Course Notes in Typed Lambda Calculus

Thierry Coquand
Chalmers University

November 1998

Introduction

Since quite good books [12, 9] or review article [3, 10, 6, 18] are available on typed lambda calculi, these notes limit themselves to some historical remarks and some points that I consider delicate/important. In particular, I will emphasize the connections with logic and one goal of this course is that the reader can understand and appreciate the use of typed lambda calculi as representations of proofs in natural deduction.

1 History of Lambda-Calculus and Types

Both have a common origin in logic. The first notion of types seem to come from Frege: he revealed the conceptual difference between objects and predicates and considered the hierarchy built on these notions. At the "bottom" level we have a collection of basic objects. Then we have properties of these objects. Then we may have properties over these properties. Typically the quantifiers (introduced by Frege) are of such a nature: to be universally true is a property of properties. In such a view, it makes no sense for instance to apply a quantifier to an object.

It is thus *not* possible to apply what is known as Russell's paradox in this logic. The predicate of predicates P that do not hold on themselves, that is $\neg P(P)$ hold, cannot simply be formed in this system! Instead the problem with Frege's system was that Frege supposed that for each predicate P there was associated an object $\bar{P}$ the *extension* of P. Furthermore, it was supposed that $\bar{P}_1 = \bar{P}_2$ implies that $P_1(x)$ and $P_2(x)$ are equivalent for any object x. We can now form the predicate $P_0(x)$ defined by

$$\exists Q\ [[\bar{Q} = x] \wedge \neg Q(x)]$$

and prove that $P_0(\bar{P_0})$ is equivalent to $\neg P_0(\bar{P_0})$, and hence deduce a contradiction.

It seems that the lambda-calculus notation originates from the representation of classes in Principia Mathematica of Russell and Whitehead. The class, or predicate, of terms x satisfying a property ϕ is written there $\hat{x}\phi$. Thus, for instance the class of predicates x that do not satisfy themselves, used in Russell's paradox is the class $R\ =\ \hat{x}\neg x(x)$. Frege had already a notation for functional abstraction. Church, in the 1930s, analysing Principia Mathematica, introduced the powerful λ notation, writing $\lambda x\ t$ instead of $\hat{x}\ t$, and explicited the crucial β conversion rule

$$(\lambda x\ t)\ u\ =\ [x/u]t.$$

Notice that $\lambda x\ \phi$ represents a predicate which is satisfied by the objects a such that $[x/a]\phi$ holds. Thus, the first purpose of λ-terms was clearly to represent *formulae* and *predicates*.

Church thought first it would be possible to avoid Russell's paradox without introducing types, but by staying within an intuitionistic logic that use only some limited form of the law of excluded middle [4]. This system was however shown to be inconsistent by his students Kleene and Rosser [16][1]. Church formulated then an elegant formulation of higher-order logic, using simply typed λ-calculus [5], which can be seen as a simplification of the type system used in Principia Mathematica, but also is in some sense a return to Frege.

It seems that Wittgenstein noticed first [30] that it was possible to represent natural numbers as formulae, or λ-terms. In the tractacus [30], one can find the definition of what is now called Church numerals, as well as the representation of addition and multiplication. It is not clear how much Church was inspired by this in his formulation of arithmetic in λ-calculus. Kleene showed how to represent the predecessor function [11], that was not at all obvious at the beginning, and could show that more and more functions were actually representable. This leads, rather surprisingly if one thinks that the starting point was the predecessor function, to the formulation of Church's thesis that any computable function is representable as a λ-term. We arrive thus at the notion that λ-terms, which at first represented only formulae, can represent *programs*.

2 Definition of Simply Typed Lambda Calculus

2.1 Notation

A systematic notation for function application has been introduced in combinatory logic (Curry,Church) and used in functional programming. This notation differs from the usual mathematical notation: the application of a function f to an argument a is written by concatenation $f\ a$ instead of $f(a)$. More generally, when more than two terms occur, association shall be to the left, so that $f\ a\ b$ denotes $(f\ a)\ b$. This notation was introduced already in Schönfinkel's paper [24], who introduces the concept of combinator. The goal of this paper is clearly stated, after recalling how the logical connectives can be explained in term of the Sheffer connective: "We are led to the idea, which at first glance certainly appears extremely bold of attempting to eliminate by suitable reduction the remaining fundamental notions, those of proposition, propositional function, and variable, from those contexts in which we are dealing with completely arbitrary, logical general propositions ... To examine this possibility more closely and to pursue it would be valuable not only from the methodological point of view that enjoins us to strive for the greatest possible conceptual uniformity but also from a certain philosophic, or if you wish, aesthetic point of view."

Besides this notation for function application, lambda calculus introduces a direct notation for functions themselves. In mathematics there is no direct notation for functions, but we have to refer to them by names. Thus we would say something like: let f be the map $x \longmapsto x^2 + 3$ and use f. Sometimes, one can even refer to f as the function $x^2 + 3$ but this may be confusing when several variables are involved. For instance $g : x \longmapsto x^2 + y$ where y is a parameter. The lambda notation allows to refer directly to these maps, respectively as $\lambda x\ x^2 + 3$ and $\lambda x\ x^2 + y$. This notation is so convenient that one often wonders why it is not used in mathematics.

2.2 Syntax

First we define types: we have a set of basic types, and if α, β are types then $\alpha{\to}\beta$ is a type. A convenient notation, when more than two types occur, is to associate to the right, so that we

[1] What is remarkable is that they obtained their paradox by a formalisation in Church's system of Richard's paradox about the "least integer definable in less than 100 words." (Poincare had stressed the importance of this paradox.)

write $\alpha_1 \to \alpha_2 \to \alpha_3$ for the type $\alpha_1 \to (\alpha_2 \to \alpha_3)$. We start from a set of identifiers V and we write $V_\alpha = V \times \{\alpha\}$ the set of variables of type α. Next, the set T_α of terms of type α is defined as follows:

- if $x \in V_\alpha$ then $x \in T_\alpha$,
- if $t \in T_\beta$ and $x \in V_\alpha$ then $\lambda x\ t \in T_{\alpha \to \beta}$,
- if $t \in T_{\alpha \to \beta}$ and $u \in T_\alpha$ then $t\ u \in T_\beta$.

2.3 Models and Conversion

We present here a model theoretic definition of conversion, that avoids subtle syntactical considerations (alpha-conversion, clashes of variables) that are rarely treated rigourously. Instead I make use of a completeness theorem to *define* the notion of conversion between lambda-terms.

2.3.1 Henkin's Definition

The intended model is the set-theoretic model where we suppose given a set D_ι for each base type ι and where $D_{\alpha \to \beta}$ is the set of all functions from D_α to D_β. We define then typed environments that are family of functions $V_\alpha \to D_\alpha$ so that $\mathsf{Env} = (\Pi\alpha)\ V_\alpha \to D_\alpha$. If $x \in V_\alpha$ and $a \in D_\alpha$ and $\rho \in \mathsf{Env}$ let $(\rho, x = a) \in \mathsf{Env}$ be the environment ν such that $\nu(y) = \rho(y)$ if $y \neq x$ and $\nu(x) = a$. By induction on $t \in T_\alpha$ it is possible to give a meaning $t\rho \in D_\alpha$ for any environment $\rho \in \mathsf{Env}$. It is defined by the laws

1. $x\rho = \rho(x)$,
2. $(\lambda x\ t)\rho = a \longmapsto t(\rho, x = a)$, $\qquad (*)$
3. $(t\ u)\rho = t\rho\ (u\rho)$.

