Axiomathes (2015) 25:61–77
DOI 10.1007/s10516-014-9260-9

ORIGINAL PAPER

An Overview of Type Theories

Nino Guallart

Received: 16 May 2014 / Accepted: 7 November 2014 / Published online: 15 November 2014

Abstract Pure type systems arise as a generalisation of simply typed lambda calculus. The contemporary development of mathematics has renewed the interest in type theories, as they are not just the object of mere historical research, but have an active role in the development of computational science and core mathematics. It is worth exploring some of them in depth, particularly predicative Martin-Löf's intuitionistic type theory and impredicative Coquand's calculus of constructions. The logical and philosophical differences and similarities between them will be studied, showing the relationship between these type theories and other fields of logic.

Keywords Higher-order logic · Type theory · Intuitionistic logic · Lambda calculus · Foundations of mathematics

1 Introduction

This paper is an overview of generalised type systems, in particular normalising dependent systems, focusing on a comparison between predicative and impredicative dependent theories. It is intended as a very basic introduction, so no previous knowledge of the topic is assumed. The first part of the paper is dedicated to preliminary notions of λ-calculus and simple type theory, and the second one to the questions regarding generalised type systems.

Type theories arose as an alternative to set theory due to some contradictory situations that appear in naïve set theory when considering certain definitions of sets. The most important example is Russell's paradox, which involves the set of all sets that are not members of themselves, $R = \{x | x \notin x\}$ (Cfr. Russell 1903, 1980). It

N. Guallart (✉)
Universidad de Santiago de Compostela, Santiago de Compostela, Spain
e-mail: nino@usal.es

is clear that this set is a member of itself if and only if it is not a member of itself, which is contradictory. The discovery of this paradox revealed a fatal flaw in naïve logic and forced to reconsider their basic principles. Logicians and mathematicians tackled this problem by restricting the way sets can be formed with two basic approaches:

- Formulation of axiomatised set theories (ZFC, NBG…). Instead of relying on a principle of unrestricted comprehension, these systems have a version of an axiom or rule of separation, which needs a previously defined set in order to build a new one from it and a certain predicate ranging over its elements.
- Type theories. In layman's terms, a type is almost the same thing as a set, except that types form a hierarchy that avoids self-reference, since a type contains elements of a lower range. In this paper we will cover some aspects of the development of type theories.

Self-reference, which plays a crucial part in many paradoxes like this, is closely related to impredicativity; a definition is impredicative if it quantifies over a set containing the entity being defined, and predicative otherwise. Russell's paradox is impredicative, and some authors like Russell himself or Poincaré thought that impredicativity was problematic, arguing that impredicative definitions lead to a vicious circle. However, many basic mathematical definitions are impredicative (e.g. least upper bound) and Ramsey (Cfr. 1978) stated that many impredicative definitions are actually harmless and non-circular. Predicative mathematics avoids any problem regarding impredicativity by relying only on predicative principles, but the rejection of impredicative definitions leaves out some very important ones, such as power sets, making thus the task of predicative mathematics quite difficult. The the vast majority of mathematicians have continued being impredicative, but we will see later how new predicative theories emerge again in generalised type theories.

Although several authors such as Russell, Ramsey or Gödel created their own versions of type theory, probably the most remarkable one is Church's system. Its main feature is that it is formulated in λ-calculus. From now on we will focus on type theories based on it, that is, typed λ-calculus. In order to understand them properly, we will study some basic notions of untyped λ-calculus, before dealing with typed theories.

2 Simple Type Theory

2.1 Untyped Lambda Calculus

Lambda calculus (λC) is a formal system created by Church in order to deal with the notions of recursion and computability in connection to the problem of halting (*Entscheidungsproblem*). Instead of relying on elements and sets as primitives, it uses λ-terms, which are used to explore the concept of function.

The set of λ-terms, or just terms in short, is defined recursively from a countable set of variables *Var*. Using Backus–Naur notation, formation of terms can be expressed in the following scheme, where x is any variable, E and F are arbitrary terms and the dots denote optional constant terms:

$$E, F := x|\lambda x \cdot E|(EF)|\ldots$$

In other words, the set Λ of terms can be defined recursively as follows:

- If $x \in Var$, $x \in \Lambda$.
- If $E \in \Lambda$, $F \in \Lambda$, then $(EF) \in \Lambda$.
- If $E \in \Lambda$ and $x \in Var$, $\lambda x \cdot E \in \Lambda$.

Simplification rules for parentheses can be applied as usual: outer parentheses are omitted and left association is assumed, so *EFG* stands for ((*EF*)*G*); abstraction is right associative, so $\lambda x \cdot \lambda y \cdot E$ stands for $\lambda x \cdot (\lambda y \cdot E)$; abstractions can be expressed in contracted mode, for example $\lambda xy \cdot E$ instead of $\lambda x \cdot \lambda y \cdot E$.

Variables may occur bound in a term by the λ abstraction or be free: in $\lambda x \cdot xyx$ is bound and y is free. A basic and important operation is the substitution of variables for terms. We will write $P[x := t]$ to indicate the substitution of the variable x in P for the term t. For example, $xyx[x := z]$ and $xyx[x := ab]$ gives out zyz and $abyab$ respectively, since all x have been substituted by z and ab. Substitution is the basis of α-conversion and β-reduction.

