Higher-Order and Dependent Types*

Sebastian Miele

Universität des Saarlandes, Germany

1 Introduction

In order to motivate higher-order and dependent types we can ask the following question: In the previous talks we considered abstractions of terms by

- terms, i.e. terms of the form $\lambda \mathsf{x}{:}\mathsf{T}\,.\,\mathsf{t}$ (abstraction values), and
- types, i.e. terms of the form $\lambda \mathsf{X}\,.\,\mathsf{t}$ (polymorphic values).[1]

The question at hand is whether it makes sense to abstract *types* by terms and/or types as well.

Such abstractions do make sense. Abstractions of types by types are called *type operators* or *higher-order types*. They have been studied by Girard [8] first. Most typed functional programming languages like SML support at least a restricted form of type operators.

Abstractions of types by terms are called *dependent types*. Restrictricted forms are used in the design of module systems [15] and have been studied in the context of practical programming languages (e.g. Russel [7, 10], Cayenne [1], Dependent ML [21, 22], Dependent Assembly Language [20] and shape types [11]).

Both – higher-order and dependent types – have applications in theorem provers and proof checking environments (e.g. AutoMath [6], NuPRL [4], LEGO [14, 19], Coq [3], ALF [16] and ELF [17]).

Assuming working knowledge about System F, the polymorphic lambda calculus, we will further motivate and introduce higher-order

* This is the extended abstract of my talk given in the context of the seminar types and programming languages in WS 2002/2003 at the Universität des Saarlandes (`http://www.ps.uni-sb.de/courses/seminar-ws02/`). It is based on Chapters 29 and 30 of Benjamin C. Pierce's recent book "Types and Programming Languages" [18] and on Section 3 of a paper by Simon Peyton Jones and Erik Meijer [12], which provides a good introduction to the $\boldsymbol{\lambda}$-cube.

[1] We follow Pierce's notational convention and use the lower x, y, z for term-level variables and the capitalized X, Y, Z for type-level variables. We use s, t, u to denote arbitrary term-level and S, T, U to denote arbitrary type-level expressions.

types in Section 2 and obtain System F_ω, the higher-order polymorphic lambda calculus. In Section 3 we will have a brief look into dependent types.

Finally, in Section 4 we will see how several interesting type systems can be collapsed into a very simple one: Barendregt's λ-cube. It provides a simple mechanism for instantiating it to eight different type systems, including System F, System F_ω, and systems with dependent types like the logical framework (System LF [9]) and the calculus of constructions (System CC [5, 13]).

2 Higher-Order Types and Polymorphism: System F_ω

In this section we will extend the types of System F by type operators (abstractions of types by types) and obtain System F_ω, the higher-order polymorphic lambda calculus.

System F_ω is type-safe. However, the corresponding theorems (progress and preservation) are not proved here. The proofs can be found in [18, Chapter 30].

2.1 Type Operators

Consider the following piece of ML code:

```
type 'a stack = 'a list ref
```

It defines a (higher-order) type `stack` of imperative stacks, which is abstracted by the element type `'a`. This type can be instantiated to the type of stacks with integers as elements:

```
type int_stack = int stack
```

Basically, the type `stack` is as a function from types to types: It takes an element type like `int` and returns the instantiated type of stacks with integers as elements. Such functions are called *type operators*. From now on we will use the term "type" to refer to both,

1. type operators like `stack`, and
2. proper types like `int` and `int stack` which actually are inhabited by terms.

In addition to the things found in System F, System F_ω provides abstractions and applications on the type-level; i.e. it additionally provides a lambda calculus on the level of types. This allows us to write the stack example from above as follows[2]:

$$\begin{aligned}\mathsf{Stack} &= \lambda \mathsf{X}\,.\,\mathsf{Ref}(\mathsf{List}\ \mathsf{X})\\ \mathsf{IntStack} &= \mathsf{Stack}\ \mathsf{Nat}\end{aligned}$$

The type Stack is a first-order type operator, because it abstracts proper types by proper types. Although they are of rare practical use, we can also consider higher-order type operators. Indeed, we immediately add higher-order type operators to System F, because adding only first-order type operators does not make the theory presented in the following subsections easier.

