texttt{App}[\tau](e)$ | $e[\tau]$ | type application |

A *type abstraction*, $\texttt{Lam}(t.e)$, defines a *generic*, or *polymorphic*, function with *type parameter* $t$ standing for an unspecified type within $e$. A *type application*, or *instantiation*, $\texttt{App}[\tau](e)$, applies a polymorphic function to a specified type, which is then plugged in for the type parameter to obtain the result. Polymorphic functions are classified by the *universal type*, $\texttt{all}(t.\tau)$, that determines the type, $\tau$, of the result as a function of the argument, $t$.

The statics of $\mathcal{L}\{\to\forall\}$ consists of two judgement forms, the *type formation* judgement,

$$\Delta \vdash \tau\ \mathsf{type},$$

and the *typing judgement*,

$$\Delta\ \Gamma \vdash e : \tau.$$

The hypotheses $\Delta$ have the form $t\ \mathsf{type}$, where $t$ is a variable of sort Typ, and the hypotheses $\Gamma$ have the form $x : \tau$, where $x$ is a variable of sort Exp.

The rules defining the type formation judgement are as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{20.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{20.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{all}(t.\tau)\ \mathsf{type}} \tag{20.1c}$$

The rules defining the typing judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{20.2a}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam[}\tau_1\mathtt{]}(x.e) : \mathtt{arr}(\tau_1;\tau_2)} \tag{20.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{20.2c}$$

$$\frac{\Delta, t \text{ type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{20.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau \text{ type}}{\Delta\ \Gamma \vdash \mathtt{App[}\tau\mathtt{]}(e) : [\tau/t]\tau'} \tag{20.2e}$$

**Lemma 20.1** (Regularity). *If $\Delta\ \Gamma \vdash e : \tau$, and if $\Delta \vdash \tau_i$ type for each assumption $x_i : \tau_i$ in $\Gamma$, then $\Delta \vdash \tau$ type.*

*Proof.* By induction on Rules (20.2). □

The statics admits the structural rules for a general hypothetical judgement. In particular, we have the following critical substitution property for type formation and expression typing.

**Lemma 20.2** (Substitution).

1. *If $\Delta, t$ type $\vdash \tau'$ type and $\Delta \vdash \tau$ type, then $\Delta \vdash [\tau/t]\tau'$ type.*
2. *If $\Delta, t$ type $\Gamma \vdash e' : \tau'$ and $\Delta \vdash \tau$ type, then $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.*
3. *If $\Delta\ \Gamma, x : \tau \vdash e' : \tau'$ and $\Delta\ \Gamma \vdash e : \tau$, then $\Delta\ \Gamma \vdash [e/x]e' : \tau'$.*

The second part of the lemma requires substitution into the context, $\Gamma$, as well as into the term and its type, because the type variable $t$ may occur freely in any of these positions.

Returning to the motivating examples from the introduction, the polymorphic identity function, $I$, is written

$$\Lambda(t.\lambda\ (x{:}t)\ x);$$

it has the polymorphic type

$$\forall(t.t \to t).$$

Instances of the polymorphic identity are written $I[\tau]$, where $\tau$ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function, $C$, is written

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda\ (f{:}t_2 \to t_3)\ \lambda\ (g{:}t_1 \to t_2)\ \lambda\ (x{:}t_1)\ f(g(x))))).$$





The function $C$ has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

Instances of $C$ are obtained by applying it to a triple of types, writing $C[\tau_1]\,[\tau_2]\,[\tau_3]$. Each such instance has the type

$$(\tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3).$$

## Dynamics

The dynamics of $\mathcal{L}\{\to\forall\}$ is given as follows:

$$\frac{}{\mathtt{lam[}\tau\mathtt{]}(x.e) \text{ val}} \tag{20.3a}$$

$$\frac{}{\mathtt{Lam}(t.e) \text{ val}} \tag{20.3b}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam[}\tau_1\mathtt{]}(x.e);e_2) \mapsto [e_2/x]e} \tag{20.3c}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)} \tag{20.3d}$$

$$\frac{}{\mathtt{App[}\tau\mathtt{]}(\mathtt{Lam}(t.e)) \mapsto [\tau/t]e} \tag{20.3e}$$

$$\frac{e \mapsto e'}{\mathtt{App[}\tau\mathtt{]}(e) \mapsto \mathtt{App[}\tau\mathtt{]}(e')} \tag{20.3f}$$

Rule (20.3d) endows $\mathcal{L}\{\to\forall\}$ with a call-by-name interpretation of application. One could easily define a call-by-value variant as well.