Henkin generalised this model by starting from a collection D_α of sets indexed by the types, such that $D_{\alpha \to \beta}$ is only a *subset* of the set of all functions from D_α to D_β. The intuition is that the objects of the models should be only those functions which preserve some given structure. Now, in general given such a collection, it is not possible to define the meaning function $t\rho$ because in the clause $(*)$ it may not be the case that the function

$$a \longmapsto t(\rho, x = a)$$

belongs to $D_{\alpha \to \beta}$.

Henkin defines a *model* to be such a collection D_α such that this recursive definition of $t\rho$ is possible.

As Henkin noticed himself, this definition has some impredicative flavour.

2.3.2 Alternative Definition

A more algebraic definition of model is the following. We suppose given for each type α a set D_α with an application map $D_{\alpha \to \beta} \times D_\alpha \to D_\beta$ that will be written as concatenation. Finally, we suppose given a map $T_\alpha \times \mathsf{Env} \to D_\alpha$

$$(t, \rho) \longmapsto t\rho$$

such that

1. $x\rho = \rho(x)$,
2. $(\lambda x\ t)\rho\ a = t(\rho, x = a)$,
3. $(t\ u)\rho = t\rho\ (u\rho)$.

These equations are quite natural. The second equation says that $(\lambda x\ t)\rho$ can be thought of as the function $a \longmapsto t(\rho, x = a)$, which is the intended semantics of lambda abstraction. Finally, the last equation interprets application as ordinary function application.

So far, we get only a notion of *weak* model, which is more general than Henkin's notion of model. In order to get back the same notion of models as Henkin's model, we have to suppose furthermore the following *extensionality condition* that $f = g \in D_{\alpha \to \beta}$ whenever $f\ a = g\ a$ for all $a \in D_\alpha$. If the model satisfies this extensionality condition, we can think of $D_{\alpha\to\beta}$ as a subset of $D_\alpha^{D_\beta}$ and this definition is equivalent to Henkin's definition. In this model, it can be checked that the law of η-conversion is valid: if x is not a free variable of t than for any environment ρ for t

$$(\lambda x\ t\ x)\rho = t\rho$$

because for any a we have

$$(\lambda x\ t\ x)\rho\ a = (t\ x)(\rho, x = a) = t\rho\ a.$$

If we want a model of β-conversion only, we have to replace the extensionality condition by the weaker so-called *Berry condition*: we have $(\lambda x\ u)\rho = (\lambda y\ v)\nu$ iff for any value a we have $u(\rho, x = a) = v(\nu, y = a)$.

2.3.3 Conversion

We can now define conversion at each types: two terms t and u in T_α are convertible iff for any model D and any environment ρ we have $t\rho = u\rho \in D_\alpha$.

For instance $t = (\lambda x\ x)y$ and y are convertible if x, y are of type α because for any model D and any environment ρ we have

$$t\rho = (\lambda x\ x)\rho\ (y\rho) = x(\rho, x = y\rho) = y\rho.$$

In the same way $f \in V_{\alpha\to\alpha}$ and $\lambda x\ f\ x$ are convertible because for any $a \in D_\alpha$

$$(\lambda x\ f\ x)\rho\ a = (f\ x)(\rho, x = a) = f(\rho, x = a)\ (x(\rho, x = a)) = f\rho\ a.$$

The advantage of this definition is that it is completely rigourous, and does not require the definition of bound variables substitution, renaming of variables, ... On the other hand one can show that it is equivalent to the usual definition, that the terms have the same η-expanded normal form.

2.4 Examples of Models

2.4.1 Set Theoretic Model

We take for D_ι, where ι is a base type, an arbitrary set and we interpret $D_{\alpha\to\beta}$ as the set of *all* set theoretic functions from D_α to D_β. (We recall that, in set theory, a function is identified with a functional relation.) There is then now problem to define $t : \mathsf{Env}{\to}D_\alpha$ by induction on t :

- if $t = x$ then $t\rho$ is $\rho(x)$,
- if $t = t_1\ t_2$ then we define inductively $t\rho$ as the value of the function $t_1\rho$ on the argument $t_2\rho$,
- if $t = \lambda x\ u$ then $t\rho$ is the function that associates to the value a the value $u(\rho, x = a)$ which is defined by induction hypothesis.

2.4.2 Domain Model

The previous model may be thought of as the intended model. However, in general, the semantics of a term is a functional preserving some structures. (We will see later that the definable functionals satisfy also remarkable uniformity property.) A typical example is the domain model. The set D_ι, for ι base types, are now partial order sets with a bottom element $\perp$ and sups of increasing chains. The set $D_{\alpha\to\beta}$ is the poset, for the pointwise ordering, of the set of all increasing maps $f : D_\alpha \to D_\beta$ preserving the sups of the increasing chains, that is $f\ (\vee x_n) = \vee(f\ x_n)$ for any chain $x_0 \leq x_1 \leq x_2 \dots$

When trying to define the meaning of lambda terms in these domains, there is a difficulty in the abstraction case, for defining $(\lambda x\ u)\rho$, which is typical of the definition of models of typed lambda calculus. We should define it as the map $a \longmapsto u(\rho, x = a)$ where, for each a, we have defined inductively the value $u(\rho, x = a)$. The difficulty is that we don't know that this value depends *continuously* on a. The solution is formally easy, but conceptually involved: we define inductively $[\![t]\!] : \mathsf{Env} \to D_\alpha$ to be a *continuous* function from Env to D_α.

The problem is then reduced to the following lemma, whose proof is a direct consequence of the definitions.

Lemma: If $f : D_1 \times D_2 \to D$ is a continuous function then the function $a \longmapsto \lambda y\ f(a, y)$ from D_1 to $[D_2 \to D]$ is a continuous function.

If $t = \lambda x^\alpha\ u$ with $u : \beta$ then by induction we have a continuous function $[\![u]\!] : \mathsf{Env} \to D_\beta$. On the other hand the update function $(\rho, a) \longmapsto (\rho, x = a)$ is a continuous function from $\mathsf{Env} \times D_\alpha$ to Env. By composition we get a continuous function $\mathsf{Env} \times D_\alpha \to D_\beta$:

$$(\rho, a) \longmapsto u(\rho, x = a)$$

and by applying the lemma we get the interpretation of $[\![t]\!] \in \mathsf{Env} \to D_{\alpha\to\beta}$.

One interest of this model is that there is a fixedpoint operator $Y_\alpha \in D_{(\alpha\to\alpha)\to\alpha}$ defined as $Y_\alpha\ f = \vee(f^n\ \perp)$. It can indeed been checked that the sequence $f^n\ \perp$ is increasing and that $Y_\alpha\ f$ depends continuously on f.

There is something quite remarkable about the cardinality of the domains D_α, which is indeed described by Dana Scott as his "first original discovery" in the theory of domains [25]. If we start from domains D_ι such that there exists a countable basis $B_\iota \subseteq D_\iota$, that is a subset such that any element can be written as a sup of an increasing chain of elements in B_ι, then it can be checked that any domain D_α, even at higher types, has a countable basis. It follows that the cardinality of each domain is at most the one of the continuum. Actually, a rigourous and satisfactory description of these domains is best done by an explicit construction of each basis, and this is achieved by the notion of *information system*, see [29].