2.2 Types of Types: Kinds

Type operators allow us to write meaningless types like Nat Nat. The situation is just like on the term-level, where e.g. the term 5 5 gets stuck. This was the motivation to introduce a typing system; in order to disallow terms that may get stuck.

So in order to allow only meaningful types, we will apply the same procedure one level up and introduce types of types, called *kinds*. Our single base kind is $\star$, the kind of proper types. For the kinds of type operators we use the double arrow $\Rightarrow$ instead of the single arrow $\rightarrow$ used in function types. The notation of the kinding relation is similar to the notation of the typing relation. We only write a double colon instead of a single one. So we write

$$\begin{aligned}&\vdash\ \mathsf{Nat} :: \star\\ &\vdash\ \lambda \mathsf{X}\,.\,\mathsf{Ref}(\mathsf{List}\ \mathsf{X}) :: \star \Rightarrow \star\\ &\vdash\ (\lambda \mathsf{X}\,.\,\mathsf{Ref}(\mathsf{List}\ \mathsf{X}))\ \mathsf{Nat} :: \star\end{aligned}$$

meaning that Nat and $(\lambda \mathsf{X}\,.\,\mathsf{Ref}(\mathsf{List}\ \mathsf{X}))\ \mathsf{Nat}$ are well-kinded proper types, and that $\lambda \mathsf{X}\,.\,\mathsf{Ref}(\mathsf{List}\ \mathsf{X})$ is a well-kinded first-order type operator.

In the simply typed lambda calculus every bound (term-level) variable carries a typing annotation. This makes typing rules considerably simpler. For the same reason we require that every bound

[2] We are assuming builtin types Nat, Ref and List with the obvious semantics here. See [18, Chapter 11] for how they can be built into lambda calculi.

$$
\begin{array}{llr}
\mathsf{K} ::= & \star & \text{(kind of proper types)} \\
\quad \mid & \mathsf{K} \Rightarrow \mathsf{K} & \text{(kinds of type operators)} \\
\\
\mathsf{T} ::= & \mathsf{X} & \text{(type variables)} \\
\quad \mid & \mathsf{Nat} \mid \ldots & \text{(base types)} \\
\quad \mid & \mathsf{T} \to \mathsf{T} & \text{(function types)} \\
\quad \mid & \forall \mathsf{T}::\mathsf{K}\,.\,\mathsf{T} & \text{(universal types)} \\
\quad \mid & \lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{T} & \text{(type operators)} \\
\quad \mid & \mathsf{T}\ \mathsf{T} & \text{(type operator application)} \\
\\
\mathsf{t} ::= & \ldots & \\
\quad \mid & \lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{t} & \text{(type abstraction)}
\end{array}
$$

Fig. 1. New and modified Syntax in System F_ω. The syntax which remains unmodified in comparison to System F (see [18, Chapter 23]) is shown in gray or replaced by "...".

type-level variable in System F_ω carries a kinding annotation. For convenience we will drop annotations which carry the kind $\star$.

Explicit kinding annotations may be dropped entirely; see [18, Chapter 22] on *type reconstruction* for how this can be done.

2.3 Syntax of System F_ω

We are now ready to formally introduce the syntax of System F_ω in Figure 1. The kinds are as described in the previous subsection. In comparison to system F, all type variable bindings have kinding annotations. Especially the type variable bindings in contexts carry this information, so that this information can be accessed during kinding (see below).

2.4 Kinding

Having introduced the syntax of types and kinds in System F_ω formally, we can now formally define the kinding relation. This is done in Figure 2. The rules K-TVar, K-Abs and K-App are precisely the corresponding rules of the simply typed lambda calculus one level up. The other rules do not match typing rules of the simply typed lambda calculus. However, their meaning is obvious.

$$\frac{\mathsf{X}{::}\mathsf{K} \in \Gamma}{\Gamma \vdash \mathsf{X} :: \mathsf{K}} \qquad \text{(K-TVar)}$$

$$\frac{\Gamma, \mathsf{X}{::}\mathsf{K} \vdash \mathsf{T} :: \mathsf{K}'}{\Gamma \vdash \lambda \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{T} :: \mathsf{K} \Rightarrow \mathsf{K}'} \qquad \text{(K-Abs)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: \mathsf{K} \Rightarrow \mathsf{K}' \qquad \Gamma \vdash \mathsf{T} :: \mathsf{K}}{\Gamma \vdash \mathsf{S}\ \mathsf{T} :: \mathsf{K}'} \qquad \text{(K-App)}$$