It is a simple matter to prove safety for $\mathcal{L}\{\to\forall\}$, using familiar methods.

**Lemma 20.3** (Canonical Forms). *Suppose that $e : \tau$ and $e$ val, then*

1. *If $\tau = \mathtt{arr}(\tau_1;\tau_2)$, then $e = \mathtt{lam[}\tau_1\mathtt{]}(x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \mathtt{all}(t.\tau')$, then $e = \mathtt{Lam}(t.e')$ with $t$ type $\vdash e' : \tau'$.*

*Proof.* By rule induction on the statics. □

**Theorem 20.4** (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*Proof.* By rule induction on the dynamics. □

**Theorem 20.5** (Progress). *If $e : \tau$, then either $e$ val or there exists $e'$ such that $e \mapsto e'$.*

*Proof.* By rule induction on the statics. □

## 20.2 Polymorphic Definability

The language $\mathcal{L}\{\to\forall\}$ is astonishingly expressive. Not only are all finite products and sums definable in the language, but so are all inductive and coinductive types. This is most naturally expressed using definitional equality, which is defined to be the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x:\tau_1 \vdash e:\tau_2 \quad \Delta\ \Gamma \vdash e_1:\tau_1}{\Delta\ \Gamma \vdash \lambda\,(x{:}\tau)\,e_2(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{20.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e:\tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t.e)\,[\rho] \equiv [\rho/t]e : [\rho/t]\tau} \tag{20.4b}$$

In addition there are rules omitted here specifying that definitional equality is reflexive, symmetric, and transitive, and that it is compatible with both forms of application and abstraction.

### 20.2.1 Products and Sums

The nullary product, or unit, type is definable in $\mathcal{L}\{\to\forall\}$ as follows:

$$\begin{aligned}\mathtt{unit} &= \forall(r.r \to r)\\ \langle\rangle &= \Lambda(r.\lambda\,(x{:}r)\,x)\end{aligned}$$

The identity function plays the role of the null tuple, since it is the only closed value of this type.

Binary products are definable in $\mathcal{L}\{\to\forall\}$ by using encoding tricks similar to those described in Chapter 17 for the untyped $\lambda$-calculus:

$$\begin{aligned}\tau_1 \times \tau_2 &= \forall(r.(\tau_1 \to \tau_2 \to r) \to r)\\ \langle e_1, e_2\rangle &= \Lambda(r.\lambda\,(x{:}\tau_1 \to \tau_2 \to r)\,x(e_1)(e_2))\\ e\cdot\mathtt{l} &= e[\tau_1](\lambda\,(x{:}\tau_1)\,\lambda\,(y{:}\tau_2)\,x)\\ e\cdot\mathtt{r} &= e[\tau_2](\lambda\,(x{:}\tau_1)\,\lambda\,(y{:}\tau_2)\,y)\end{aligned}$$

The statics given in Chapter 11 is derivable according to these definitions. Moreover, the following definitional equalities are derivable in $\mathcal{L}\{\to\forall\}$ from these definitions:

$$\langle e_1, e_2\rangle\cdot\mathtt{l} \equiv e_1 : \tau_1$$

and

$$\langle e_1, e_2\rangle\cdot\mathtt{r} \equiv e_2 : \tau_2.$$





The nullary sum, or void, type is definable in $\mathcal{L}\{\to\forall\}$:

$$\begin{aligned}\mathtt{void} &= \forall(r.r)\\ \mathtt{abort}[\rho](e) &= e[\rho]\end{aligned}$$

There is no definitional equality to be checked, there being no introductory rule for the void type.

