Gödel's system T as a precursor of modern type theory

Gilles Dowek

Preliminary version

The name "type theory" has been given to several theories that can be classified in two rough categories: the early type theories of Russell, Ramsey, Church, ... and the modern type theories of de Bruijn, Martin-Löf, Coquand and Huet, Paulin, ... A characteristic property of the modern type theories is that proofs are objects of the theory, while they are not in the early type theories. The early type theories have been developed from the beginning of the 20th century to the publication of Church's paper, in 1940, and the modern ones from the first version of Automath, in 1967, to current time.

Gödel's system T has been published in Gödel's 1958 paper, but according to Troelstra's introductory note to this paper, the ideas date back to 1941. A revised version of the paper has been written by Gödel between 1965 and 1972 and Tait's paper on system T has been published in 1967. Thus Gödel's system T has been developed exactly at the time of the transition between early and modern type theories.

In this talk I will not try to give an overview of Gödel's 1958 paper as very detailed presentations have been given, for instance by Troelstra, but instead I will focus on one aspect: on the way Gödel's system T prepares and announces the development of modern type theory.

1 The form of the system T

In its original definition, Gödel's system T is an axiomatic theory. In many aspects, this axiomatic theory is reminiscent of simple type theory: types are simple types, the axioms are logical axioms, the axioms of equality, Peano axioms, ... The main novelty with respect to simple type theory is the fact that the theory contains "rules which permit the definition of a function by an equality with a term constructed from variables and previously defined functions or by primitive recursion with respect to a number variable". Thus, unlike simple type theory that permits only the construction of explicitly definable functions, the system T also permits the construction of functions by induction. The fact that the system T only permits these two constructions allows to view the elements of the type $T \to U$ as algorithms transforming elements of the type T to elements of the type U.

The modern definition of the system T, as a set of terms, has been given later by Tait: the fact that Gödel system T can express natural numbers is replaced by the fact that it contains the symbols 0 and $Succ$, the fact that it allows explicit definitions is replaced by the fact that it contains typed combinators S and K in the style of Curry and the fact that it allows definitions by induction is replaced by a combinator R.

This language contains no predicate symbols, thus there are no formulae and no truth judgments, but reduction rules allowing to compute with terms

$$Kxy \longrightarrow x$$

$$Sxyz \longrightarrow xz(yz)$$

$$Rxf0 \longrightarrow x$$

$$Rxf(Succ\ y) \longrightarrow fy(Rxfy)$$

The termination of this rewrite system has been proved by Tait in 1967, introducing the notion of reducible term. In the same way that Gödel's theory could be seen as an extension of simple type theory, this language can be seen as an extension of the simply typed language of combinators, extended with the construction of functions by induction.

2 The use of the system T

So, if we look at what the system T is made of, we see that in both formulations it is much more connected to the early type theories than to the modern ones. But, if we look now at what the system T is used for, the situation is quite different. The motivation for introducing this theory is to give a consistency proof of arithmetic, setting a proof theoretic objective to the paper, and this proof proceeds by building a translation.

2.1 The translation

Let us first consider classical logic. Any formula A of arithmetic can be embedded into simple type theory and then put in prenex form. The axiom of choice allows to transform the formula $\forall x \exists y\ B(x,y)$ into $\exists f \forall x\ B(x, fx)$ and hence to transform the formula A into a formula A' of the form $\exists f_1 ... \exists f_n\ B(f_1, ..., f_n)$ where $B(f_1, ..., f_n)$ is purely universal. Then, this formula holds if and only if there exists a sequence of functions $F_1, ..., F_n$ such that the purely universal formula $B(F_1, ..., F_n)$ holds. Thus the truth of any formula can be reduced to the truth of a purely universal formula. Let us call a sequence of function $F_1, ..., F_n$ such that $B(F_1, ..., F_n)$ holds a *prenex interpretation* of the formula A.