$$\Gamma \vdash \mathsf{Nat} :: \star \qquad \text{(K-Nat)}$$

$$\Gamma \vdash \mathsf{Bool} :: \star \qquad \text{(K-Bool)}$$

$$\vdots \qquad\qquad \vdots$$

$$\frac{\Gamma \vdash \mathsf{S} :: \star \qquad \Gamma \vdash \mathsf{T} :: \star}{\Gamma \vdash \mathsf{S} \to \mathsf{T} :: \star} \qquad \text{(K-Arrow)}$$

$$\frac{\Gamma, \mathsf{X}{::}\mathsf{K} \vdash \mathsf{T} :: \star}{\Gamma \vdash \forall \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{T} :: \star} \qquad \text{(K-All)}$$

Fig. 2. Kinding rules of System F_ω.

2.5 Typing and Type Equivalence

Figure 3 shows the typing rules of System F_ω. All rules except T-Eq are taken from System F, with some little additions reflecting the introduction of type operators and kinding.

In order to understand the rule T-Eq, assume that we are given terms s and $\mathsf{isEmpty}\,[\mathsf{Nat}]$ for which we can derive the following typing statements:

$$\Gamma \vdash \mathsf{s} : \mathsf{Ref}(\mathsf{List\ Nat})$$
$$\Gamma \vdash \mathsf{isEmpty}\,[\mathsf{Nat}] : ((\lambda \mathsf{X} \,.\, \mathsf{Ref}(\mathsf{List\ X}))\ \mathsf{Nat}) \to \mathsf{Bool}$$

Now we would like to be able to derive:

$$\Gamma \vdash (\mathsf{isEmpty}\,[\mathsf{Nat}])\ \mathsf{s} : \mathsf{Bool}$$

However, the rule T-App does not apply, because the argument types $\mathsf{Ref}(\mathsf{List\ Nat})$ and $(\lambda \mathsf{X} \,.\, \mathsf{Ref}(\mathsf{List\ X}))\ \mathsf{Nat}$ do not match syntactically.

In general, the problem is that type operators allow us to write a single type in different ways: $\mathsf{Ref}(\mathsf{List\ Nat})$ and $(\lambda \mathsf{X} \,.\, \mathsf{Ref}(\mathsf{List\ X}))\ \mathsf{Nat}$ are in some sense equivalent, but not syntactically equal. In order to

$$\frac{\mathsf{x}{:}\mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathsf{S} :: \star \qquad \Gamma, \mathsf{x}{:}\mathsf{S} \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{S} \,.\, \mathsf{t} : \mathsf{S} \to \mathsf{T}} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{S} \to \mathsf{T} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{S}}{\Gamma \vdash \mathsf{t}_1\, \mathsf{t}_2 : \mathsf{T}} \qquad \text{(T-App)}$$

$$\frac{\Gamma, \mathsf{X}{::}\mathsf{K} \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{t} : \forall \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{T}} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathsf{t} : \forall \mathsf{X}{::}\mathsf{K} \,.\, \mathsf{T} \qquad \Gamma \vdash \mathsf{S} :: \mathsf{K}}{\Gamma \vdash \mathsf{t}\, [\mathsf{S}] : [\mathsf{X} \mapsto \mathsf{S}]\mathsf{T}} \qquad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{S} \qquad \mathsf{S} \equiv \mathsf{T} \qquad \Gamma \vdash \mathsf{T} :: \star}{\Gamma \vdash \mathsf{t} : \mathsf{T}} \qquad \text{(T-Eq)}$$

Fig. 3. Typing rules of System F_ω. The parts that are already present in System F are shown in gray.

handle this we must be able to substitute an equivalent type in an already derived typing statement. This is what T-Eq accomplishes. It uses the type equivalence $\equiv$ defined in Figure 4.

The most important rules of type equivalence are Q-Beta and Q-Eta which exploit $\beta-$ and $\eta-$equivalence of lambda terms. The rules Q-Arrow, Q-All, Q-Abs, Q-App exploit the congruence properties of the lambda calculus and allow replacing arbitrary subexpressions. Finally, Q-Refl, Q-Symm, Q-Trans make the type equivalence an equivalence relation.