α-Conversion Variable names are conventional, so any variable in a term can be renamed, that is, substituted by another variable which does not occur in the term. This is called α-conversion: $\lambda x \cdot x$ can be converted to $\lambda y \cdot y$, so they are equivalent under α-conversion, $\lambda x \cdot x \rightarrow_\alpha \lambda y \cdot y$. Since the new variables must not occur in the term, $\lambda x \cdot xy \rightarrow_\alpha \lambda y \cdot yy$ is wrong, whereas $\lambda x \cdot xy \rightarrow_\alpha \lambda z \cdot zy$ is correct.

β-Reduction and Operational Semantics of λ C Term evaluation in λC is based on β-reduction, the application of functions over their arguments substituting the bound variables in the function by their corresponding arguments (actual process may be more complex, but for the sake of simplicity this work will focus solely on β-reduction as the basis of term evaluation):

$$(\lambda x \cdot E)F \Longrightarrow E[x := F]$$
$$(\lambda x \cdot xy)a \Longrightarrow xy[x := a] \Longrightarrow ay$$

After a β-reduction, a term may or may not need ulterior reduction. A reducible subterm within a term is usually called a reducible expression or a redex in short. If a term cannot be reduced further, it is said to be in its normal form, or to be a value. In λ-calculus, any term evaluation converges to a value or diverges. Untyped lambda calculus is not normalising, that is, it is not always possible to reach a value.

Example Identity function $I := \lambda x \cdot x$, when applied to a term, gives the same term as an output:

$$(\lambda x \cdot x)e \Longrightarrow x[x := e] \Longrightarrow e$$

Since I is also a term, it can be applied to itself, giving itself as an output:

$$II \Longrightarrow I$$
$$(\lambda x \cdot x)(\lambda x \cdot x) \Longrightarrow x[x := \lambda x \cdot x] \Longrightarrow \lambda x \cdot x$$

Example ω operator $\lambda x \cdot xx$ takes an input e and gives ee as an output; this new term may or may not be reduced again. If applied to itself, the result is the same as the original term and therefore a value is never reached.

$$\omega\omega \Longrightarrow \omega\omega$$
$$(\lambda x \cdot xx)(\lambda x \cdot xx) \Longrightarrow xx[x := \lambda x \cdot xx] \Longrightarrow (\lambda x \cdot xx)(\lambda x \cdot xx)$$

Example Sometimes, the reduction of a term gives a more complex term and thus a value is never reached:

$$(\lambda x \cdot xxx)(\lambda x \cdot xxx) \Longrightarrow (\lambda x \cdot xxx)(\lambda x \cdot xxx)(\lambda x \cdot xxx) \Longrightarrow$$
$$\Longrightarrow (\lambda x \cdot xxx)(\lambda x \cdot xxx)(\lambda x \cdot xxx)(\lambda x \cdot xxx) \Longrightarrow \cdots$$

Different strategies can be applied to reduce a term. The normal way of reducing terms is named *call-by-value*, which means reducing the rightmost redex first. The opposite strategy, reducing the leftmost redex first, is named *call-by-name*. There are many other strategies. However, if the term eventually reaches a normal form, it holds the Church–Rosser property, that is, the normal form is independent of the strategy that has been chosen. It is worthwhile noting that this semantics of λC is operational, since it describes the way λ-terms function, not what they denote.

Recursion in λC is possible using fixed point combinators, which are terms R such as, when over another term T, give

$$RT \Longrightarrow T(RT) \Longrightarrow T(T(RT)) \Longrightarrow \cdots$$

If we make $Q := RT$ then $Q := TQ$, which is a fixed point, hence its name. There are infinite fixed point combinators, although the most known is Curry's Y combinator.

The following closing remarks can be made:

- Lambda calculus' foundations rely on the use of abstraction and application: $\lambda x \cdot P$ is an anonymous function which takes an input x and gives $P(x)$ as an output. The application PQ is the result of considering term Q as the input of P.
- Recursion allows the construction of complex formulae and is the basis of the computational power of lambda calculus.
- λC is equivalent to a Turing machine, and therefore is an abstract model of a programming language (Landin 1965). In λC, Boolean values are possible and natural numbers can be defined in the form of Church's numerals, and there are

also logical and arithmetical operations over them and flux control operators. It is not decidable, so in principle it is not possible to know whether the evaluation of a term will end or not. Whereas TM is the model of imperative programming languages, λC is the basis of functional ones.

2.2 Simply Typed Lambda Calculus

Simply typed lambda calculus was also originally developed by Church (1940, 1941). It is a higher order logic system based on lambda calculus and it uses the same syntax. We will not see the original formulation of Church here, but a more recent one which will allow us to connect with current developments.

A simply typed lambda calculus (STLC) has a non-empty set of base types. The rest of the types, which are function types, can be built from them by the application of the type constructor $\rightarrow$. Types and terms can be defined by recursive rules, where E and F are any terms, σ and τ any types, x any variable, β a base type, and the dots denote optional terms belonging to a given type:

$$\begin{aligned} \sigma, \tau &:= \beta | \sigma \rightarrow \tau \\ E, F &:= x | \lambda x : \sigma \cdot E | EF | \ldots \end{aligned}$$

$\lambda x : \sigma \cdot E$ is also written $\lambda x^{\sigma} \cdot E$, the first form will be preferred, although the second one will be used sometimes in order to increase clarity. There are two styles of typing, Church style and Curry style. Although in a subtle way, their differences are not only syntactical, but also semantical. In Curry style, variable types are not explicitly stated, whereas in Church style it is imperative to declare the type of every variable. The problems with Curry style will not be discussed here, and this entire section refers only to Church style. The following paragraphs will focus on the difference between well-formed terms (actually they are called *pre-terms*) and well-typed terms.