The order in which we permute quantifiers affects the form of the formula A'. For instance, the formula $(\forall x \exists y \forall z\ P(x,y,z)) \Rightarrow Q$ may be transformed in a first step into $\exists x \forall y \exists z\ (P(x,y,z) \Rightarrow Q)$ and then into $\exists x \exists h \forall y\ (P(x,y,hy) \Rightarrow$

Q). But we may also apply first the transformation to the antecedent of the implication yielding the formula $(\exists g \forall x \forall z\ P(x, gx, z)) \Rightarrow Q$ and then transform it into $\forall g \exists x \exists z\ (P(x, gx, z) \Rightarrow Q)$ and finally into $\exists F \exists H \forall g\ (P(Fg, g(Fg), Hg) \Rightarrow Q)$.

Such a translation can be also carried out in the intuitionistic case although the transformation of a formula into a prenex form is more complicated to justify and requires to use the decidability of equality, Markov's principle, the principle of independence of premises, ... But the benefit of doing this for intuitionistic arithmetic is higher: the arbitrary functions F_1, ..., F_n can be replaced by algorithms and even by algorithms expressed in the system T. For the case of intuitionistic arithmetic, a more precise statement can be proved: if A is provable and $A' = \exists f_1 ... \exists f_n\ B(f_1, ..., f_n)$, then there exists a sequence of algorithms constructed in the system T such that $B(F_1, ..., F_n)$ is provable. The definition of algorithms by induction is needed to interpret the induction scheme.

As the translation of $\perp$ is $\perp$ and there is no empty sequence of algorithms such that $\perp$, arithmetic is proved consistent.

More precisely, if there were a proof of $\perp$ in arithmetic, then we could transform this proof into a proof of $\perp$ in the system T and this proof would contain only purely universal formulae with constants expressing algorithms defined in the system T.

2.2 A hitherto unutilized extension of the finitary standpoint

Of course, the consistency of arithmetic is proved assuming the consistency of a much stronger theory: the system T. Thus this proof does not solve Hilbert's second problem of providing a finitary consistency proof of arithmetic. Yet, it may be argued that this proof is given in a slight extension of finitary mathematics. What is reminiscent of the finitary standpoint is the use of purely universal formulae, but what is an extension of the finitary standpoint is that the objects involved in the constructions are not finite data but algorithms. The fact that adding algorithms is a modest extension of finitary mathematics, specially if we compare it to adding arbitrary functions, is advocated by Gödel in several places: "one can use, for the proof of consistency of number theory, a certain concept of *computable function of finite type over the natural numbers* and some very elementary axioms and principles of constructions for such functions", "The phrase 'well-defined mathematical procedure' is to be accepted as having a clear meaning without any further explanation".

Thus, the value of the translation is to replace quantifier alternation of arithmetic by this easy to grasp notion of "mathematical procedure". The fact that this notion is indeed easier to grasp than quantifier alternation is the main philosophical issue developed in the paper, specially in its second version (in particular in the footnote to a footnote to a footnote **h**). It may also be the weak point of the paper as Troelstra notices : "It was not possible to avoid a certain 'impredicatvity' in the notion of a finite-type function [...] It is this fact which makes it difficult to formulate the epistemological gain obtained in replacing the general concept of intuitionistic proof by ‘computable function of finite type’ ”.

The very project of finding an extension of the finitary standpoint where the consistency of arithmetic can be proved may be interrogated. As we know from the second incompleteness theorem, there is no way to prove the consistency of arithmetic in arithmetic itself and *a fortiori* in finitary mathematics. Thus, something must be added to the finitary standpoint to prove the consistency of arithmetic. The idea is to add an existence axiom allowing to prove the consistency of arithmetic, the object whose existence is asserted by the axiom being “easy to grasp” or “intuitive”.

If we give the word “intuition” its usual meaning in mathematics: we have the intuition that some formula is true when we do not have a proof yet, but we tend to think that such a proof exists, then there is nothing that we may have the intuition of the existence of, and whose existence entails the consistency of arithmetic, because the fact that the existence of this object entails the consistency of arithmetic shows that this existence cannot be proved in arithmetic and hence that our inclination to think that it could was a mistake.