Now we are able to derive the equivalence of $((\lambda \mathsf{X} \,.\, \mathsf{Ref}(\mathsf{List}\ \mathsf{X}))\ \mathsf{Nat}) \to \mathsf{Bool}$ and $(\mathsf{Ref}(\mathsf{List}\ \mathsf{Nat})) \to \mathsf{Bool}$:

$$\dfrac{\dfrac{}{(\lambda \mathsf{X} \,.\, \mathsf{Ref}(\mathsf{List}\ \mathsf{X}))\ \mathsf{Nat} \equiv \mathsf{Ref}(\mathsf{List}\ \mathsf{Nat})}\ \text{Q-Beta}}{((\lambda \mathsf{X} \,.\, \mathsf{Ref}(\mathsf{List}\ \mathsf{X}))\ \mathsf{Nat}) \to \mathsf{Bool} \equiv (\mathsf{Ref}(\mathsf{List}\ \mathsf{Nat})) \to \mathsf{Bool}}\ \text{Q-Arrow}$$

Using T-Eq, this allows us to derive the statement

$$\Gamma \vdash \mathsf{isEmpty}\, [\mathsf{Nat}] : (\mathsf{Ref}(\mathsf{List}\ \mathsf{Nat})) \to \mathsf{Bool}$$

with the type $\mathsf{Ref}(\mathsf{List}\ \mathsf{Nat})$ on the left hand side, which is the type derived for s. So using T-Arrow we can now derive the desired

$$(\lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{T})\ \mathsf{S}\ \equiv\ [\mathsf{X}\mapsto\mathsf{S}]\mathsf{T} \qquad \text{(Q-Beta)}$$

$$\frac{\mathsf{X}\notin FV(\mathsf{T})}{\lambda \mathsf{X}::\mathsf{K}\,.\,(\mathsf{T}\ \mathsf{X})\ \equiv\ \mathsf{T}} \qquad \text{(Q-Eta)}$$

$$\frac{\mathsf{S}_1\equiv\mathsf{T}_1 \qquad \mathsf{S}_2\equiv\mathsf{T}_2}{\mathsf{S}_1\to\mathsf{S}_2\equiv\mathsf{T}_1\to\mathsf{T}_2} \qquad \text{(Q-App)}$$

$$\frac{\mathsf{S}\equiv\mathsf{T}}{\lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{S}\equiv\lambda \mathsf{X}::\mathsf{K}\,.\,\mathsf{T}} \qquad \text{(Q-Abs)}$$

$$\frac{\mathsf{S}\equiv\mathsf{T}}{\forall \mathsf{X}::\mathsf{K}\,.\,\mathsf{S}\equiv\forall \mathsf{X}::\mathsf{K}\,.\,\mathsf{T}} \qquad \text{(Q-All)}$$

$$\mathsf{T}\equiv\mathsf{T} \qquad \text{(Q-Refl)}$$

$$\frac{\mathsf{T}\equiv\mathsf{S}}{\mathsf{S}\equiv\mathsf{T}} \qquad \text{(Q-Symm)}$$

$$\frac{\mathsf{S}\equiv\mathsf{T} \qquad \mathsf{T}\equiv\mathsf{U}}{\mathsf{S}\equiv\mathsf{U}} \qquad \text{(Q-Trans)}$$

Fig. 4. Type equivalence.

statement:

$$\Gamma\ \vdash\ (\mathsf{isEmpty}\,[\mathsf{Nat}])\ \mathsf{s}\ :\ \mathsf{Bool}$$

2.6 Deciding Typing

All rules except T-Eq are syntax-directed, i.e. the premises only mention proper subterms of the term in the conclusion. Contrarily, the rule T-Eq can be applied arbitrarily often without reducing the size of the term under consideration. So if we can somehow get rid of T-Eq, then the other rules already provide a decision procedure for the typing relation.

So the question at hand is: What do we need T-Eq for? It has two purposes:

1. *Testing whether a type (or some equivalent of it) has a specific shape:* In T-App we must be able to test, whether the term t_1 has an arrow type, and in T-TApp whether t_1 has a universal type.

2. *Testing whether two types are equivalent:* In T-APP we must be able to match argument types. In the previous section we already saw an example of this.