2.4.3 Recursive Model

I refer here to [18]. Intuitively, this model keeps only the functions and functionals that are "computable". It is important to realise that the meaning of computable is quite subtle however.

At type $\iota\to\iota$, it coincides with the notion of *recursive* function. At type $(\iota\to\iota)\to\iota$, it is not clear if to be "computable" at this type should imply that it is defined only for computable functions or on an arbitrary function.

2.4.4 Term Model

We take $D_\alpha = T_\alpha$ and $t\rho$ is obtained by substituting x to be $\rho(x)$ in t.

This term model is a nice illustration of Skolem "paradox" since all types are interpreted by a countable set. In Henkin's paper [13], where such a construction is first defined, it is explicitely noted that the existence of a countable model is not clear a priori, given the apparent impredicativity or circularity of the notion of models (we have to give a set D_α for all types α). In contrast, both the set-theoretic and the domain models are determined by the choice of the domains at base types and built by induction on the type hierarchy.

2.4.5 Implementation Model

We extend the language with one *fixpoint* operator $Y : (\alpha\to\alpha)\to\alpha$ and the new conversion rule $Y\ u = u\ (Y\ u)$ for any $u \in T_{\alpha\to\alpha}$.

3 Logical Relations

Let (D_α) and (L_α) be two models of simple typed lambda-calculus. Let (R_α) be a family of relations $R_\alpha \subseteq D_\alpha \times L_\alpha$. We say that (R_α) is a *logical relation* iff $R_{\alpha\to\beta}(t,t')$ is equivalent to $(\forall u, u')\ R_\alpha(u,u') \Rightarrow R_\beta(t\ u, t'\ u')$. The following result is known as the *Fundamental Theorem of Logical Relations*:

Theorem: for any term t of type α we have $R_\alpha(t\rho, t\rho')$ if $R(\rho(x), \rho'(x))$ for any free variable x of t.

Proof: by a direct induction on the term t. $\square$

This theorem deserves its name, for, while its proof is rather simple, most of the results about lambda calculus can be seen as corollary of this result.

A generalisation concerns the addition of constant; one then checks that each constant has its interpretation related to itself.

3.1 Uniformity of Lambda Terms

We suppose to have only one ground type ι. If (D_α) is a set-theoretic model and σ a permutation on D_ι it is possible to extend σ at all types by defining at type $\alpha\to\beta$

$$\sigma\ f = a \longmapsto \sigma_\beta(f\ (\sigma_\alpha^{-1}\ a)).$$

In this way, we have $\sigma\ (f\ a) = \sigma\ f\ (\sigma\ a)$.

Intuitively, any *definable*, $a \in D_\alpha$, that is such that there exists a closed term t such that $a = t()$, satisfies strong uniformity property, which should be expressed by $\sigma\ a = a$ for any permutation σ. Logical relation can be used to prove such a result. Define at all types $R(a, a')$ to be $\sigma\ a = a'$.

Lemma: $R(f, f')$ holds at type $\alpha\to\beta$ iff $R(a, a') \Rightarrow R(f\ a, f'\ a')$.

Indeed, this implication can be rewritten as $(\sigma\ f)\ a' = f'\ a'$ for all $a' \in D_\alpha$, and we see that the lemma simply expresses the extensionality of the models.

It follows from the fundamental theorem of logical relation that $R(t(), t())$ that is $\sigma\ t = t$ for any closed term t.

3.2 Definability Problem

Given the strong uniformity condition of definable terms, one can ask if conversely, this condition is enough to ensure that a term is definable. A related question is the *definability problem*: is there an algorithm that computes when a given object $a \in D_\alpha$ built from given *finite* sets is definable? It was shown by Sieber [26] that the answer is positive if α is of order ≤ 2 (where the order of a ground type is 0 and the order of $\alpha \to \beta$ is the maximum of the order of β and the successor of the order of α.) A surprising result of Loader is that the answer is *not* decidable at order 3! An implementation of Sieber's algorithm is described in [26] and is a nice application of logical relation and of representation of inductive definitions in second-order logic.

3.3 Friedman's Completeness Theorem

We say that a logical relation R is a *partial onto morphism* iff it satisfies the two conditions at each type

- $\forall a' \exists a\ R_\alpha(a, a')$,
- $R_\alpha(a, a'_1), R_\alpha(a, a'_2) \Rightarrow a'_1 = a'_2$.

The following result justifies somewhat the use of the word "morphism".

Lemma: If R is a partial onto morphism and ρ, ρ' are related then $M_1\rho = M_2\rho$ implies $M_1\rho' = M_2\rho'$.

Given the axiom of choice, the set theoretic model (full type structure) has the following property that any relation defined on base types satisfying the two conditions above can be extended to a partial onto morphism. It follows that there is a partial onto morphism from the set theoretic model where all the base types are countable in the term model. One deduces from this and the lemma Friedman's completeness theorem.

Theorem: two closed terms of the same type are convertible iff they have same interpretation in any infinite set-theoretic model.

3.4 Normalisation Theorem

It should be clear that in the fundamental theorem of logical relation, we could have used predicate instead of relations. Let us reformulate the theorem in this case: we suppose to have a subset $P_\alpha \subseteq D_\alpha$ such that $[\![c]\!] \in P_\alpha$ for any constant of type α and $d \in P_{\alpha \to \beta}$ iff $d\ u \in P_\beta$ for all $u \in P_\alpha$. In this case we have

Theorem: if $x\rho \in P_\alpha$ for any x of type α then $M\rho \in P_\tau$ whenever $M : \tau$.

As an application take the term model and let P_τ be the set of normalisable terms of type τ for τ atomic. We extend P_α by induction on α in a unique way so that it becomes a logical predicate. It follows from the theorem that we have $M() \in P_\tau$ for any closed term M and hence that M is normalisable.

3.5 Adequacy Theorem

Here is still another application of the notion of logical relation, which is the theoretical foundation of the use of domain theory for proving equivalence of programs.

The adequacy theorem shows that there are good relations between the syntax and the semantics for ground type. It is stated here for an idealised programming language: simple typed λ-calculus + a fixed-point combinator Y. We assume that among the base types, there is a type N of natural numbers, and we take for its semantics the "flat" domain $D_N = \{\perp\} \cup N$.

We consider two models: one is the domain model, where Y is the least fixed-point operator $Y\ f = \bigvee f^n\ \ \perp$ and the other is any model (L_α) with a family $Y_\alpha \in D_{(\alpha\to\alpha)\to\alpha}$ such that $Y\ u = u\ (Y\ u)$ if $u \in D_{\alpha\to\alpha}$.

If t is a closed term it should be clear that its semantics $t\rho$ in any model does not depend on ρ. (In general $t\rho$ depends only of the value of ρ on the finite set of the free variables of t.) We write $t()$ this common value

Theorem: Let t be a closed term of type N. If we have $t() = n$ is the domain model then we have also $t() = n \in L_N$.

We define $R_N(u, v)$ to mean that $v = \perp$ or else that there exists n such that $v = n$ and $u = n \in L_N$. At higher type $R_{\sigma\to\tau}(w, f)$ means that $R_\sigma(u, v)$ implies $R_\tau(w\ u, f\ v)$. We know that, by the fundamental theorem of logical relations, for all term $t : \sigma$, ρ and ρ' related environments, we have $R_\sigma(t\rho, t\rho')$, *provided* the constants are related. This implies the theorem when σ is N. We are reduced to check that $R(Y, Y)$ holds.