Then, if we give to the word “intuition” its platonist meaning of direct access to an object, it is always arguable that we have a more direct access to this object than to the set of natural numbers, that is sufficient to build a model of arithmetic.

Thus, it is not only not clear what the epistemological gain of replacing the set of natural numbers by this notion of “computable function of finite type” is, but it is also not clear also what the epistemological gain could be expected when extending the finitary standpoint to prove the consistency of arithmetic.

2.3 From proofs to algorithms

Thus, the value of the system T must be found somewhere else than in an hypothetical easy to grasp extension of the finitary standpoint where arithmetic could be proved consistent.

The proof that whenever the formula A is provable then there exists a sequence of algorithms $F_1, ..., F_n$ such that $B(F_1, ..., F_n)$ is provable is a simple induction on the proof of A and the structure of the algorithm reflects the structure of the proof. For instance, if A' is $\exists y\ \forall z\ C(y, z)$ and B' is $\exists v\ \forall w\ D(v, w)$ then, by definition $(A \Rightarrow B)' = \exists f \exists g \forall y \forall w (C(y, gyw) \Rightarrow D(fy, w))$. If from a proof of $A \Rightarrow B$ and a proof of A we build a proof of B, using the *modus ponens* rule, then by induction hypothesis, we have algorithms Y, F and G and proofs of $\forall z\ C(Y, z)$ and $\forall y \forall w\ (C(y, Gyw) \Rightarrow D(Fy, w))$. Then, the algorithm associated to the proof of B is FY. To prove the formula $\forall w\ D(FY, w)$, we take an arbitrary w and we prove $D(FY, w)$. To do so, we apply $\forall y \forall w\ (C(y, Gyw) \Rightarrow D(Fy, w))$ to Y and w yielding $C(Y, GYw) \Rightarrow D(FY, w)$ and $\forall z\ C(Y, z)$ to GYw yielding $C(Y, GYw)$ and we conclude with the *modus ponens* rule.

This application of the algorithm F to Y is reminiscent of the Brouwer-Heyting-Kolmogorov interpretation, where the *modus ponens* rule is interpreted by application. Thus, another way to view Gödel's translation is to associate, to each formula A of intuitionist arithmetic, a formula containing an extra free variable f, that we can write $f \Vdash A$, expressing that f is a proof of A according to the Brouwer-Heyting-Kolmogorov interpretation.

Actually, as unlike modern presentations of the system T, the version used by Gödel does not have pairs, f is not a single variable, but a sequence of variables (this idea anticipating the Schütte's translation of terms of the system T with pairs to sequences of terms of the system T without pairs).

The oddities of the translation for an eye used to the Brouwer-Heyting-Kolmogorov interpretation are due partly to this lack of pairs, partly to the fact that the formulae $f \Vdash A$ are kept universal.

For instance, if the formula $y \Vdash A$ is $\forall z\ C(y,z)$ and the formula $V \Vdash B$ is $\forall w\ D(V,w)$, then we would define $f \Vdash (A \Rightarrow B)$ by saying that f is an algorithm that transforms any proof of A to a proof of B

$$f \Vdash (A \Rightarrow B) = \forall y\ (y \Vdash A \Rightarrow fy \Vdash B)$$

i.e.

$$f \Vdash (A \Rightarrow B) = \forall y\ (\forall z\ C(y,z) \Rightarrow \forall w\ D(fy,w))$$

but this formula is not universal and it is replaced, in a first step, by

$$\forall y \forall w \exists z\ (C(y,z) \Rightarrow D(fy,w))$$

and in a second by

$$\exists g \forall y \forall w\ (C(y,gyw) \Rightarrow D(fy,w))$$

Finally, the algorithm g is included in the proof yielding

$$f,g \Vdash (A \Rightarrow B) = \forall y \forall w\ (C(y,gyw) \Rightarrow D(fy,w))$$