In subsection 2.4 we already mentioned that – to some degree – higher-order types and kinds in System F_ω correspond to terms and types in the simply typed lambda calculus. The correspondence is only "to some degree", because the simply typed lambda calculus does not provide term-level counterpart to proper types per se. However, this can be fixed by adding term-level constants as counterparts to base types, $\rightarrow$, and $\forall$.

So well-kinded types in System F_ω have the same nice properties as well-typed terms in the simply typed lambda calculus; in particular: strong normalization and confluence under $\beta, \eta-$reduction. Since we only deal with well-kinded types, testing whether a type or an equivalent of it has a specific shape can be done by normalizing the type and then testing whether the result has the desired shape. Furthermore, from the confluence property it follows that two types are equivalent if and only if their normal forms are equal.

So both problems arising with the rule T-EQ are decidable and hence the typing relation of System F_ω is decidable.

2.7 Higher-Order Polymorphism

Now we have completely introduced System F_ω, the higher-order polymorphic lambda calculus. What is still missing is an example of higher-order polymorphism. Consider the following piece of SML code:

```
signature STACK_ADT =
sig type 'a stack
    val stack : unit -> 'a stack
    val isEmpty : 'a stack -> bool
    val push : 'a stack -> 'a -> unit
    val pop : 'a stack -> 'a
end
```

With higher-order existential types we could write it as follows:

$$
\begin{aligned}
\mathsf{StackADT} = \exists \mathsf{Stack}{::}\star \Rightarrow \star\,.\,\{&\mathsf{stack} &&: \forall \mathsf{X}\,.\,\mathsf{Unit} \to \mathsf{Stack}\,\mathsf{X},\\
&\mathsf{isEmpty} &&: \forall \mathsf{X}\,.\,\mathsf{Stack}\,\mathsf{X} \to \mathsf{Bool},\\
&\mathsf{push} &&: \forall \mathsf{X}\,.\,\mathsf{Stack}\,\mathsf{X} \to \mathsf{X} \to \mathsf{Unit},\\
&\mathsf{pop} &&: \forall \mathsf{X}\,.\,\mathsf{Stack}\,\mathsf{X} \to \mathsf{X}\}
\end{aligned}
$$

Note that we did not introduce higher-order existential types in System F_ω, although they can be added easily (see [18, Figure 30-1]). However, this is not necessary, because existentials can be expressed using universals (see [18, Section 24.3]).

2.8 Fragments of System F_ω

System F is often called System F_2 or the second-order lambda calculus. Let us see, why. Consider the following hierarchy of kinds:

$$
\begin{aligned}
\mathcal{K}_2 &= \{\star\}\\
\mathcal{K}_{i+1} &= \mathcal{K}_i \cup \{\mathsf{K}_1 \Rightarrow \ldots \Rightarrow \mathsf{K}_n \mid \mathsf{K}_j \in \mathcal{K}_i\}\\
\mathcal{K}_\omega &= \textstyle\bigcup_{i \in \mathbb{N}_{\geq 2}} \mathcal{K}_i
\end{aligned}
$$

The set $\mathcal{K}_2$ only contains the kind of proper types. The set $\mathcal{K}_3$ additionally contains the kinds of first-order type operators. $\mathcal{K}_4$ additionally contains the kinds of second-order type operators, and so on. Finally, $\mathcal{K}_\omega$, the limit of these sets, is the set of all F_ω-kinds.

We can now define a hierarchy of fragments of System F_ω:

System	Quantification over Types from	Types from
F_1	$\emptyset$	$\{\star\}$
F_2	$\{\star\}$	$\{\star\}$
$F_{i\geq 3}$	$\mathcal{K}_i$	$\mathcal{K}_i$
F_ω	$\mathcal{K}_\omega$	$\mathcal{K}_\omega$

System F_1 does neither allow quantification over types nor type operators; so it is the simply typed lambda calculus. System F_2 adds quantification over types, i.e. it is System F, the second-order lambda calculus. F_3 adds first-order type operators. F_4 adds second-order type operators, and so on.