In typed theories there are *typing environments*, also called variable assignments, which are (possibly empty) sets of associations between types and variables, so each variable has its type, and we write $x : \sigma$. In a typing environment, a *typing judgment* $\Gamma \vdash E : \sigma$ is a statement of the fact that, under environment Γ, term E is well-formed and has type σ. In other words, it asserts that E is a term of type σ if and only if its free variables are of the types specified in the typing environment.

Typing Rules The associations between terms and types are maintained by *typing rules*, which are inference rules from a group of premises to a conclusion, all of them being typing judgments. The following typing rules specify how to assign a type to a certain well-formed syntactic construction, where VAR stands for variable (a variable within a context is a well-typed term), ABS for abstraction (given a variable and a well-typed term, the λ-abstraction of the term is well-typed) and APP for application (the application of two well-typed terms is also well-typed):

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\,\text{VAR} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma . e) : (\sigma \to \tau)}\,\text{ABS}$$

$$\frac{\Gamma \vdash e : \sigma \to \tau \qquad \Gamma \vdash f : \sigma}{\Gamma \vdash ef : \tau}\,\text{APP}$$

In STLC well formed terms may be not well-typed, and the typability of terms are given by the previous rules, so the set of well-typed terms is a proper subset of the set of all well formed terms. For example, $(\lambda x : \sigma \cdot x)(y : \tau)$ is well formed since it can be built following the mentioned syntactic rules of formation, but not well-typed, because it does not satisfy rule APP, since $\lambda x : \sigma \cdot x$ has type $\sigma \to \sigma$ and y has type τ. In this case, its evaluation gives as an output an error.

Provided that the types are right, term evaluation is based on β-reduction, like in untyped λ-calculus:

$$\frac{(\lambda x : \sigma . E)F}{E[x := F]}\,\text{E-APP}$$

$$\frac{t \to t'}{tu \to t'u}\,\text{E-APP1} \qquad \frac{t \to t'}{ut \to ut'}\,\text{E-APP2}$$

We can also consider the void type 0 with no terms and the unit type with a single term, $* : 1$ such as for any type T, it holds $\lambda x^T \cdot * : 1$. Since type 0 has no terms, it is also easy to see that we can not construct a valid term of type $T \to 0$.

STLC can be extended to STLC with pairs, in which the product type $\sigma \times \tau$ appears, which is the type of pairs of terms (s, t). The notation for types and terms in a system with two type constructors, $\to$ and $\times$ is the following:

$$\sigma, \tau := \beta | \sigma \to \tau | \sigma \times \tau$$
$$E, F := x | \lambda x : E\sigma \cdot F | EF | (E, F)$$

The rules for the formation and elimination of pairs are the following:

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash f : \tau}{\Gamma \vdash (e, f) : \sigma \times \tau}\,\text{I-PAIR}$$

$$\frac{\Gamma \vdash (e, f) : \sigma \times \tau}{\Gamma \vdash \pi_1(e, f) : \sigma}\,\text{E-PAIR L} \qquad \frac{\Gamma \vdash p : \sigma \times \tau}{\Gamma \vdash \pi_2(e, f) : \tau}\,\text{E-PAIR R}$$

π_1 and π_2 are respectively the first and second projection of the pair, so $\pi_1(e,f) = e$ and $\pi_2(e,f) = f$. Both STLC and STLC with pairs are strongly normalising, that is, the evaluation of a term eventually gives a value if it is well-typed or an error if not, and therefore they are decidable. In this way, STLC and its derivatives may serve as a basis for type checking, which has its practical, computational applications. Not all well-formed terms are well-typed, unlike untyped λC, which has no typing restrictions. Because of it, STLC and STLC with pairs are not Turing-complete, since they are not equivalent to λC, and they are less expressive than it.

Type Theories and Category Theory Until now, we have considered syntactical and operational aspects of type theories, neglecting their semantical facet. Now we are going to see the semantical application of category theory to type theories. Category theory was developed by Eilenberg and MacLane (1945) for the study of algebraic topology, but it soon showed to be useful in many other fields. Its connection to logic has been studied since the works of Lawvere (1963); in this paper we will also follow the works of Lambek and Scott (1988). Lawvere and other authors have also studied category theory as a new approach to the foundations of mathematics, dealing with abstract mathematical structures, instead of using the traditional set-theoretical frame.