However, if we compare Gödel's translation to the Brouwer-Heyting-Kolmogorov interpretation, we can see that the algorithms built in this translation do not contain all the information contained in the proofs and this explains why the formula $F \Vdash A$ requires a proof. For instance, the algorithm F such that $F \Vdash \forall x \exists y\ (\neg x = 0 \Rightarrow x = Succ(y))$ is the predecessor function but the formula

$$\forall x\ (\neg x = 0 \Rightarrow x = Succ(Pred\ x))$$

requires a proof. Using, the fact that point-wise equality of two recursive primitive functions is undecidable, we can elaborate on this example and show that the formula $F \Vdash A$ cannot be decided in general.

The fact that the algorithms contain only part of the information contained in the proofs is even more visible in the clauses of quantifiers. A "proof" of a formula $\forall x\ P(x)$ where P is atomic is not a function mapping x to a proof of $P(x)$, but is empty. A "proof" of a formula $\exists x\ P(x)$ where P is atomic is not a pair formed with a witness and a proof the that witness verifies the formula P, but just a witness.

The fact that these two ideas — the prenex interpretation and the Brouwer-Heyting-Kolmogorov interpretation — lead to the same translation is a consequence of the fact that the axiom of choice can be derived once we have the Brouwer-Heyting-Kolmogorov interpretation. It is not clear to me which of these two ideas Gödel had in mind, but we can see objectively that the fact that algorithms are built by translating proofs connects this translation to the Brouwer-Heyting-Kolmogorov interpretation and hence the system T to modern type theory, while the fact that these algorithms are still incomplete proofs connects it to the prenex interpretation and hence to early type theories.

This partiality has been remarked by Gödel: "Of course, it is not claimed that Definition 1-6 [the translation] express the meaning of the logical particles introduced by Brouwer and Heyting. The question to what extent they can replace them requires closer investigation." This sentence seems to suggest that the starting point of the system T was rather the prenex interpretation, but that it was already understood that further investigations might connect it to the Brouwer-Heyting-Kolmogorov interpretation.

2.4 Another extension of the finitary standpoint

Later developments of the system T, such as the introduction of the system T as a set of terms, lead to a even more modest extension of the finitary standpoint. In these extensions, the grasp of a general notion of computable function of finite type is not needed anymore, but this abstract notion of computable function of finite type could be replaced by a finite piece of data: a normal term of the system T.

The process of associating a normal term to a proof is not finitary because if we associate a term F to a proof of $A \Rightarrow B$ and a term Y to a proof of A, we cannot associate the term FY to the proof of B obtained by the modus ponens rule, but we need to associate to this proof the normal form of the term FY. Adding to finitary mathematics this algorithm to compute the normal form of a term is another extension of finitary standpoint that allows to prove the consistency of arithmetic.

We could also imagine to associate to each proof a not necessarily normal term of system T, but the termination of such terms would still be needed to be able to prove that if A is provable in arithmetic then A' is provable in the system T.

2.5 A digression on the second incompleteness theorem

With our modern view, associating a normal term to each proof is the same thing as restricting to cut free proofs in arithmetic. If we define the provability predicate $Bew(x)$ in such a way the $Bew(\ulcorner A \urcorner)$ means that A has a cut free proof, then the formula $\neg Bew(\ulcorner \bot \urcorner)$ has a finitary proof and hence a proof in arithmetic. This seems to contradict the second incompleteness theorem, but it does not, as the predicate Bew defined this way does not fulfill the Hilbert-

Bernays conditions, in particular the second condition that the formula

$$Bew(\lceil A \Rightarrow B\rceil) \Rightarrow Bew(\lceil A\rceil) \Rightarrow Bew(\lceil B\rceil)$$

must be provable in the theory itself.