3 Dependent Types

In order to use dependent types it is necessary to generalize function types $\mathsf{S} \to \mathsf{T}$ to dependent function types $\Pi \mathsf{x} : \mathsf{S}\,.\,\mathsf{T}$ first. In comparison to arrow types, the result type T now may depend on the argument x. This allows us to write funny types like:

$$\Pi \mathsf{n}{:}\mathsf{Nat}\,.\,\mathsf{if}\ \mathsf{n} > 3\ \mathsf{then}\ \mathsf{Nat}\ \mathsf{else}\ \mathsf{Bool}$$

A function of this type returns a natural number if the argument is greater than 3. Otherwise it returns a boolean! We can define arrow types in terms of dependent function types:

$$\mathsf{S} \to \mathsf{T} \quad \stackrel{\mathsf{def}}{=} \quad \Pi \_{:}\mathsf{S}\,.\,\mathsf{T}$$

I.e. arrow types correspond to the degenerate case of dependent function types where the result type does not mention x.

In a setting with dependent types we could have – in addition to a non-dependent type NatList of lists over integers – a dependent type

$$\mathsf{DNatList}{::}(\mathsf{NatList} \to \mathsf{Bool}) \to \star$$

which, given a predicate over NatLists, returns the type of NatLists fulfilling the predicate. In combination with a predicate

$$\mathsf{isSortedVersion}{:}\mathsf{NatList} \to \mathsf{NatList} \to \mathsf{Bool}$$

and dependent function types this allows us to write the type

$$\Pi \mathsf{l}{:}\mathsf{NatList}\,.\,\mathsf{DNatList}\ (\mathsf{isSortedVersion}\ \mathsf{l})$$

of sorting functions for lists over natural numbers. In general dependent types give us the desirable possibility to state the entire correctness property of a purely functional program in its type.

However, proving the correctness of a program without writing extensive annotations is computationally intractable. Since writing extensive annotations is not worth the effort in most cases, most programming languages do not support dependent types.

Like already mentioned in the introduction, some programming languages nonetheless support very restricted forms of dependent types [7, 10, 1, 21, 22, 20, 11] and other restricted forms are used in module systems [15]. These applications restrict dependent types so that type-checking does not involve arbitrary computations on the term-level; in order to keep type-checking simple and feasible.

$$
\begin{array}{rll}
\mathsf{E} ::= & \mathsf{K} & \text{(constants)} \\
\mid & \mathsf{x} & \text{(variables)} \\
\mid & \lambda\mathsf{x}{:}\mathsf{E}\,.\,\mathsf{E} & \text{(abstraction)} \\
\mid & \mathsf{E}\;\mathsf{E} & \text{(application)} \\
\mid & \Pi\mathsf{x}{:}\mathsf{E}\,.\,\mathsf{E} & \text{(dependent function type)}
\end{array}
$$

Fig. 5. The syntax of Barendregt's λ-cube.

4 The $\boldsymbol{\lambda}$-cube

In system F_ω we have three sorts of expressions: terms, types and kinds. Apart from that we have two sorts of types and typing relations: usual types and kinds. This is quite a lot.

Figure 5 shows the syntax of Barendregt's λ-cube [2], where all sorts of expressions are collapsed into one syntactic category. Furthermore the λ-cube has just one typing relation for all sorts of expressions. In order to type kinds, the λ-cube provides an additional sort of expressions with just one constant: the type $\Box$ of kinds. So in the λ-cube the following typing statements hold:

$$
\begin{aligned}
\Gamma &\vdash 5 : \mathsf{Nat} \\
\Gamma &\vdash \mathsf{Nat} : \star \\
\Gamma &\vdash \star : \Box
\end{aligned}
$$

The λ-cube has the same expressive power as the calculus of constructions [5, 13], i.e. system F_ω plus dependent types. In order to understand this, we need to understand how non-dependent function types and universal types can be expressed with dependent function types provided by the λ-cube. We already defined non-dependent function types in terms of dependent ones in the previous section.