Lemma: For each type σ, we have

- $R_\sigma(u, \perp)$ for any $u \in L_\sigma$,
- if v is the sup of the increasing chain (v_n), and $R(u, v_n)$ for each n, then $R(u, v)$.

The proof of this lemma is rather direct.

Corollary: We have $R(Y, Y)$ that is, if $R(u, v)$ then $R(Y\ u, \bigvee v^n\ \perp)$.

Proof: By the lemma, it is enough to show $R(Y\ u, v^n\ \perp)$ for all n. We show this by induction on n. The lemma shows the case $n = 0$. If it is true for n, since $R(u, v)$ we deduce $R(u\ (Y\ u), v^{n+1}\ \perp)$. But $Y\ u$ is convertible to $u\ (Y\ u)$, hence the result. □

This result is important in order to prove operational equivalence: let say that t_1 and t_2 programs of type σ (that may be functional) are *operationally equivalent* iff for all context $C[\_]$ of type N, $C[t_1]$ is convertible to an integer iff $C[t_2]$ is convertible to an integer, and in that case, these integers are equal. The intuition is that we can "test" programs only by embedding them into programs that are "observable", and functions, functionals... are not really observable (they are really ideal elements). A direct application of the theorem is the following result.

Corollary: Denotational equality implies operational equivalence.

For proving the corollary we apply the theorem to the term model. The derivation of this corollary uses in an essential way the fact that the semantics is *compositional*: the semantics of an expression is a function of the semantics of its subpart. (This is reflected that the definition of the semantics of a term is by structural induction.) This corollary gives an elegant way of proving the operational equivalence of part of the programs.

4 Where does this come from?

It is extremely instructive to read Reynolds' papers [20, 21] to understand one motivation of the introduction of type variables and logical relations. One wants to represent the idea of type definitions and type abstractions. The idea is that one can define a type with some operations, for instance

$$\tau = int, \quad mk:\tau = 0, \quad inc:\tau{\rightarrow}\tau = \lambda x\ x+1, \quad get:\tau{\rightarrow}int = \lambda x\ x$$

and then uses the type τ and these operations mk, inc, get but, and this is the important point, *without* having access to their definitions. This is represented quite naturally in polymorphic λ-calculus as a term of the form

$$(\lambda\ \tau(\lambda\ mk:\tau)(\lambda\ inc:\tau{\rightarrow}\tau)(\lambda\ get:\tau{\rightarrow}int)\ \ e)\ int\ 0\ (\lambda x\ x+1)\ (\lambda x\ x)$$

This motivates the introduction of type variables. For motivating logical relations, suppose that we change the given implementation and we take instead

$$\tau = int, \quad mk:\tau = 0, \quad inc:\tau{\rightarrow}\tau = \lambda x\ x-1, \quad get:\tau{\rightarrow}int = \lambda x\ (-x)$$

We can prove using logical relation that if we have a term $e : int$ using τ, mk, inc, get as variables and we consider two close instantions e_1 and e_2 then we have $e_1 = e_2 : int$. In order to prove this, consider the language where we add a new type constant τ and news operations $mk : \tau$, $inc : \tau{\rightarrow}\tau$, $get : \tau{\rightarrow}int$. A model of this language gives a meaning to the type τ and the operations mk, inc and get. We give two different models, with the logical relation given by $R_\tau(x, y))$ iff $x + y = 0$. Applying the fundamental theorem we get that for any related pairs of environment ρ_1, ρ_2 we have $R_\alpha(e\rho_1, e\rho_2)$. In particular if α does not mention τ we get $e\rho_1 = e\rho_2$.

For another example, consider the two implementations

$$\tau = bool, \quad bit:\tau = true, \quad flip:\tau{\rightarrow}\tau = not, \ \ read:\tau{\rightarrow}bool = \lambda x\ x$$

and

$$\tau = int, \quad bit:\tau = 1, \quad flip:\tau{\rightarrow}\tau = \lambda x\ \ -x, \ \ read:\tau{\rightarrow}bool = \lambda x\ x>0$$

then these are two related different implementations of the same abstract structure

$$(\tau, a:\tau, f:\tau{\rightarrow}\tau, g:\tau{\rightarrow}bool).$$

Here we take $R(b, x)$ iff $b = true$ and $x > 0$ or $b = false$ and $x < 0$. Then $R(a_1, a_2)$ holds but also $R(f_1, f_2)$ and $R(g_1, g_2)$.

Notice that from the adequacy theorem we get as a corollary a purely operational result, that should be hard to prove in an operational way. This results also holds in presence of a fixed-point operator.

5 Higher-Order Logic

An elegant formal system is "minimal" higher-order logic. The types and the terms of this system are the same as the terms of Church simple type theory [5]. The types are built from the ground types $\mathbf{o}$ (type of "propositions") and ι (type of individuals). There is a constant $\Rightarrow\ :\mathbf{o}{\rightarrow}\mathbf{o}{\rightarrow}\mathbf{o}$ (we write $\phi \Rightarrow \psi$ for $\Rightarrow\ \ \phi\ \psi)$) and a "polymorphic" constant $\forall : (\alpha{\rightarrow}\mathbf{o}){\rightarrow}\mathbf{o}$ (we write usually $(x)\phi$ for $\forall(\lambda x.\phi))$. A proposition is a term of type $\mathbf{o}$.

We can define what is a "true" proposition relatively to a finite set of "hypotheses" Γ, that are propositions i.e. terms of type $\mathbf{o}$, relation that we write $\Gamma \vdash \phi\ \mathsf{true}$, inductively as follows: first $\Gamma \vdash \phi\ \mathsf{true}$ if $\phi \in \Gamma$, then

$$\frac{\Gamma, \phi \vdash \psi\ \mathsf{true}}{\Gamma \vdash \phi \Rightarrow \psi\ \mathsf{true}}$$

$$\frac{\Gamma \vdash \phi \Rightarrow \psi\ \mathsf{true} \qquad \Gamma \vdash \phi\ \mathsf{true}}{\Gamma \vdash \psi\ \mathsf{true}}$$

$$\frac{\Gamma \vdash \phi\ \mathsf{true}}{\Gamma \vdash (x)\phi\ \mathsf{true}} \qquad (*)$$

$$\frac{\Gamma \vdash (x)\phi\ \mathsf{true}}{\Gamma \vdash [t/x]\phi\ \mathsf{true}}$$

In the rule (*), it is suppose that the variable x doesn't occur free in any formula of Γ.

This logic is known as "minimal" higher-order logic. It is a simplification of Church's formulation of the simple theory of types [5]. It is based on the so-called *deduction theorem* in logic, which explains how to prove conditional statement.

As a typical example, we can consider *Leibniz' equality*: if x, y are two terms of the same type, we define Eq as $\lambda x \lambda y\ \forall P\ [(P\ y) \Rightarrow (P\ x)]$. This corresponds to the following rule: if $\mathsf{Eq}\ x\ y$ and we want to prove the proposition $\phi(x)$, then it's enough to prove $\phi(y)$. This name is justified since Leibniz considered a similar purely logical definition of identity.

Exercice: Prove the reflexivity, symmetry and transitivity of the relation Eq.