Indeed, if we have a normal proof π of $A \Rightarrow B$ and a normal proof of π' of A, we need, in order to build a normal proof of B to normalize the proof $\pi\pi'$ and the termination of this process cannot be proved in arithmetic.

Again, adding this algorithm to the finitary standpoint allows to prove the consistency of arithmetic.

Instead of being phrased as the fact that theories that can prove their own closure by modus ponens cannot prove their own consistency, the second incompleteness theorem could be phrased as the fact that theories that can prove their own consistency cannot prove their own closure by *modus ponens*. And the extensions of the finitary standpoint considered above are various ways to phrase this closure of the theory by *modus ponens*.

3 Paving the way to modern type theory

As already said, the value of the system T must be found somewhere else than in an hypothetical easy to grasp extension of the finitary standpoint where arithmetic could be proved consistent. A point I will not discuss in this talk is how the system T paved the way to relative consistency proofs and realizability models. Another point I will not discuss is how it paved the way to the notion of computational content of a proof. Instead, I will mainly focus on one point: how the system T paved the way to modern type theory, trying to clarify what is already in the system T and what is not. I will also add a few words at the end on the notion of inductive type anticipated by the system T.

3.1 Proofs are algorithms

The main idea of modern type theory is that the proofs should be objects of the theory. In set theory, or in early type theories, the number 2 is an object, the set of even numbers is an object, the function multiplying its argument by 2 is an object, but the proof that 2 is an element of the set of even numbers is not. In modern type theories, in contrast, it is. Modern type theories also agree on the fact that a proof is not just any object of the theory, but that it is an algorithm, through the Brouwer-Heyting-Kolmogorov interpretation. In particular, a proof of a formula $A \Rightarrow B$ is an algorithm mapping proofs of A to proofs of B.

In this respect, modern type theory changes the perspective on proofs. Proofs were considered, by the axiomatic method, as the very first notion of mathematics. Now proofs are defined in term of a more primitive notion: the notion of algorithm. This means that an elementary theory of algorithm has to be developed before, and without, the notion of proof.

We can see that the system T, already in Gödel's formulation anticipates these ideas: first the idea that the notion of algorithm is more fundamental than that of proof, then that proofs must be objects of the theory, and finally that they should be algorithms.

All these ideas, if anticipated, are not yet expressed in their full clarity. For instance, the fact that algorithms are more fundamental than proofs is anticipated as this interpretation allows to reduce the existence of a proof to the existence of an algorithm, but it is not completely implemented because the formula $F \Vdash A$ still requires a proof.

The same holds for the fact that proofs should be objects of the theory. Indeed, neither arithmetic, nor the system T can express its own proofs, but the proofs of one theory are expressed in the other. This is why the extensions of this interpretation to simple type theory are important, as they are a first step towards a theory that expresses its own proofs.

Finally, if proofs are translated to algorithms, they are not expressed as algorithms through the full Brouwer-Heyting-Kolmogorov interpretation.

3.2 Computation rules

Another property of the system T that anticipates modern type theory is that it is the first extension of simple type theory where most of the computable functions can be expressed. It is well-known that Church's initial goal with λ-calculus was to provide a formalization of mathematics where all computable functions could be computed, but that the discovery of the Kleene-Rosser paradox lead to split the project: the λ-calculus to express all computable functions and Church's theory of type, to formalize mathematics. We know, by a result of Schwichtenberg, that the computational power of the simply typed β rule is much weaker than the untyped one. Strangely enough, the fact that simple type theory could be extended in such a way that all primitive recursive functions could be computed does not seem to have been put forward before the system T.

Of course, both in the system T and in modern type theory, this computational power is a prerequisite for proofs to be expressed as algorithms.

3.3 Proofs are finite

When a theory can express its own proofs, to prove that A is a theorem we may provide a proof π of A expressed in the same language, but we need also to prove the formula expressing that π is a proof of A, to do so we may provide a proof π', but then we need to prove that π' is a proof that π is a proof of A, and so on. To avoid this infinite regression, we need to stop at some point and make the fact that π is indeed a proof of A a decidable judgment.