So what is a universal type? It is the type of a function from types to terms. Moreover it is a dependent function type, because the result type may depend on the argument type, like in the polymorphic identity function:

$$\vdash \lambda\mathsf{X}{:}\star\,.\,\lambda\mathsf{x}{:}\mathsf{X}\,.\,\mathsf{x} : \forall\mathsf{X}{:}\star\,.\,\mathsf{X} \to \mathsf{X}$$

So in order to express universal types in the λ-cube we simply have to replace $\forall$ by $\mathsf{\Pi}$:

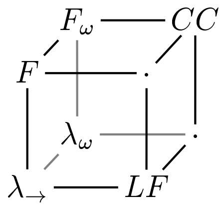
$$\forall\mathsf{X}{:}\mathsf{K}\,.\,\mathsf{T} \quad \stackrel{\mathsf{def}}{=} \quad \mathsf{\Pi}\mathsf{X}{:}\mathsf{K}\,.\,\mathsf{T}$$

The λ-cube not only has the expressive power of the calculus of constructions. Additionally it provides a simple mechanism for restricting it to several less expressive type systems; one of its typing rules (the rule PI, see [12, Section 3] for details) uses a *judgment relation* $\rightsquigarrow$ which tells what kinds of abstractions are allowed:

$\star \rightsquigarrow \star$	terms can be abstracted by terms
$\Box \rightsquigarrow \star$	terms can be abstracted by types
$\Box \rightsquigarrow \Box$	types can be abstracted by types
$\star \rightsquigarrow \Box$	types can be abstracted by terms

So $\rightsquigarrow = \{\star \rightsquigarrow \star\}$ instantiates the λ-cube to the simply-typed lambda calculus. Adding $\Box \rightsquigarrow \star$ adds abstractions of terms by types, i.e. instantiates the λ-cube to System F. Adding $\Box \rightsquigarrow \Box$ adds type operators, so that we get System F_ω. Finally, the calculus of constructions can be obtained by adding the last element $\star \rightsquigarrow \Box$.

Since type systems without term-level abstractions are pointless, the judgment relation will almost always contain $\star \rightsquigarrow \star$. However, the remaining three judgments allow the instantiation to eight different type systems, which are summarized in the following diagram[3]:



Type systems on the top provide polymorphism, those on the back provide type operators and those on the right provide dependent types.

5 Summary

In section 2 we introduced the higher-order polymorphic lambda calculus, showed that its typing relation is decidable and gave examples for higher-order types and higher-order polymorphism.

[3] $\lambda_\rightarrow$ is the simply typed lambda calculus. λ_ω is $\lambda_\rightarrow$ plus type operators. LF, the *logical framework* [9], is $\lambda_\rightarrow$ plus dependent types. CC is the calculus of constructions [5, 13], i.e. the unrestricted λ-cube.

In section 3 we had a brief look into (unrestricted) dependent types and saw that they provide the possibility to express every imaginable type, especially types of correct programs like the type of sorting functions. Since type-checking of unrestricted dependent types is computationally untractable, only restricted forms are used in practice.

Last but not least we saw how we can put several type systems considered so far into a nutshell: the λ-cube. It provides just one syntactic category of expressions with a only a hand full of variants. Nonetheless it can be instantiated to such powerful calculi as System F, System LF or the calculus of constructions.

References

1. Lennart Augustsson. Cayenne – a language with dependent types. *International Conference on Functional Programming Languages (ICFP)*, pages 239–250, 1998.
2. Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.
3. Bruno Barras et al. *The Coq proof assistant reference manual: Version 6.1*, 1997.
4. Robert L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.
5. Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
6. Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.
7. James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, 1985.
8. Jeans-Yves Girard. *Interprétation fonctionnelle et élimination des courpures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.
9. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992.
10. J.G. Hook. Understanding Russel – a first attempt. *Proc. Int. Symp. on Semantics of Data Types*, 1984.
11. C. Barray Jay and Milan Sekanina. Shape checking of array programs. *Computing: The Austalasion Theory Seminar (Proceedings)*, 19:113–121, 1997.
12. Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language, 1997.
13. Zhaohui Luo. Computation and Reasoning: A Type Theory for Computer Science. In *Number 11 in International Series of Monographs on Computer Science.* Oxford University Press, 1994.
14. Zhaohui Luo and Robert Pollack. *The LEGO proof development system: A user's manual*, May 1992.
15. David MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 277–286, 1986.

16. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag, 1994.
17. Frank Pfenning. Elf: a meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815. Springer-Verlag, June 1994.
18. Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, first edition, 2002.
19. Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.
20. Hongwei Xi and Robert Harper. A dependently typed assembly language. *International Conference on Functional Programming Languages (ICFP)*, 2001.
21. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Conference on Programming Language Design and Implementation (ICFP)*, 1998.
22. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. *ACM Symposium on Principles of Programming Languages (POPL)*, pages 214–227, 1999.