A category $\mathcal{C}$ is made of objects and morphisms or arrows between objects. Each morphism has as a domain a given object and as codomain another one, so for example morphism $f : A \rightarrow B$ has $dom(f) = A$ and $cod(f) = B$. The composition of arrows is associative, $(fg)h = f(gh)$, and for every object X there is an identity arrow $1_X : X \rightarrow X$ such as for an arbitrary pair of morphisms $g : Y \rightarrow X$ and $f : X \rightarrow Z$, $f1_X = f$ and $1_X g = g$. There can be additional compositions of objects that will be useful later, such as the product $X \times Y$ of two objects X and Y, which is an object $Z = X \times Y$ with two projection morphisms π_1 and π_2 satisfying $\pi_1 Z = X$ and $\pi_2 Z = Y$; the exponential or power object X^Y can be seen as the class of all morphisms from Y to X (the actual definition is more complex, but for the purposes of this paper, this definition is valid). A category has a given structure that can be simple or complex, and we can be consider categories made of categories. We call *functors* the mappings between these categories preserving the structure from one category into the other.

There is a bidirectional relationship between categories and type theories (Cfr. Asperti and Longo 1991). Type theories can be interpreted using category theory, and conversely, we can formalise categories in the language of type theories. Generally, we can say that the relation between a type theory and its corresponding category is akin to *syntax vs semantics*. We can show this studying the relationship between STLC with pairs and Cartesian closed categories (Cfr. Lambek and Scott 1988).

A category is a Cartesian closed category (CCC) if and only if it has a terminal object T such as for every object X there is a unique morphism $X \rightarrow T$, and if for any two objects X and Y there exists the product $X \times Y$ and the exponential object X^Y. We can see that the elements of STLC $\mathcal{T}$ form a CCC: we can consider base types of $\mathcal{T}$ as objects, and it is easy to see that type 1 can be interpreted as the terminal object T. Constant terms a of a given type A can be seen as morphisms $a : 1 \rightarrow A$ from the terminal object to their corresponding object A, and more broadly, terms can be interpreted as morphisms, whereas application of terms equates to composition of morphisms. For every object A there is an identity function 1_A which can be seen as the identity morphism for A, and it can be proved that composition of terms is associative. Product type $A \times B$ equates to the product of objects, and function type $A \rightarrow B$ is interpreted as the power object B^A, which is the class of equivalence of all morphisms from A to B with a free variable of type A. Therefore, we have that the interpretation of $\mathcal{T}$, $Syn(\mathcal{T})$, is a category, and more precisely a CCC which we can call $\mathcal{C}'$.

Table 1 Correspondence between CCC and STLC

Type theory	Category theory
Types	Objects
Unit type ($\top$ or 1)	Terminal object
Product type $A \times B$	Product of objects $A \times B$
Function type $A \to B$	Exponential object B^A
Terms	Morphisms
Pair of terms (f, g)	Pair of morphisms (f, g)
Projections of terms, π_1 and π_2	Projections of morphisms, π_1 and π_2
Abstraction $\lambda x^A \cdot f : B$	Arrow $f : A \to B$ with a free variable $x : A$
Application fg	Composition of arrows fg

Conversely, we can see that the structure of a given CCC $\mathcal{C}$ can be described using the language of a STLC $\mathcal{T}' = Lang(\mathcal{C})$, where $Lang(\mathcal{C})$ is the smallest type theory that preserves the structure of the category, that is, $\mathcal{T}'$ is the internal language of $\mathcal{C}$ (Cfr. Johnstone 2003). We can denote the terminal object T with the unit type, and assign a type to each object in the category. Product of objects and exponential objects equate to product types and function types respectively, and for each morphism f between objects X and Y, there is a term $\lambda x^X \cdot f$ with a type $X \to Y$.

Actually, this relation between categories and type theories is subtler, since given a CCC $\mathcal{C}$ and a STLC $\mathcal{T}$ with the same structure, the internal language of the category $Lang(C)$ is not equal to $\mathcal{T}$, but homomorphical to it ($Lang(\mathcal{C}) \cong \mathcal{T}$), and the same with the semantical interpretation of the theory, $Syn(\mathcal{T}) \cong \mathcal{C}$. Syn and $Lang$ are actually two functors between categories, so we can succinctly write their relationship in this way: $\mathcal{C} \underset{Lang}{\overset{Syn}{\leftrightarrows}} \mathcal{T}$ (Table 1).

3 The Lambda Cube and Generalised Type Systems

From the 70s onwards there have been prominent new works in the field of type theories. A complex family of typed lambda systems, each with their own features, flaws and strengths, has been developed from Church's simple type theory. There are several reasons that explain this renewed interest. Firstly, some practical reasons: type theory is closely related to proof theory and therefore to computing and typechecking. Secondly, from a theoretical point of view, the rise of category theory and further developments into their connections to logic have been the object of intensive research, as type theories can be studied from a categorical point of view. In relation to this, there have been renewed efforts in the research of type theories as an alternative foundation of mathematics, particularly in constructive mathematics, mainly since the works of Per Martin-Löf, and new fields within logic and mathematics such as homotopical type theory have appeared, advancing this study.

Pure Type Systems STLC can be considered as the basis of a family of type systems that have been named pure type systems (PTS) or generalized type systems (GTS).

Fig. 1 Barendregt's lambda cube

$$\begin{tikzcd}
 & \lambda_\omega \arrow[rr] & & \lambda_{P\omega} \\
\lambda_2 \arrow[ur] \arrow[rr] & & \lambda_{P2} \arrow[ur] & \\
 & \lambda_{\underline{\omega}} \arrow[uu] \arrow[rr] & & \lambda_{P\underline{\omega}} \arrow[uu] \\
\lambda_\to \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda_P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

They are a group of type systems that, unlike STLC, allow dependencies between types and terms. Broadly speaking, the main difference between a simply typed theory and a pure type system is that the latter allows judgments over types. In STLC types and terms are two disjoint groups, in PTS this distinction is blurred or erased.