If we want the fact that π is a proof of A to be decidable, then we first need the proof π to be given as a finite datum, *i.e.* we cannot just prove the existence of π by arbitrary means, but we have to express it by a term that can be processed by the proof-checking algorithm.

This idea that proofs should be finite constructions is much clearer in Tait's paper than in Gödel's. However, Gödel's paper already speaks about "rules which permit the definition of ..." and not existence axioms and it explicitly says that the intention is to replace the existentially bound variables by such constructions and not just to prove this existence by arbitrary means: "to be more precise, if $F' = (\exists y)(z)A(y, z, x)$, then a finite (possibly empty) sequence Q of functional constants can be defined in T such that $A(Q(x), z, x)$ is provable in T." Although explicit in both versions of the paper, more emphasis on this point is put in the second version of the paper.

If this idea that proofs should be finite constructions is anticipated the idea that proof-checking should be decidable is not there yet. In particular because algorithms contain only a partial information on proofs.

3.4 Full and partial proofs

We have said several times that, in Gödel's translation, algorithms contain only partial information in proofs. For instance, a proof of a formula $\exists x\ P(x)$ where $P(x)$ is atomic is just a natural number n and not a pair formed with a natural number n and a proof of $P(n)$. This point can be connected to a point we shall discuss later: the lack of dependent types, but it may be discussed independently, as, in modern type theory, dependent types can be eliminated. For instance, the full proof of the formula

$$\forall x \exists y\ (\neg x = 0 \Rightarrow x = Succ(y))$$

can be expressed in system T with pairs, by deciding that proofs of atomic formulae and of $\bot$ have the same type as natural numbers. Then, the proofs of

$$\forall x \exists y\ (\neg x = 0 \Rightarrow x = Succ(y))$$

have type

$$nat \to (nat \times ((nat \to nat) \to nat))$$

and a proof expressed in system T with pairs and where combinators are replaced by λ-calculus is

$$R\ \langle 0, \lambda\alpha\ (\delta(\alpha(\mathit{refl}\ 0)))\rangle\ \lambda z \lambda\beta\ \langle z, \lambda\gamma\ (\mathit{refl}\ (Succ\ z))\rangle$$

where refl and δ are constants of type $nat \to nat$, z is a variable of type nat, α and γ of type $nat \to nat$ and β of type $nat \times ((nat \to nat) \to nat)$. Thus, it is more the lack of pairs and the burden of mixing pairs with combinators that forbid to use the full Brouwer-Heyting-Kolmogorov interpretation than the lack of dependent types.

When algorithms contain the full proofs, proof-checking becomes decidable, even without dependent types. We just have to give the formula the constants and the variables are hypothetical proofs of: refl is a proof of $\forall x\ x = x$, δ maps a proof of $\bot$ to a proof of $0 = S(0)$, α is a proof of $\neg 0 = 0$, β a proof of $\exists y\ (\neg z = 0 \Rightarrow z = S(y))$, γ a proof of $\neg S(z) = 0$ and R maps a proof of

$\exists y\ (\neg x = 0 \Rightarrow 0 = Succ(y))$ and a proof of $\forall z\ ((\exists y\ (\neg x = 0 \Rightarrow z = Succ(y))) \Rightarrow (\exists y\ (\neg Succ(z) = 0 \Rightarrow Succ(z) = Succ(y))))$ to a proof of $\forall x \exists y\ (\neg x = 0 \Rightarrow x = Succ(y))$. Then we can bottom-up associate a formula to each subterm of the proof-term and compute the formula proved by this term.

3.5 The types of proofs

Of course, as remarked by de Bruijn and Howard, the similarity of this process with type-checking leads to replace the types by the formulae and give the type $\forall x\ (x = x)$ rather than $nat \rightarrow nat$ or $nat \rightarrow\ =$ to the constant *refl*, introducing at the same time dependent types and the Curry-de Bruijn-Howard correspondence between types and formulae.