Higher-order logic is ordinary presented with extensionality. In this case, as explained in [2], it is better to take the (extensional) equality as primitive and it is then possible to define all other logical connectives from it. It is thus important to relate the intensional presentation of higher-order logic given above to an extensional one.

Such an interpretation was done in [8] and is an early appearance of the notion of logical relation. The idea is to define for every type an extensional equality by induction on the type. On the type $\mathbf{o}$ we take the logical equivalence as equality and we extend it at all types by the technique or logical relations. In this way, we can interpret extensional higher-order logic in intensional higher-order logic.

Henkin has an enjoyable and important discussion in [14] describing how he found his completeness theorem for type theory and first-order logic. I found it quite interesting that the starting point was an analysis of the *definable* elements of the set-theoretic model. Also, his proof of completeness for first-order logic came as an application of the method used for higher-order logic. Henkin was working in a logic that included the following form of the axiom of choice: we have a special polymorphic constant ϵ of type $(\alpha \rightarrow \mathbf{o}) \rightarrow \alpha$ with the following axiom schema

$$(f)(x)[f\ x \Rightarrow f\ (\epsilon\ f)].$$

This axiom expresses that, if ever there is an object satisying f then $\epsilon\ f$ satisfies f.

We can also consider the case where we have only o as a base type. We get then higher-order propositional logic. This is a decidable logic, and it can be shown that it is not elementary.

5.1 Unification Problem

Higher-order logic allows for a quite elegant and sometimes rather compact representation of mathematical arguments. It is thus a natural candidate for mechanical representation of propositions and proofs, as a natural generalisation of first-order logic. We analyse the corresponding unification and matching problem but limit ourselves to some remark about this problem.

First the example of finding f such that $f\ 0 = 0$ shows that there is no hope of having a most general unifier. There are two natural solutions $f = \lambda x\ x$ and $f = \lambda x\ 0$ that are clearly independent. There may even be an infinity of independent unifiers, as shown by the example of finding f such that $f\ (F\ A) = F\ (f\ A)$ where $A : \iota$ and $F : \iota \rightarrow \iota$ are given constants.

Huet found that the unification problem is not decidable. At the time the result about undecidability of diophantine equations was not known. The undecidability, even at order 2, is a rather direct corollary of this result and of the representation of polynomial functions in simply typed lambda calculus, as we shall see later.

Huet showed furthermore that the matching problem is decidable at order 2, and conjectured the decidability at all orders. It was shown recently to be decidable at order 3 by Dowek, but the general problem is still open.

6 Representation of Number Theoretic Functions

This is a quite old topic, since it appears already in Wittgenstein [30], and it was developped systematically by Kleene [11] for his Ph. D. thesis. He did it in an untyped framework, but it so happens that all his terms can be typed, and some number theoretic functions appear in Church's paper [5] on typed lambda calculus. We shall describe the representation of the predecessor function, that was historically an important problem, and is solved in a conceptual way in polymorphic lambda calculus. I will follow the presentation in the survey paper [6], notice that the exponential is *not* definable without "shifting" types, and explain why there *cannot* be a good representation of exponential in simply typed lambda calculus.

By this, we mean the following result. It is possible to define $N_\alpha = (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ and functions of type $N_\alpha \rightarrow N_\alpha \rightarrow N_\alpha$ representing the addition and the multiplication. Thus one can wonder if one can represent the function $n \longmapsto 2^n$ as a term of type $N_\alpha \rightarrow N_\alpha$. That this is not possible follow from the super-exponential bound on the depth of a normal form of a term.

Proposition: if M is a term of depth $\leq n$ and of complexity $\leq k$, then M reduces to a term of depth $\leq 2^n$ and of complexity $< k$.

I have to define the depth of a term: this is the depth of the term seen as a tree. Its complexity is the maximal complexity of its redexes where the complexity of a redex $P\ Q$ is the complexity of the type of the function P. The complexity $c(\tau)$ of a type τ is 0 if the type is atomic and $max(c(\tau_1) + 1, c(\tau_2))$ if $\tau = \tau_1 \rightarrow \tau_2$. We write also $c(M)$ the complexity of a term M.

Lemma: If M reduces to M' then $c(M') \leq c(M)$.

The lemma follows from the fact that the reduction of a redex creates only redexes that are strictly less complex.

The proof of the proposition is by induction on M:

- If M is a variable x, then we take $M' = x$;
- If M is an abstraction $\lambda x\ P$, then by induction P reduces to P' of depth $\leq 2^{n-1}$ and hence M reduces to $M' = \lambda x\ P'$ of depth $\leq 2^{n-1} + 1 \leq 2^n$;

- If M is an application $P\ Q$, then by induction P reduces to P' of depth $\leq 2^{n-1}$ and Q reduces to Q' of depth $\leq 2^{n-1}$. If the complexity of $P'\ Q'$ of $< k$, we can take $M' = P'\ Q'$ which is of depth $\leq 2^{n-1}+1 \leq 2^n$. By the lemma, the complexity of M' is $\leq k$ and it is k iff $P'\ Q'$ is a redex of complexity k. In this case $P' = \lambda x\ N$ and $P'\ Q'$ reduces to $[x/Q']N$ which is of depth $\leq 2^{n-1} + 2^{n-1}$ and of complexity $< k$.

We can now prove that there is no term $E : N_\alpha \rightarrow N_\alpha$ representing exponentiation $k \longmapsto 2^k$. Indeed, if there were such a term, we could form the term $\bar{k}\ E\ \bar{k} : N_\alpha$ where $\bar{k}$ is $\lambda f \lambda x\ f^k(x)$. This term will reduce to $2^{2^{2^{\dots k}}}$ where the tower of exponential is of length k and this contradicts the upper bound of the proposition, since these terms, when k varies, are of fixed complexity.

Finally, notice that since we can represent multiplication and addition, we can also represent any polynomial $N^k \rightarrow N$ with positive coefficient. Since we cannot decide whether or not an equation $P(n_1, \dots, n_k) = Q(n_1, \dots, n_k)$ has a solution, a corollary is the following result.

Theorem: The unification problem for simply typed λ-calculus is undecidable.

7 Polymorphism

Here also, I will follow the presentation in [6], and be informal about the syntax. I will describe the presentation of natural numbers and of ordinals, and show how to represent Ackerman's function, which is an example in the paper of Reynolds that introduces polymorphic lambda calculus.

I will use an untyped notation for abstraction, and explain Tait's normalisation proof, which is a simplification of Girard's argument, and can be seen as a realisability semantics of propositional second-order logic.

7.1 Non existence of set theoretic model

We explain informally Reynolds' proof, and what is the main problem in the translation of this argument in type-theory. We apply later this method to the construction of a paradox in the system with a type of all types. What I will use is a extension of logical relation to polymorphic types. Since I will use only universally quantified types, this extension is rather intuitive.

To understand Reynolds' derivation, it is useful first to explain [21], which was written a little before Reynolds found its proof [22], and where it is explained why it may seem at first that there does exist a set-theoretic model of polymorphism. This will be also an introduction to [28], which is a nice application of the technique of logical relations. Let us look at simple type, like $\Pi\alpha.\alpha\rightarrow\alpha$ or $\Pi\alpha.\alpha\rightarrow(\alpha\rightarrow\alpha)\rightarrow\alpha$. A type like $\Pi\alpha.\alpha\rightarrow\alpha$ has no direct set-theoretic interpretation: the product over all set X of X^X is "too big" to be a set. The basic idea is to restrict this product to the collection of families (t_X), with $t_X \in X^X$, that are "uniform", the hope being that a strong enough notion of uniformity will reduce this collection to a set. (A similar idea works indeed for giving a domain model of polymorphism.)