Instead of creating confusing categories such as types of types, types of types of types and so on, the concepts of kind and sort are used instead, which are generalisations of the notion of type. $\bigstar$, also called *Prop*, is any usual type of terms or propositions. $\square$ or *Type* denotes the higher-order type of a type. The set of sorts is thus $S = \{\bigstar, \square\}$. A kind $\bigstar \to \square$ maps from a type of terms to a certain kind of types and so on. In a given context, it can be asserted that $\sigma : \square$ for specifying that σ is a valid type in the same way that $e : \sigma$. The scheme for kinds and expressions in PTS is the following, where V is any variable and S any sort and K any kind (Π operator will be explained later):

$$K := \bigstar | \square | K \to K$$
$$T, U := V | S | TU | \lambda V : T \cdot U | \Pi V : T \cdot U$$

Not all PTS are normalising, that is, when evaluated not all of them reach a value. In the next subsections we will study a more reduced family of generalised type theories, all of which are decidable and normalising.

3.1 Barendregt's Lambda Cube

Barendregt (1991) considers four possible relations between terms and types: terms depending on terms, types on types, terms on types and types on terms. If we omit the first one, we have three possibilities that can be represented as axes of a cube, other features such as subtyping could be represented in additional dimensions (also Cfr. Barendregt 1992) (Fig. 1).

These three possibilities considered by Barendregt are the following:

- Type polymorphism (terms depending on types, $\Box \to \bigstar$): universal quantification over types in order to be able to define type variables. System $\lambda 2$, also called system F, which was discovered independently by Girard (1972) and Reynolds (1974), is a second order lambda calculus or polymorphic lambda calculus.

Example In simple type theory there is no unified identity function, unlike $I := \lambda x \cdot x$ of untyped λC. Instead of a single function, in STLC there is a family of identity functions, one function $\lambda x : \sigma \cdot x$ for each type σ. If quantification over types is allowed, then the previous idea can be formalised succinctly in the way system F does:

$$\Lambda\alpha \cdot \lambda x^{\alpha} \cdot x : \forall\alpha \cdot \alpha \to \alpha$$

Analogously to the way simply typed lambda calculus defines types and terms, so does system F, being β a base type and α any type variable:

$$\begin{aligned} \sigma, \tau &:= \beta | \alpha | \sigma \to \tau | \forall\alpha \cdot \sigma \\ E, F &:= x | EF | \lambda x : \sigma \cdot E | \Lambda\alpha \cdot E \end{aligned}$$

Unlike in STLC, types can appear in terms, like the case of the previous example $\Lambda\alpha \cdot \lambda x^{\alpha} \cdot x$.

Two new rules for introduction ($\forall I$) and elimination ($\forall E$) of generalisation over types:

$$\frac{\Gamma \vdash M : \tau \qquad \alpha \notin \Gamma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau} \forall \text{ I}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma \qquad \Gamma \vdash T : \tau}{\Gamma \vdash MT : \sigma[\alpha := \tau]} \forall \text{ E}$$

- Type constructors (types depending on types, $\Box \to \Box$): abstraction of new types from previous ones. This is a remarkable implementation, because new types are built within the language, not the metalanguage of kinds (Roorda 2000). There will be rules for kinds and expressions (which comprise both types and terms) instead of for types and terms. λ_{ω} calculus (Girard 1972) and system $F_{\omega}/\lambda 2_{\omega}$ use type constructors.

Rules for this implementation can be quite complex and they will not be covered here. Product type is a simple example of this feature. Other type constructors such as list constructors or higher order types are within this category.

- Dependent types (types depending on terms, $\bigstar \to \Box$): The last possibility is building types depending on previous types. Here are two possibilities, product types and dependent sum (or pair) types. Product types (Π-types) generalise the idea of universal quantification. Considering a non-empty type A that will serve as an index, Π-types generate a family of types $B(a)$ depending on every $a \in A$.

Table 2 Summary of the logical systems in the lambda cube

System	Relations	Examples
$\lambda_{\rightarrow}$	$\bigstar \rightarrow \bigstar$	STLC
λ_2	$\bigstar \rightarrow \bigstar, \Box \rightarrow \bigstar$	System F
$\lambda\underline{\omega}$	$\bigstar \rightarrow \bigstar, \Box \rightarrow \Box$	Weak λ_ω
λ_ω	$\bigstar \rightarrow \bigstar, \Box \rightarrow \bigstar, \Box \rightarrow \Box$	System Fω
λP	$\bigstar \rightarrow \bigstar, \bigstar \rightarrow \Box$	System LF
λP_2	$\bigstar \rightarrow \bigstar, \bigstar \rightarrow \Box, \Box \rightarrow \bigstar$	λP_2
$\lambda P_{\underline{\omega}}$	$\bigstar \rightarrow \bigstar, \bigstar \rightarrow \Box, \Box \rightarrow \Box$	Weak λP_ω
λP_ω	$\bigstar \rightarrow \bigstar, \Box \rightarrow \bigstar, \bigstar \rightarrow \Box, \Box \rightarrow \Box$	CoC

If $B(x)$ is a constant type B, then $\Pi_{x:A}B(x)$ equals $A \rightarrow B$. Conversely, sum types (Σ-types) in the form $\Sigma_{x:A}B(x)$, are the type of pairs of terms of the form $(a^A, b^{B[x:=A]})$ in which the type of the second element depends on the value of the first term. Sum types are the equivalent of existential quantification and, if B is a constant type, $\Sigma_{x:A}B(x)$ equals $A \times B$. Two examples of systems with dependent types are system LF, which is STLC with dependent types, and calculus of constructions, that will be covered soon.