Then, we can associate to each formula A, not a formula $f \Vdash A$ such that when A is provable then there exists a term F such that $F \Vdash A$ is provable, but a type A' such that when A is provable then there exists a term of type A'. The fact that a term has a given type is decidable and require no proof.

In Gödel's translation, the type of the algorithm expressing a proof is very remote from the formula and this analogy between types and formulae does not seem to be understood yet.

Once this correspondence is understood, keeping the formula $\pi \Vdash A$ universal loses its importance. The negative occurrence of the universal quantifier $(\forall x\ P(x)) \Rightarrow Q$ is not more harmful than the use of higher types algorithms.

A final point that lacks in system T to make it a modern type theory is the relation between reduction and cut elimination.

To sum up, the system T anticipates that proofs are algorithms expressed by finite terms but the fact that algorithms contain only part of the proofs, probably a consequence on the fact that the starting point is closer to the prenex interpretation than to Brouwer-Heyting-Kolmogorov interpretation, forbids to achieve the decidability of proof-checking and the correspondence between formulae and types.

4 Inductive types

Another important feature of modern type theory that is introduced in system T is the notion of inductive types, *i.e.* types, such as the type of natural numbers, equipped with a recursor R and the associated computation rules.

It has been known since the thirties that all computable functions can be expressed in λ-calculus and hence that no rule besides (untyped) β-reduction was needed. It was also known that simply typed β-reduction was much less powerful, although the precise characterization of the definable functions in typed λ-calculus was only given in 1976 by Schwichtenberg.

Thus, to design programming languages that would be at the same time powerful enough to express proofs through Brouwer-Heyting-Kolmogorov interpretation and always terminating, two ways are possible. The fist is to keep

β-reduction as the only computation rule and to extend the typing rules in order to type more untyped terms. The second is to add more computation rules. The second option has been initiated by the system T as extra rewrite rules are required for the functions constructed by induction and leads to the modern notion of inductive types. The choice of one option or the other is an important difference between various modern type theories. As we have said a lot about the system T, let us turn to the other option: typing more terms.

Let us reintroduce in arithmetic Peano's predicate N for natural numbers and write the induction scheme not

$$P(0) \Rightarrow \forall x\ (P(x) \Rightarrow P(S(x))) \Rightarrow \forall n\ P(n)$$

but

$$P(0) \Rightarrow \forall x\ (P(x) \Rightarrow P(S(x))) \Rightarrow \forall n\ (N(n) \Rightarrow P(n))$$

or equivalently

$$\forall n\ (N(n) \Rightarrow P(0) \Rightarrow \forall x\ (P(x) \Rightarrow P(S(x))) \Rightarrow P(n))$$

Let us also introduce a sort for classes of natural numbers and a way to construct classes in comprehension and let us transform this axiom into

$$\forall n\ (N(n) \Rightarrow \forall p\ (0 \in p \Rightarrow \forall x\ (x \in p \Rightarrow S(x) \in p) \Rightarrow n \in p)$$

Then, the converse of this axiom becomes provable and we can formulate it in an equivalent way

$$\forall n\ (N(n) \Leftrightarrow \forall p\ (0 \in p \Rightarrow \forall x\ (x \in p \Rightarrow S(x) \in p) \Rightarrow n \in p)$$

This formula can be read as a definition of the symbol N and, with this definition, we do not need an induction axiom anymore. From a proof π of $N(n)$, a proof a of $0 \in p$ and a proof f of $\forall x\ (x \in p \Rightarrow S(x) \in p)$ we can build the proof πpaf of $n \in p$. As the proof π of $N(n)$ is just Church's numeral associated to n, the functions defined by induction are expressed exactly as in pure λ-calculus.