Binary sums are also definable in $\mathcal{L}\{\to\forall\}$:

$$\begin{aligned}\tau_1 + \tau_2 &= \forall(r.(\tau_1 \to r) \to (\tau_2 \to r) \to r)\\ \mathtt{l}\cdot e &= \Lambda(r.\lambda\,(x{:}\tau_1 \to r)\,\lambda\,(y{:}\tau_2 \to r)\,x(e))\\ \mathtt{r}\cdot e &= \Lambda(r.\lambda\,(x{:}\tau_1 \to r)\,\lambda\,(y{:}\tau_2 \to r)\,y(e))\\ \mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\} &=\\ e[\rho](\lambda\,(x_1{:}\tau_1)\,e_1)(\lambda\,(x_2{:}\tau_2)\,e_2)&\end{aligned}$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in $\mathcal{L}\{\to\forall\}$:

$$\mathtt{case}\,\mathtt{l}\cdot d_1\,\{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\} \equiv [d_1/x_1]e_1 : \rho$$

and

$$\mathtt{case}\,\mathtt{r}\cdot d_2\,\{\mathtt{l}\cdot x_1 \Rightarrow e_1 \mid \mathtt{r}\cdot x_2 \Rightarrow e_2\} \equiv [d_2/x_2]e_2 : \rho.$$

Thus the dynamic behavior specified in Chapter 12 is correctly implemented by these definitions.

### 20.2.2 Natural Numbers

As we remarked above, the natural numbers (under a lazy interpretation) are also definable in $\mathcal{L}\{\to\forall\}$. The key is the representation of the iterator, whose typing rule we recall here for reference:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{natiter}(e_0; e_1; x.e_2) : \tau}\,.$$

Since the result type $\tau$ is arbitrary, this means that if we have an iterator, then it can be used to define a function of type

$$\mathtt{nat} \to \forall(t.t \to (t \to t) \to t).$$

This function, when applied to an argument $n$, yields a polymorphic function that, for any result type, $t$, given the initial result for $\mathtt{z}$ and a transformation from the result for $x$ into the result for $\mathtt{s}(x)$, yields the result of iterating the transformation $n$ times, starting with the initial result.

Since the *only* operation we can perform on a natural number is to iterate up to it in this manner, we may simply *identify* a natural number, $n$, with the polymorphic iterate-up-to-$n$ function just described. This means that we may define the type of natural numbers in $\mathcal{L}\{\to\forall\}$ by the following equations:

$$\begin{aligned}
\mathtt{nat} &= \forall(t.t \to (t \to t) \to t)\\
\mathtt{z} &= \Lambda(t.\lambda\,(z{:}t)\,\lambda\,(s{:}t \to t)\,z)\\
\mathtt{s}(e) &= \Lambda(t.\lambda\,(z{:}t)\,\lambda\,(s{:}t \to t)\,s(e[t](z)(s)))\\
\mathtt{natiter}(e_0;e_1;x.e_2) &= e_0[\tau](e_1)(\lambda\,(x{:}\tau)\,e_2)
\end{aligned}$$

It is easy to check that the statics and dynamics of the natural numbers type given in Chapter 9 are derivable in $\mathcal{L}\{\to\forall\}$ under these definitions.

This shows that $\mathcal{L}\{\to\forall\}$ is *at least as expressive* as $\mathcal{L}\{\mathtt{nat}\to\}$. But is it *more* expressive? Yes! It is possible to show that the evaluation function for $\mathcal{L}\{\mathtt{nat}\to\}$ is definable in $\mathcal{L}\{\to\forall\}$, even though it is not definable in $\mathcal{L}\{\mathtt{nat}\to\}$ itself. However, the same diagonal argument given in Chapter 9 applies here, showing that the evaluation function for $\mathcal{L}\{\to\forall\}$ is not definable in $\mathcal{L}\{\to\forall\}$. We may enrich $\mathcal{L}\{\to\forall\}$ a bit more to define the evaluator for $\mathcal{L}\{\to\forall\}$, but as long as all programs in the enriched language terminate, we will once again have an undefinable function, the evaluation function for that extension.