A plausible definition of uniformity, coming from logical relation, is: a family $(t_X) \in \Pi X.X^X$ is uniform if and only if it sends related input to related output. That is for all sets X, Y and relation $R(x, y)$ between $x \in X, y \in Y$, if $R(x, y)$ then $R(t_X\ x, t_Y\ y)$. The identity family satisfies this uniformity condition, but conversely it is possible to see that if the identity family is the *only* uniform family. Not only we get a set in this case, but even a singleton set.

We can try something similar in the case of the type

$$\Pi\alpha.\alpha\rightarrow(\alpha\rightarrow\alpha)\rightarrow\alpha.$$

We say that a family $(t_X) \in \Pi X.(X^{(X^X)})^X$ is uniform if, and only if, whenever $R(x,y)$ is a relation between $x \in X, y \in Y$ and $R(x,y)$, and $f \in X^X, g \in Y^Y$ satisfies that $R(u,v)$ implies $R(f\ u, g\ v)$, then $R(t_X\ x\ f, t_Y\ y\ g)$. It is then possible to show that if (t_X) is uniform, then there exists a fixed $n \in N$ such that $t_X\ x\ f = f^n(x)$ for all $x \in X, f \in X^X$. In this case again, we get a set, which is isomorphic to N.

There is a clear pattern in these two examples. We can get in this way the set-theoretic free initial algebra as the interpretation of the corresponding type. For instance, for the type

$$\Pi\alpha.\alpha \to (\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha,$$

the collection of all uniform family will be isomorphic to the free initial algebra with one constant and two unary functions.

The fact of being an initial algebra can be stated in purely abstract categorical term (with the usual existence and unicity property). Reynolds noticed that, without even trying to compute the semantics of a type like $\Pi\alpha.\alpha \to \alpha$, we can prove abstractly that the collection of uniform family is an initial algebra.

Now, in order to show that there cannot be any set-theoretic model, we start with a notion of algebra that has no initial algebra in set-theory, for instance the algebra A with one constructor $2^{(2^A)} \to A$. This corresponds to the type

$$\Pi\alpha.(((\alpha \to B) \to B) \to \alpha) \to \alpha$$

(see [22]) and this type cannot then be interpreted as a set.

As we will see below, this reasoning can be actually done syntactically in the system with a type of all types, and this was for some time actually one of the best way of producing a paradox in this system. The idea is to represent a set as a pair of a type together with a equivalence relation on it. The concept of "uniformity" fits well in this framework, since it appears as a definition of a particular equivalence relation.

8 Types as Sets of Programs

8.1 Models of Untyped Lambda Calculus

The simplest notion of algebra may be the one of a set A with a binary map $A \times A \to A$, with no conditions. This is a representation of an application of a program to another, which may be seen as derived from Schönfikel's idea of application as a binary operation. Given a set of variables we can form the *free* algebra $A[V]$ over A: if we have another algebra L and an application $f : V \to L$ any algebra map $i : A \to L$ has a unique extension $i' : A[V] \to L$ such that $i'(v) = f(v)$ for $v \in V$. We say that A is a *combinatory algebra* iff for any $t \in A[x_1, \ldots, x_n]$ there exists $c \in A$ such that $c\ x_1\ \ldots\ x_n = t$.

We suppose given one such combinatory algebra D which is furthermore extensional: we have $f = g \in D$ whenever $f\ u = g\ u$ for all $u \in D$.

This can be seen as an abstract model of an untyped programming language. Given any typed lambda term M and any environment $\rho : V_\alpha \to D$ we can define $M\rho \in D$. Intuitively, we forget all type informations. We are going to give properties of $M\rho$ that we can infer from the type of M.

As emphasised by Reynolds, there are two opposed views of types. One view stresses the importance of types for *abstraction.* As Reynolds puts it: "type structure is a syntactic discipline for maintaining levels of abstraction." In this view, we cannot form for instance the intersection or union of two types. The other view, less pure, defines a type as a set of untyped programs, i.e. as a subset of a combinatory algebra D.

From now on, we assume that we have an element $n \in D$ for each natural number $n \in N$ and that we have two constants $s, case \in D$ such that $s\ n = n+1$, $case\ 0\ x\ y = x$ and $case\ n+1\ x\ y = y$.

8.2 Models of Simply Typed Lambda Calculus

A type is interpreted as a subset of D. If $A, B \subseteq D$ we define

$$A \Rightarrow B = \{v \in D \mid (\forall u)\ u \in A \Rightarrow v\ u \in B\}$$

The following result has the same proof as the proof of the fundamental theorem of logical relation. We let $[\![\iota]\!]$ be the set $N \subseteq D$ of elements equal to one n, n natural number, and $[\![\alpha{\rightarrow}\beta]\!]$ to be $[\![\alpha]\!] \Rightarrow [\![\beta]\!]$.

Theorem: If $M \in T_\alpha$ and $\rho(x) \in [\![\beta]\!]$ whenever $x \in V_\beta$ then $M\rho \in [\![\alpha]\!]$.

We can define a constant *natrec* such that $natrec\ 0\ f\ x = x$ and $natrec\ n+1\ f\ x = f\ n\ (natrec\ n\ f\ x)$.

Proposition: We have $natrec\ u\ f\ x \in A$ if $u \in N$, $x \in A$ and $f \in N \Rightarrow (A \Rightarrow A)$.

Proof: By induction on n such that $u = n$. If $n = 0$ we have $natrec\ u\ f\ x = x$ and $x \in A$. If $n = m+1$, then by induction $v = natrec\ m\ f\ x \in A$. Since $natrec\ u\ f\ x = f\ m\ v$ and $f \in N \Rightarrow (A \Rightarrow A)$ we get $natrec\ u\ f\ x \in A$. $\square$

A variation of this model is the following: a type is interpreted as a *partial equivalence relation* (p.e.r.) on D that is as a relation which is symmetric and transitive. We define

$$A \Rightarrow B = \{(v_1, v_2) \in D^2 \mid (\forall u_1, u_2)\ A\ u_1\ u_2 \Rightarrow B\ (v_1\ u_1)\ (v_2\ u_2)\}$$

Proposition: If A, B are p.e.r. on D then so is $A \Rightarrow B$.

Theorem: If $M \in T_\alpha$ and $[\![\beta]\!]\ \rho(x)\ \rho(x)$ whenever $x \in V_\beta$ then $[\![\alpha]\!]\ (M\rho)\ (M\rho)$.

8.3 Application to Normalisation

We take for D the combinatory algebra of all untyped lambda-terms with β, η-conversion. Let $S \subseteq D$ be the set of all *simple* terms that are terms of the form $x\ t_1 \ldots t_k$ where $t_1, \ldots, t_k$ are normalisable, and $N \subseteq D$ be the subset of all normalisable terms. Obviously $S \subseteq N$.

Proposition: If $S \subseteq A \subseteq N$ and $S \subseteq B \subseteq N$ then $S \subseteq A \Rightarrow B \subseteq N$.