To sum up (Table 2):

All application and abstraction rules of these systems can be summarised as follows if we consider the whole cube system (Cfr. Roorda 2000).

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : t \in \{\star, \Box\}}{\Gamma \vdash (\lambda x : A.b) : \Pi x : A.B}\ \text{ABS}$$

$$\frac{\Gamma \vdash f : (\Pi x : A.B) : t \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := A]}\ \text{APP}$$

Calculus of constructions, which will be mentioned later, allows all of these extensions and, therefore, it is placed in the uppermost right back the cube. Other systems such as system F only allow some of them, being placed in other corners. All of them are strongly normalising, that is, end giving a value or an error in a finite number of steps.

Semantically, 2-categories (categories over morphisms) or more generally n-categories and higher-order type theories are related in a way similar to the link between CCC and STLC. Semantics of these type theories is far more complicated than the semantics of STLC and, therefore, just a brief sketch will be made. The starting point of the categorical interpretation of dependent types is the consideration of slice categories, which are categories in which the objects are morphisms over a given object. It is easy to see the relationship within slice categories and dependent types, since an object A in which every morphism a gives out a slice category $B(a)$ can be considered the semantical interpretation of the product type

$\Pi x : A \cdot B$. *Locally closed Cartesian categories* (lCCC) are categories in which all slice categories are CCC and a kind of dependent type theory is the internal language of lCCC (Seely 1984), in the same way that STLC with pairs is the internal language of CCC.

3.2 Calculus of Constructions

Coquand's calculus of constructions (CoC in short) is a higher-order lambda calculus theory that combines polymorphism and type construction of Girard's system F_ω with dependent types. The syntax of its kinds and terms is the following:

$$K := \bigstar | \Box_i \ (i \geq 1)$$
$$\sigma, \tau, M, N := x | K | \Pi x^\sigma \cdot \tau | \lambda x^\sigma \cdot M^\tau | MN_{\tau[x:\sigma]}$$

CoC distinguishes between the impredicative type of predicates ($\bigstar$, small types), a predicative hierarchy of types of types ($\Box_i$, large types), and the type of all large types. As in other PTS, from Π constructor it is possible to derive the usual logical operators. CoC also has void type and 1 type, and it is easy to create types for truth values and natural numbers. It has several variations, such as CoC with inductive types, but they will not be treated here.

So far, all considered type theories are impredicative. None of them are problematic, since they are strongly normalising. For example, the polymorphic identity in System F $\Lambda\alpha\lambda x : \alpha \cdot x$ can take as arguments its own type $\forall\alpha\cdot \to \alpha$ and then itself, but in a way that leads to no circularity. Some impredicative theories such as Girard's system U are inconsistent, but this is not the case of the type systems considered in the λ-cube.

3.3 Intuitionistic Type Theory

Per Martin-Löf's intuitionistic type theory (ITT in short) allows to introduce contemporary predicative theory types in this discussion. It can be considered and extension of STLC with higher order predicates and quantification over types within a mathematical constructivist programme. Strictly speaking, ITT does not belong to the λ-cube, but it has an expressive power similar to the one of CoC, so a brief comparison between them seems reasonable. Semantically, Seely (1984) showed that there is a relationship between locally Cartesian closed categories and ITT. It has shown some prominent features in the field of programming due to its connections to proof theory, and it also aims to serve as a constructivist theory for the foundations of mathematics. Homotopy type theory expects to follow this aspiration, since the programme of the Univalent Foundations of Mathematics conceive this field as an extension of ITT with a homotopical interpretation (Voevodsky et al. 2013).

As has been said before, previous type theories are impredicative, yet this feature is unproblematic. ITT's first formulation was also impredicative, but was soon discovered to be inconsistent. Later developments (Martin-Löf 1975) avoided the problems of this earlier version with a predicative formulation, a common feature in other constructivist approaches to the foundations of mathematics. Categorically, predicative theories have a more general structure; whereas many impredicative theories lie on topos theories, predicative ones rely on structures such as pretopos, which are more broad categories but require more complex proofs, since impredicative definitions are rejected.