## 20.3 Parametricity Overview

A remarkable property of $\mathcal{L}\{\to\forall\}$ is that polymorphic types severely constrain the behavior of their elements. One may prove useful theorems about an expression knowing *only* its type—that is, without ever looking at the code. For example, if $i$ is *any* expression of type $\forall(t.t \to t)$, then it must be the identity function. Informally, when $i$ is applied to a type, $\tau$, and an argument of type $\tau$, it must return a value of type $\tau$. But since $\tau$ is not specified until $i$ is called, the function has no choice but to return its argument, which is to say that it is essentially the identity function. Similarly, if $b$ is *any* expression of type $\forall(t.t \to t \to t)$, then $b$ must be either $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,x)$ or $\Lambda(t.\lambda\,(x{:}t)\,\lambda\,(y{:}t)\,y)$. For when $b$ is applied to two arguments of some type, its only choice to return a value of that type is to return one of the two.

What is remarkable is that these properties of $i$ and $b$ have been derived *without knowing anything about the expressions themselves*, but only their





types. The theory of parametricity implies that we are able to derive theorems about the behavior of a program knowing only its type. Such theorems are sometimes called *free theorems* because they come "for free" as a consequence of typing, and require no program analysis or verification to derive. These theorems underpin the remarkable experience with polymorphic languages that well-typed programs tend to behave as expected when executed. That is, satisfying the type checker is sufficient condition for correctness. Parametricity so constrains the behavior of a program that there are relatively few programs of the same type that exhibit unintended behavior, ruling out a large class of mistakes that commonly arise when writing code. Parametricity also guarantees representation independence for abstract types, a topic that is discussed further in Chapter 21.

## 20.4 Restricted Forms of Polymorphism

In this section we briefly examine some restricted forms of polymorphism with less than the full expressive power of $\mathcal{L}\{\to\forall\}$. These are obtained in one of two ways:

1. Restricting type quantification to unquantified types.
2. Restricting the occurrence of quantifiers within types.

### 20.4.1 Predicative Fragment

The remarkable expressive power of the language $\mathcal{L}\{\to\forall\}$ may be traced to the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let $\tau$ be the type $\forall(t.t \to t)$, and, assuming that $e : \tau$, we may apply $e$ to its own type, obtaining the expression $e[\tau]$ of type $\tau \to \tau$. Written out in full, this is the type

$$\forall(t.t \to t) \to \forall(t.t \to t),$$

which is larger (both textually, and when measured by the number of occurrences of quantified types) than the type of $e$ itself. In fact, this type is large enough that we can go ahead and apply $e[\tau]$ to $e$ again, obtaining the expression $e[\tau](e)$, which is again of type $\tau$ — the very type of $e$.

This property of $\mathcal{L}\{\to\forall\}$ is called *impredicativity*[1]; the language $\mathcal{L}\{\to\forall\}$ is said to permit *impredicative (type) quantification*. The distinguishing char-

[1] pronounced *im-PRED-ic-a-tiv-it-y*

acteristic of impredicative polymorphism is that it involves a kind of circularity in that the meaning of a quantified type is given in terms of its instances, including the quantified type itself. This quasi-circularity is responsible for the surprising expressive power of $\mathcal{L}\{\to\forall\}$, and is correspondingly the prime source of complexity when reasoning about it (for example, in the proof that all expressions of $\mathcal{L}\{\to\forall\}$ terminate).

Contrast this with $\mathcal{L}\{\to\}$, in which the type of an application of a function is evidently smaller than the type of the function itself. For if $e : \tau_1 \to \tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of $e$. This situation extends to polymorphism, provided that we impose the restriction that a quantified type can only be instantiated by an un-quantified type. For in that case passage from $\forall(t.\tau)$ to $[\rho/t]\tau$ decreases the number of quantifiers (even if the size of the type expression viewed as a tree grows). For example, the type $\forall(t.t \to t)$ may be instantiated with the type $u \to u$ to obtain the type $(u \to u) \to (u \to u)$. This type has more symbols in it than $\tau$, but is smaller in that it has fewer quantifiers. The restriction to quantification only over unquantified types is called *predicative*[2] *polymorphism*. The predicative fragment is significantly less expressive than the full impredicative language. In particular, the natural numbers are no longer definable in it.