Proof: If $t \in S$ then $u \in A$ implies $u \in N$ and hence $t\ u \in S \subseteq B$. Hence $t \in A \Rightarrow B$.
If $t \in A \Rightarrow B$ let x be a variable not free in t. We have $x \in S \subseteq A$ and hence $t\ x \in B$. Since $B \subseteq N$ it follows that $t\ x$ is normalisable and so is $t = \lambda x.\ t\ x$. $\square$

This proposition is the essence of Tait's method for showing normalisation for typed lambda-calculus.

8.4 Models of Dependent Types

We assume that the combinatory algebra D contains special constants $\hat{N}, U \in D$ and $\pi \in D$ such that $\pi\ u_1\ v_1 = \pi\ u_2\ v_2$ implies $u_1 = u_2$ and $v_1 = v_2$. Let T be the set of all subsets of D. Intuitively T is the set of all possible types. We have already defined a special type $N \in T$ and an operation $X \Rightarrow Y \in T$ for $X, Y \in T$.

We define now two functions ψ, ϕ from subsets of D, repectively $S \subseteq D$ (for *small* types) and $L \subseteq D$ (for *large* types). Thus we shall have $\psi : S \rightarrow T$ and $\phi : L \rightarrow T$. First we define S, ψ "inductively":

- $\hat{N} \in S$ and $\psi(\hat{N}) = N \in T$,
- if $A \in S$ and $B\ u \in S$ for all $u \in \psi(A)$ then $\pi\ A\ B \in S$ and $\psi(\pi\ A\ B)$ is the set of all $v \in D$ such that $v\ u \in \psi(B\ u)$ for all $u \in \psi(A)$.

We define next L, ϕ:

- $U \in L$ and $\phi(U) = S$,
- $N \in L$ and $\phi(\hat{N}) = N \in T$,
- if $A \in L$ and $B\ u \in L$ for all $u \in \phi(A)$ then $\pi\ A\ B \in L$ and $\phi(\pi\ A\ B)$ is the set of all $v \in D$ such that $v\ u \in \phi(B\ u)$ for all $u \in \phi(A)$.

Proposition: We have $S \subseteq L$ and ϕ extends ψ, that is $\phi(x) = \psi(x) \in T$ if $x \in S$.

In this way, we get a model of one version of Martin-Löf's type theory with one universe. For instance

Proposition: We have *natrec* $u\ f\ x \in \phi(B\ u)$ assuming $B\ v \in L$ if $v \in N,\ u \in N,\ x \in \phi(B\ 0)$ and $f\ u\ w \in \phi(B\ (s\ u))$ whenever $u \in N,\ w \in B\ u$.

In this model, we have "transfinite types." We can define $\rightarrow X\ Y = \pi\ X\ (k\ Y)$ and we have $\rightarrow : \phi(U) \Rightarrow (\phi(U) \Rightarrow \phi(U)$. If we define T by $T\ 0 = \hat{N} \in U$ and $T\ (n+1) = \rightarrow \hat{N}\ (T\ n)$. We have $T\ u \in L$ if $u \in N$ and hence we can form $\pi\ \hat{N}\ T \in L$.

9 A System with a Type of all Types

What I present now is in some sense the simplest system with dependent types. The presentation will be informal, since I don't precise important issues about handling of names.

The terms of this formal system are the one of *untyped* lambda-calculus, extended with a constant $*$ which will represent the type of all types, the product operation $[x : a]b$ that can be formed for any terms a, b and variable x. It is simple to extend the notion of β-reduction for this calculus, and the proof the this reduction is Church-Rosser (the argument of Tait-Martin-Löf was indeed presented first for such a system [17].) A context will be a sequence of type assignements of the form $x_1 : a_1, \ldots, x_k : a_k$, since types will be represented as special terms, namely terms of type $*$. Using the letters $\Gamma, \Delta, \ldots$ for representing contexts, we write $\Gamma \vdash t : a$ for expressing that in the context Γ, the term t is a correct term of type a. The rule for forming contexts, starting from the empty context, are that Γ can be extended to $\Gamma, x : a$ if x does not occur in Γ and $\Gamma \vdash a : *$. The rules for forming terms are

$$\frac{}{\Gamma \vdash x : a} \qquad x : a \in \Gamma$$

$$\overline{\Gamma \vdash * : *}$$

$$\frac{\Gamma \vdash w : [x : a]b \qquad \Gamma \vdash u : a}{\Gamma \vdash w\ u : [u/x]b}$$

$$\frac{\Gamma, x : a \vdash u : b}{\Gamma \vdash \lambda x\ u : [x : a]b}$$

$$\frac{\Gamma \vdash a : * \qquad \Gamma, x : a \vdash b : *}{\Gamma \vdash [x : a]b : *}$$

Notice that the side condition that appeared in the rules of higher-order logic has disappeared.

It is possible to present such a system formally, and to prove that it has the so-called subject reduction property: if $\Gamma \vdash u : a$ and u reduces to u' by β-reduction then $\Gamma \vdash u' : a$.

What is quite remarkable is that, though we present a reasonably simple logic free functional system, this system actually contains both the presentation of minimal higher-order logic, with an explanation, or justification, of each logical rules. The recognition that both the functional and the logical side of a logic have deep analogies is known as the Curry-Howard isomorphism between propositions and types (and proofs and objects).

The notation used is mainly the one of Automath. The important notion of context is also present there. An important aspect of this system is that it is always possible to work *relative* to a given context. For instance, in a context that introduces an arbitrary type $\iota : *$ we can define

$$\mathsf{Eq} = \lambda x \lambda y\ [P : \iota \rightarrow *]P\ y \rightarrow P\ x$$

that represents Leibniz equality, and a type $[x : \iota]\mathsf{Eq}\ x\ x$ expressing the reflexivity of this equality. We used here the notation $a \rightarrow b$ for $[x : a]b$ in the case where x is not a free variable of b. It can be checked that $\lambda x \lambda P \lambda z\ z$ is a term of type $[x : \iota]\mathsf{Eq}\ x\ x$. This expression is a good notation for the proof that Leibniz equality is reflexive.

Exercice: Show how to represent higher-order logic in the present system. Give the terms corresponding to the proof that Leibniz equality is symmetric.

Exercice: Tries to represent directly one form of Russell's paradox in this formal system, i.e. tries to build a term of type $[x : *]x$.

9.1 Paradoxes

The system above is a variation of a logic presented in [17]. It was found quickly however that this system, as a logical system, is inconsistent. (Remark that the Church-Rosser property shows that this system is not equationaly inconsistent.) Though the notion of type of all types seem a priori suspicious in the light of paradoxes in set theory, this result is more surprising than it may seem because the usual derivation of Russel's paradox or the paradox in Frege's system, cannot be carried out directly here. Indeed, the first derivation of a paradox in this system by Girard was a variation of Burali-Forti paradox (which considers roughly an ordinal of all ordinals, see for instance [3]). I found out later that it was also possible to represent Reynolds' result of the non existence of a set-theoretic model of polymorphism in this system, and that this gives a different paradox, with some better properties. The problem of building a paradox can be formulated in combinatorial terms: build a closed term t of type $[x : *]x$.

Exercice: Explain, using Church-Rosser and the subject reduction property, why such a term cannot have a normal form.