The concept of universe in ITT, which is similar to Grothendieck's universe, is crucial in this theory, which is in consonance with the constructive theses of its author. It can be considered as a closed type of types built according to certain conditions. An earlier version of ITT had an impredicative universe, but it showed to be inconsistent. Later versions (Cfr. Martin-Löf 1975) consider a hierarchy of predicative universes $(U_i, Type_i)$ or (U_i, T_i) in which a given $Type_i$ has a type $Type_{i+1}$.

$$\frac{x \in U_i}{x : Type_i}\ \text{U-FORM}$$

$$\frac{x \in U_i}{Type_i : Type_{i+1}}\ \text{U-FORM}$$

Like CoC, ITT uses dependent types, and universes are closed under operations. Intuitionistic type theory can be formalised stating its own typing context and its typing rules and judgments, in the same way as other type theories. There are also several predefined finite types, void type (0), unity or truth (1), and bool (2). Both CoC and ITT are strongly normalising and therefore non Turing complete.

3.4 Curry–Howard Isomorphism

From the works of Curry and Howard on, it has been established the correspondence between each type theory and a style of logical calculus, or more broadly between type theory and proof theory (Cfr. Sørensen and Urzyczyn 2006). In simply typed lambda calculus, types can be built in a way akin to predicate logic well-formed formulae. This establishes a link between type theories and logical calculi, since the types of different systems can be treated as well-formed formulae of the corresponding logical systems. We will see this covering mainly the relationship between STLC and intuitionistic predicate logic.

In intuitionistic logic, the semantics is given by the Brouwer–Heyting–Kolmogorov interpretation: truth is identified with provability, so saying that A is true means that there is a proof a for A. If we interpret types as predicates and terms as proofs, we can see that these two judgments are equivalent:

- Proof a proves the predicate A.
- Type α can be built and this type is inhabited by a term $a : \alpha$.

Let 0 be the void type (the type with no proofs/terms) and $\neg\alpha \equiv_{def} \alpha \rightarrow 0$, that is, α leads to contradiction; $\alpha \times \beta$ means having a pair of terms, $a : \alpha$ and $b : \beta$, so it can be identified with $\alpha \wedge \beta$; $\alpha + \beta$ is having either a term $a : \alpha$ or a term $b : \beta$, that is, $\alpha \vee \beta$. The rule of term application ($e : \sigma \rightarrow \tau, f : \sigma \vdash ef : \tau$) equates to *modus*

Table 3 Correspondence between type theory and logical calculus

Type theory	Logic
Types and terms, $a : A$	a is a proof of A
Unit type and void type	Tautology and contradiction
Product type $A \times B$	$A \wedge B$
Function type $A \rightarrow B$	$A \rightarrow B$
Σ and Π dependent types	Quantifiers, $\forall$ and $\exists$
Application, $(\lambda x^A.t : B)u^A \vdash (tu) : B$	*Modus ponens*, $A \rightarrow B, A \vdash B$

ponens. We can see that there is an isomorphism between STLC and predicate logic. However, the constructed predicate logic is not classical predicate logic, but an intuitionistic system, and some classical tautologies such as Peirce's law cannot be obtained unless extra axioms are added.

In generalised type systems, this bijective equivalence between types and proofs is broadened. Predicate logic is related to dependent types systems, and higher order logic to polymorphic types. Type constructors are identified with the usual operators and quantifiers: Π-constructor equates to generalisation and implication, Σ-constructor to existential quantification; tautology equates to type 1, contradiction to void type (Table 3).

In sum, there is a correspondence between type systems and logical calculus systems, and between the elements and rules of these systems, and from an intuitionistic interpretation this correspondence serves as a link between proof theory and type theory. This isomorphism can be extended to the Curry–Howard–Lambek correspondence if we include the isomorphism between type theories and category theory that we have seen before, and the equivalence between CCC and intuitionistic propositional calculus observed by Lambek (1972).

3.5 Girard's Paradox

However, this correspondence cannot be fully maintained in certain type systems, since Girard's paradox (Cfr. Girard 1972; Coquand 1986) states that a type theory cannot quantify over all propositions and identify types and propositions at the same time. Therefore, one of these two points has to be left aside in order to maintain the validity of the other one. This issue will serve as a major difference between predicative ITT and impredicative CoC. The first one identifies types and propositions and thus leaves aside universal quantification over propositions, whereas the second one takes away the bijective identification between types and propositions:

- In an earlier, impredicative version of ITT, types *Prop* and *Type* are each identified with one another. The problem arises because the proposition *Type* : *Type* is not normalizing, thus it is not a well-typed term and it leads to a contradiction in the field of types analogous to Burali–Forti's paradox (Cfr. Reinhold 1989). As has been mentioned before, later versions of this theory

avoid this paradox by proposing a hierarchy of predicative universes in which the bijective relationship between propositions as types is not problematic. According to Curry–Howard correspondence, universes can be seen both as a constructive hierarchy of types or as a hierarchy of predicates (Palmgren 1998).
- CoC prefers to maintain quantification over propositions, thus not identifying isomorphically propositions and types. In this calculus, the identification of each proposition with the type of its proof is maintained, but it does not allow to identify every type with a proposition, because in CoC there are non-propositional types. In this way, CoC can be understood as a variation of Curry–Howard isomorphism, since strictly speaking it does not present a real isomorphic relation between types and predicates (Coquand 1986), but a weaker one.

Therefore, there cannot be a single unified, normalising type theory with the aforementioned properties, universal quantification and identification of types and propositions.