### 20.4.2 Prenex Fragment

A rather more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level — not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is often imposed for the sake of type inference, which permits type annotations to be omitted entirely in the knowledge that they can be recovered from the way the expression is used. We will not discuss type inference here, but we will give a formulation of the prenex fragment of $\mathcal{L}\{\to\forall\}$, because it plays an important role in the design of practical polymorphic languages.

The prenex fragment of $\mathcal{L}\{\to\forall\}$ is designated $\mathcal{L}^1\{\to\forall\}$, for reasons that will become clear in the next subsection. It is defined by *stratifying* types into two sorts, the *monotypes* (or *rank*-0 types) and the *polytypes* (or *rank*-1 types). The monotypes are those that do not involve any quantification, and may be used to instantiate the polymorphic quantifier. The polytypes

[2] pronounced *PRED-i-ca-tive*





include the monotypes, but also permit quantification over monotypes. These classifications are expressed by the judgements $\Delta \vdash \tau$ mono and $\Delta \vdash \tau$ poly, where $\Delta$ is a finite set of hypotheses of the form $t$ mono, where $t$ is a type variable not otherwise declared in $\Delta$. The rules for deriving these judgements are as follows:

$$\frac{}{\Delta, t \text{ mono} \vdash t \text{ mono}} \tag{20.5a}$$

$$\frac{\Delta \vdash \tau_1 \text{ mono} \quad \Delta \vdash \tau_2 \text{ mono}}{\Delta \vdash \mathtt{arr}(\tau_1;\tau_2) \text{ mono}} \tag{20.5b}$$

$$\frac{\Delta \vdash \tau \text{ mono}}{\Delta \vdash \tau \text{ poly}} \tag{20.5c}$$

$$\frac{\Delta, t \text{ mono} \vdash \tau \text{ poly}}{\Delta \vdash \mathtt{all}(t.\tau) \text{ poly}} \tag{20.5d}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes.

The statics of $\mathcal{L}^1\{\to\forall\}$ is given by rules for deriving hypothetical judgements of the form $\Delta\ \Gamma \vdash e : \rho$, where $\Delta$ consists of hypotheses of the form $t$ mono, and $\Gamma$ consists of hypotheses of the form $x : \rho$, where $\Delta \vdash \rho$ poly. The rules defining this judgement are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{20.6a}$$

$$\frac{\Delta \vdash \tau_1 \text{ mono} \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1;\tau_2)} \tag{20.6b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2;\tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau} \tag{20.6c}$$

$$\frac{\Delta, t \text{ mono}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{20.6d}$$

$$\frac{\Delta \vdash \tau \text{ mono} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau')}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{20.6e}$$

We tacitly exploit the inclusion of monotypes as polytypes so that all typing judgements have the form $e : \rho$ for some expression $e$ and polytype $\rho$.

The restriction on the domain of a $\lambda$-abstraction to be a monotype means that a fully general `let` construct is no longer definable—there is no means of binding an expression of polymorphic type to a variable. For this reason it is usual to augment $\mathcal{L}\{\to\forall_{\mathsf{p}}\}$ with a primitive `let` construct whose statics is as follows:

$$\frac{\Delta \vdash \tau_1 \text{ poly} \quad \Delta\ \Gamma \vdash e_1 : \tau_1 \quad \Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}[\tau_1](e_1;x.e_2) : \tau_2}\ . \tag{20.7}$$

For example, the expression

$$\texttt{let}\ I\texttt{:}\forall(t.t \to t)\ \texttt{be}\ \Lambda(t.\lambda\ (x\texttt{:}t)\ x)\ \texttt{in}\ I[\tau \to \tau](I[\tau])$$

has type $\tau \to \tau$ for any polytype $\tau$.

### 20.4.3 Rank-Restricted Fragments

The binary distinction between monomorphic and polymorphic types in $\mathcal{L}^1\{\to\forall\}$ may be generalized to form a hierarchy of languages in which the occurrences of polymorphic types are restricted in relation to function types. The key feature of the prenex fragment is that quantified types are not permitted to occur in the domain of a function type. The prenex fragment also prohibits polymorphic types from the range of a function type, but it would be harmless to admit it, there being no significant difference between the type $\rho \to \forall(t.\tau)$ and the type $\forall(t.\rho \to \tau)$ (where $t \notin \rho$). This motivates the definition of a hierarchy of fragments of $\mathcal{L}\{\to\forall\}$ that subsumes the prenex fragment as a special case.