Both derivations lead to complicated terms. Since we get non normalisable terms, it was natural to consider their behaviour by reduction. One may expect a term that eventually reduces to itself, like $(\lambda x\ x\ x)\ (\lambda x\ x\ x)$. However, in both cases, the term gets more and more complex when one reduces them. There is an analogy between the process of reducing a lambda term and the process of understanding the proof that corresponds to it, so that in some sense, these proofs of paradoxes get more and more complicated when one tries to understand them!

Suprisingly in this context, a quite short derivation of a paradox can actually be found, see [15]. In this presentation of the system which uses untyped abstraction, one gets furthermore a term that reduces eventually to itself.

9.2 Martin-Löf Type Theory

Because of this paradox, Martin-Löf introduced a distinction between "small" types, the one that does not mention $*$, and "large" types, the one that mentions $*$. The pure version of this system allows interesting representation of number theoretic functions on the type $N = [x : *](x \rightarrow x) \rightarrow x \rightarrow x$.

Exercice: Show how to represent the exponential function $n \longmapsto 2^n$ on this type.

However, it is then natural to introduce primitive small types, that correspond to data types in a functional programming language. Curry-Howard analogy holds even at this level and we can write a kind of dictionary between functional programming and proof theory:

- introduction rules are represented as constructors,
- canonical proofs are represented by terms starting with a constructor,
- the method of producing a canonical proof from a proof is represented by head-reduction,
- doing a proof by induction over an object is seen as a recursive definition of a proof object,
- derived rules are represented as (recursively) defined constants.

The computational semantics of type theory not only ensures, but gives direct and elegant proofs of the following strong properties

- consistency: the absurd proposition $\perp$ is not provable,
- existence property: if a property P of natural numbers can be represented in type theory, and if $(\exists x)P(x)$ has a proof in type theory, then, from this proof, we can effectively find a natural number n that satisfies the property P.

One most specific aspect of the current version of Martin-Löf is the use of a sigma type $<x : a>\ b$ in some sense dual to proudct types for representing existence. This allows for a natural derivation of the axiom of choice and the epsilon choice operator [19].

10 A Brief Survey of Paradoxes in Formal Systems

We can cite [27]. "The list of logicians who have tried to construct substantial interpreted formal languages adequate for sizable parts of mathematics, but who failed in their early attempts, is impressive: it comprises, among others, Frege, Church, Curry, Quine and Martin-Löf."

We can list here shortly some attempts of designing formal systems for representing consequent portion of mathematics that have found later to result in paradoxes. As we have seen, the first such example was the system of Frege, where the inconsistency was noticed by Russell. The same kind of paradoxes was found later by Curry in a variation of Church's system [4]. The inconsistency found in Church's untyped system was a variant of Richard's paradox.

The paradox of Burali-Forti was used by Rosser to get an inconsistency in one version of Quine's new foundation system (the consistency of the current version is open). A variation of this paradox can be done in the system with a type of all types. In [3] there is a quite concrete version of this paradox. We consider all finite games with two players I and II and then the following "hypergame": player I chooses a finite game, and the game goes on along this choice. Is hypergame finite?

As I indicated Reynolds' theorem on the non existence of set-theoretic models of polymorphism gives another paradox, while Hurkens' paradox is a subtle variation of Russell's paradox.

Summary

The main points that have been stressed and illustrated in this course are:

- the fundamental theorem of logical relation,
- the representation of data types in polymorphic lambda calculus, and representation of inductively defined predicates and relations in second-order logic,
- Curry-Howard analogy between functional and logical systems.

References

[1] P. Andrews. General Models, Descriptions, and Choice in Type Theory. Journal of Symbolic Logic, 37 (1972), p. 385-394.

[2] P. Andrews. General Models and Extensionality. Journal of Symbolic Logic, 37 (1972), p. 395-397.

[3] H. Barendregt. Typed Lambda Calculi. In Handbook of Logic in Computer Science, Abramsky et al eds, Oxford University Press, 1992

[4] A. Church. A Set of Postulates for the Foundation of Logic. Annals of Mathematics, 33 (1932), p. 346-366.

[5] A. Church. A formulation of the simple theory of types. Journal of Symbolic Logic, 5 (1940), p. 56-68.

[6] S. Fortune, D. Leivant and M. O'Donnell. The expressiveness of simple and second order lambda calculus. Journal of the ACM 30 (1983), p. 151-185.

[7] H. Friedman. Equality between Functionals. in Lecture Notes in Mathematics 453, Springer-Verlag, R. Parikh ed., pp. 22-37.

[8] R.O. Gandy. On The Axiom of Extensionality, part I. Journal of Symbolic Logic, 21 (1956).

[9] J.R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda Calculus.* London Mathematical Society, London, 1986.

[10] M. Hofmann. Syntax and Semantics of Dependent Type. In Semantics of Logics of Computation, P. Dybjer and A. Pitts, eds., Cambridge University Press, 1997.

[11] S.C. Kleene. A Theory of Positive Integers in Formal Logic. Part I. American Journal of Mathematics, 57 (1935), p. 153-173.

[12] J.Y. Girard, Y. Lafont, P. Taylor. *Proof and Types.* Cambridge Tracts in Theoretical Computer Science, 7, 1989.

[13] L. Henkin. Completeness in the theory of types. Journal of Symbolic Logic, 15 (1950), p. 81-91.

[14] L. Henkin. The discovery of my completeness proof. Bulletin of Symbolic Logic, 2 (1996), p. 127-158

[15] A.J.C. Hurkens. A Simplification of Girard's Paradox. In LNCS 902, TLCA' 95, M. Dezani and G. Plotkin eds, 1995, p. 266-278

[16] S.C. Kleene and B. Rosser. The inconsistency of certain formal logics. Annals of Mathematics, 36 (1935), p. 630-636.

[17] P. Martin-Löf. A Theory of Types. Technical Report, Stockholm, 1971.

[18] J. C. Mitchell. Type systems for programming languages In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, p. 367-458, 1990.

[19] B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Lf's Type Theory.* Oxford University Press, 1990.

[20] J.C. Reynolds. Towards a Theory of Type Structures. In: LNCS 19, p. 408-425.

[21] J.C. Reynolds. Types, abstraction and parametric polymorphism. In: Information Processing 83, edired by R.E.A. Mason. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1983, pp. 513 - 523.

[22] J.C. Reynolds. Polymorphism is not Set-Theoretic. In Semantics of Data Types, edited by G. Kahn, D.B. MacQueen, and G.D. Plotkin, LNCS 173, Springer-Verlag, 1984, pp. 145 - 156.

[23] W. Tait. A Realizability Semantics of the Theory of Species. in Lecture Notes in Mathematics 453, Springer-Verlag, R. Parikh ed., pp. 240-251.

[24] M. Schönfinkel. On the building blocks of mathematical logic. In "From Frege to Gödel", van Heijenoort, p. 355-366.

[25] D. Scott. Domains and Logics, *extended abstract.* In Logic in Computer Science, 1989, p. 4-5.

[26] A. Stoughton. Mechanizing Logical Relations. LNCS 802, p. 359-377, 1994

[27] G. Sundholm. The General From of Operations in Wittgenstein *Tractacus* Grazer Philosophische Studien, 42 (1992), p. 57-76.

[28] P. Wadler. Theorems for free! ACM Functional Programming Languages and Computer Architecture, 1989, 347-359.

[29] G. Winskel. *The Formal Semantics of Programming Languages, an Introduction.* MIT Press, 1993.

[30] L. Wittgenstein. *Tractacus Logico-philosophicus.* 1921