4 Conclusions

To sum up, the following remarks can be made:

- First of all, it is worthwhile mentioning the significance of classical paradoxes and their roles in the foundations of mathematics. Russell's paradox and Church's type theories are still the object of fruitful studies.
- Nowadays there are two basic aspects that make the study of pure type systems appealing. The first one is related to mathematics and the second one, to theory of computation: Firstly, in recent decades type theories are being studied in connection to new areas of mathematics and more specifically to the foundations of mathematics, such as category theory or homotopy theory. In a wider sense, it is interesting to observe how some of the problems of generalised type theory are similar to classical problems of Frege's logic and naïve set theory. Secondly, type theories as computing languages are the object of intensive research these days. Proof checkers, theorem provers and type checkers, tools within the field of automated reasoning, are founded on the principles of type theories. Turing incompleteness, that can sometimes be seen as a flaw, is revealed in other contexts as an advantage, since a Turing-incomplete type system is decidable. In this way, research on Coq or Agda relies on a heavy study on type theories, since they are decidably verifiable.

A few closing remarks can be made in a more general style. It can be useful to see how, instead of a single logical theory, a plurality of logical systems have emerged, giving new stimulus to the renewal of logical studies. Type theories were initially created as a response to the paradoxes of naïve set theory. This connection has never been lost, since several contemporary authors still study these theories as an alternative basis for the foundations of mathematics. Although the desire of a *lingua* *universalis*, a logical language able to verily code the structure of reality, still persists (Cfr. Granström 2011), the current direction points to a different goal. Instead of a global logical system, we can conceive several frames with invariances between them in which several logical theories can be developed (Cfr. Bell 1986). The question is not which logic is the real one, but which one we desire to use for the purposes of our research, because each type theory and more broadly each logical system has its own advantages and weaknesses and there is no system that can comprise all desired logical properties into a global, unified theory.

References

Asperti A, Longo G (1991) Categories, types, and structures: an introduction to category theory for the working computer scientist. MIT Press, Cambridge, MA

Barendregt HP (1991) Introduction to generalized type systems. J Funct Program 1(2):125–154

Barendregt HP (1993) Lambda calculi with types. In: Abramsky S, Gabbay DM, Maibaum TSE (eds) Handbook of logic in computer science, vol 2. Oxford University Press, New York

Bell JL (1986) From absolute to local mathematics. Synthese 69(3):409–426

Church A (1940) A formulation of the simple theory of types. J Symb Logic 5:56–68

Church A (1941) The calculi of lambda-conversion. Ser Ann Math Stud 6

Coquand T (1986) An analysis of Girard's paradox. In: Proceedings of the IEEE symposium on logic in computer science, pp 227–236

Eilenberg S, MacLane S (1945) General theory of natural equivalences. Trans Am Math Soc 58(2):231–294

Girard JY (1972) Interpretation fonctionelle et eleimination des coupures dans l'arithmetique d'ordre superieure. PhD thesis, Université Paris 7

Granström JG (2011) Treatise on intuitionistic type theory. Springer, Berlin

Jacobs B (1999) Categorical logic and type theory, vol 141. Elsevier, Amsterdam

Johnstone PT (2003) Sketches of an elephant: a topos theory compendium-2 volume set. In: Sketches of an elephant: a topos theory compendium-2 volume set, p 1

Lambek L (1972) Deductive systems and categories III. Cartesian closed categories, intuitionist propositional calculus, and combinatory logic. In: Toposes, algebraic geometry and logic. Springer, Berlin, pp 57–82

Lambek J, Scott PJ (1988) Introduction to higher-order categorical logic, vol 7. Cambridge University Press, Cambridge, MA

Landin PJ (1965) A correspondence between ALGOL 60 and Church's lambda-notation: Part I. Commun ACM 8(2):89–101

Lawvere FW (1963) Functorial semantics of algebraic theories. Proc Natl Acad Sci USA 50(5):869

Martin-Löf P (1975) An intuitionistic theory of types: predicative part. In: Logic colloquium '73, pp 73–118

Palmgren E (1998) On universes in type theory. In: Sambin G, Smith J (eds) Twenty-five years of constructive type theory, Oxford University Press

Peruzzi A (1991) Categories and logic. In: Problemi fondazionali nella teoria del significato, pp 137–211

Ramsey FP (1978) Foundations: Essays in philosophy, logic, mathematics and economics.In: DH Mellor, Routledge & Kegan Paul

Reinhold M (1989) Typechecking is undecidable when 'type' is a type. Technical report, Massachusets Institute of Technology

Reynolds JC (1974) Towards a theory of type structure. In: Programming symposium. Springer, Berlin, pp 408–425

Roorda J-W (2000) Pure type systems for functional programming. Master's thesis, University of Utrecht

Russell B (1903) The principles of mathematics. Cambridge University Press, Cambridge, MA

Russell B (1908) Mathematical logic as based on the theory of types. Am J Math 30(3):222–262

Russell B (1980) Correspondence with frege. In: Philosophical and mathematical correspondence, pp 130–170

Seely RAG (1984) Locally cartesian closed categories and type theory. In: Mathematical proceedings of the Cambridge philosophical society, vol 95. Cambridge University Press, Cambridge, MA, pp 33–48

Sørensen MH, Urzyczyn P (2006) Lectures on the curry-Howard isomorphism, vol 149. Elsevier

Voevodsky VA et al (2013) Homotopy type theory: univalent foundations of mathematics. Univalent Foundations Program. http://homotopytypetheory.org/book/