We will define a judgement of the form $\tau\ \mathsf{type}\,[k]$, where $k \geq 0$, to mean that $\tau$ is a type of *rank* $k$. Informally, types of rank 0 have no quantification, and types of rank $k+1$ may involve quantification, but the domains of function types are restricted to be of rank $k$. Thus, in the terminology of Section 20.4.2, a monotype is a type of rank 0 and a polytype is a type of rank 1.

The definition of the types of rank $k$ is defined simultaneously for all $k$ by the following rules. These rules involve hypothetical judgements of the form $\Delta \vdash \tau\ \mathsf{type}\,[k]$, where $\Delta$ is a finite set of hypotheses of the form $t_i\ \mathsf{type}\,[k_i]$ for some pairwise distinct set of type variables $t_i$. The rules defining these judgements are as follows:

$$\frac{}{\Delta, t\ \mathsf{type}\,[k] \vdash t\ \mathsf{type}\,[k]} \tag{20.8a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[0] \quad \Delta \vdash \tau_2\ \mathsf{type}\,[0]}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2)\ \mathsf{type}\,[0]} \tag{20.8b}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[k] \quad \Delta \vdash \tau_2\ \mathsf{type}\,[k+1]}{\Delta \vdash \texttt{arr}(\tau_1;\tau_2)\ \mathsf{type}\,[k+1]} \tag{20.8c}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type}\,[k]}{\Delta \vdash \tau\ \mathsf{type}\,[k+1]} \tag{20.8d}$$

$$\frac{\Delta, t\ \mathsf{type}\,[k] \vdash \tau\ \mathsf{type}\,[k+1]}{\Delta \vdash \texttt{all}(t.\tau)\ \mathsf{type}\,[k+1]} \tag{20.8e}$$





With these restrictions in mind, it is a good exercise to define the statics of $\mathcal{L}^k\{\to\forall\}$, the restriction of $\mathcal{L}\{\to\forall\}$ to types of rank $k$ (or less). It is most convenient to consider judgements of the form $e : \tau\,[k]$ specifying simultaneously that $e : \tau$ and $\tau\ \mathsf{type}\,[k]$. For example, the rank-limited rules for $\lambda$-abstractions is phrased as follows:

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[0] \quad \Delta\ \Gamma, x : \tau_1\,[0] \vdash e_2 : \tau_2\,[0]}{\Delta\ \Gamma \vdash \texttt{lam}[\tau_1](x.e_2) : \texttt{arr}(\tau_1;\tau_2)\,[0]} \tag{20.9a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type}\,[k] \quad \Delta\ \Gamma, x : \tau_1\,[k] \vdash e_2 : \tau_2\,[k+1]}{\Delta\ \Gamma \vdash \texttt{lam}[\tau_1](x.e_2) : \texttt{arr}(\tau_1;\tau_2)\,[k+1]} \tag{20.9b}$$

The remaining rules follow a similar pattern.

The rank-limited languages $\mathcal{L}^k\{\to\forall\}$ clarify the need for a primitive (as opposed to derived) definition mechanism in $\mathcal{L}^1\{\to\forall\}$. The prenex fragment of $\mathcal{L}\{\to\forall\}$ corresponds to the rank-one fragment $\mathcal{L}^1\{\to\forall\}$. The `let` construct for rank-one types is definable in $\mathcal{L}^2\{\to\forall\}$ from $\lambda$-abstraction and application. This definition only makes sense at rank two, since it abstracts over a rank-one polymorphic type, and is therefore not available at lesser rank.

## 20.5 Notes

System F was introduced by Girard (1972) in the context of proof theory and by Reynolds (1974) in the context of programming languages. The concept of parametricity was originally isolated by Strachey, but was not fully developed until the work of Reynolds (1983). The description of parametricity as providing "theorems for free" was popularized by Wadler (1989).