The type system has been extended because the definition of N identifies the type $N(n)$ with

$$\forall p\ (0 \in p \Rightarrow \forall x\ (x \in p \Rightarrow S(x) \in p) \Rightarrow n \in p)$$

and taking a way to define classes in comprehension, for instance a class E of even numbers, also identifies the types $x \in E$ and $\exists y\ (N(y) \wedge x = 2 \times y)$. Extending dependently typed λ-calculus by identifying types this way leads to a calculus that is just an alternative presentation of Girard's polymorphic λ-calculus.

This way of constructing functions by induction, as opposed to inductive types as in the system T, has long suffered from two drawbacks, that have justified the presence of inductive types in Martin-Löf's type theory or in the Calculus of Inductive Constructions, for instance. The first drawback is that this way of dealing with induction seems to have a necessary connection with the impredicative comprehension axiom of second order arithmetic. The second drawback is that the predecessor function cannot be programmed with an efficient algorithm.

In fact, these two drawbacks can been circumvented. First, a folklore result, attributed to Takeuti, is that the variant of second-order arithmetic where the construction of classes in comprehension is restricted to formulae containing no class quantifiers is a conservative extension of arithmetic. Thus, introducing classes of natural numbers is independent of impredicativity. This idea has also been exploited by Leivant in his stratified polymorphic λ-calculus.

For the second, the well-known problem is that Church's numerals allow direct constructions for the functions given by an inductive definition of the form

$$f0 = a$$

$$f(Succ\ n) = g(fn)$$

but only indirect ones for those given by an inductive definition of the form

$$f0 = a$$

$$f(Succ\ n) = gn(fn)$$

so, the function mapping n to 2^n has a direct construction, but neither the function mapping n to $n!$ nor the predecessor function. These functions have only indirect constructions, *à la* Kleene, but the algorithms given by these indirect constructions are not efficient, in particular the computation of the predecessor of n requires a number of steps proportional to n and not independent of n like in the system T. This problem has been solved by Parigot who has proposed to identify the formula $N(n)$ not with

$$\forall p\ (0 \in p \Rightarrow \forall x\ (x \in p \Rightarrow S(x) \in p) \Rightarrow n \in p)$$

but with

$$\forall p\ (0 \in p \Rightarrow \forall x\ (N(x) \Rightarrow x \in p \Rightarrow S(x) \in p) \Rightarrow n \in p)$$

Then, the proof of $N(n)$ is not Church's numeral associated to n, but a modification of it: if π is the proof of $N(n)$, then the proof of $N(Succ\ n)$ is not $\lambda p \lambda \alpha \lambda \beta\ (\beta\ n\ (\pi p \alpha \beta))$ as with Church's numerals but $\lambda p \lambda \alpha \lambda \beta\ (\beta\ n\ \pi\ (\pi p \alpha \beta))$. Again, if π is a proof of $N(n)$, a is a proof of $x \in p$ and f is a proof of $\forall x\ (N(x) \Rightarrow x \in p \Rightarrow S(x) \in p)$ then $\pi p a f$ is a proof of $n \in p$.

Parigot's numerals allow to simulate all algorithms of system T, in particular the one step predecessor. But they have in turn a drawback, the size of the numeral n is proportional, not to n, but to the exponential of n. To circumvent this drawback, we can decide to use an abbreviation for the proof of $N(n)$ and write for instance $[n]$ for the pair formed by the number n and the proof of $N(n)$. Then, instead of unfolding this definition, we can use derived reduction rule. If we write $Rpafk$ for $snd(k)paf$ and allow the notation $f\langle n, \pi\rangle$ for $fn\pi$ then these rules write

$$Rpaf[0] \longrightarrow a$$

$$Rpaf[Succ(n)] \longrightarrow f[n](Rpaf[n])$$

i.e. they are exactly the rules of system T. Thus the different kind of modern type theories may be unified, if we decompose both inductive types and polymorphism into smaller entities.

This shows that this question, opened by Gödel's system T, of the representation of proofs by induction as algorithms is not over yet.

Acknowledgments

To Benjamin Werner for his careful reading and comments on an earlier draft of this paper.