N.G. de Bruijn's Contribution to the Formalization of Mathematics

Herman Geuvers
Radboud University Nijmegen, The Netherlands
Eindhoven University of Technology, The Netherlands
herman@cs.ru.nl

Rob Nederpelt
Eindhoven University of Technology, The Netherlands
r.p.nederpelt@tue.nl

1 Introduction

N.G. de Bruijn was one of the pioneers to explore the idea of using a computer to formally check mathematical proofs. The Automath project, that started in 1967 and ran until 1980, was the first in developing computer programs to actually check mathematical proofs. But Automath is more than that: it is a language for doing mathematics and it has philosophical implications for the way we look at logic and the foundations of mathematics.

In the beginning, Automath did not get very much attention from mathematicians, computer scientists or logicians. In the meantime, computer assisted proof-checking and interactive theorem proving have gained momentum, notably in computer science, for the formal verification of properties of software and hardware. Developers of these systems, usually called *proof-assistants*, pay tribute to de Bruijn as one of the founding fathers of the field. In the present paper, we outline the technical contributions that de Bruijn has made to the field and how they have been incorporated in present day proof-assistants. As de Bruijn was really a pioneer, inventing his own formalism and notions in relative isolation, a lot of the technical contributions of de Bruijn have only become known through the later work of others, or were reinvented independently by others.

In the present paper we want to lay out the motivations of de Bruijn and explain the main technical contributions that the Automath project has made to proof checking. Some of de Bruijn's original ideas are looked at differently now, also because computers are ubiquitous, have become so much faster and memory seems unlimited. Nevertheless, these ideas are still relevant and some have regained momentum again recently.

2 History

John McCarthy, the creator of Lisp and the founding father of the field of AI, was the first to propose the idea of computer checked proofs. In [McCarthy 1960], he explicitly states that Lisp was (among other things) developed for

- Writing a program to check proofs in a class of formal logical systems.
- Writing programs to realize various algorithms for generating proofs in predicate calculus.

In [McCarthy 1962], he makes another clear case for using a computer for checking proofs, especially for the verification of programs:

> It is part of the definition of formal system that proofs be machine checkable. ... For example, instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties.

McCarthy also intended to use Lisp for proof search: he was of course aware of Gödel's incompleteness result, but he anticipated that for various practically relevant cases it would still be feasible to do proof search. The ideas of McCarthy found their way into AI and all kinds of techniques to automate proof search and the automatic verification of programs, like finding program invariants.

The idea of building a proof checking system for *mathematics in general* only really established itself in the late 1960s. It is interesting to see that around 1970 at various places the idea came up to use computers for checking general mathematical proofs. We give a short overview of these, to briefly introduce the main ideas behind these approaches.

- **Automath** De Bruijn (Eindhoven, The Netherlands). The Automath project [Nederpelt et al. 1994, De Bruijn 1968] was initiated by de Bruijn in 1967. We will review it in the rest of this paper.
- **Evidence Algorithm** Glushkov (Kiev, Ukrain) In the end of the 1960s, the Russian mathematician Glushkov started investigating automated theorem proving, with the aim of formalizing mathematical texts. This was later called the *Evidence Algorithm*, EA, and the first citation is in Russian in the journal *Kibernetika* in 1970. The EA project seems to have run since the beginning of the 1970s, but it hasn't become known outside of Russia and Ukraine and publications about the project are all in Russian. The project has evolved into a *System for Automated Deduction*, SAD [Lyaletski et al 2004], which checks mathematical texts written in the language *ForTheL* (FORmal THEory Language) [Verchinine et al 2008]. The latter is a formal language for writing definitions, lemmas and proofs, very much in the spirit of the Mizar language (see below), but developed independently from it.
- **Mizar** Trybulec (Białystok, Poland) The Mizar system [Mizar] has been developed by Trybulec since 1973 at the university of Białystok. It is the longest continuously running proof assistant project. One of its main goals is to create and maintain the *Mizar Mathematical Library* MML, as a coherent repository of formalized mathematics. MML is by far the largest of its kind. Basically, Mizar consists of the Mizar language and the Mizar system itself, which is the computer program that checks text files written in the Mizar language for mathematical correctness. The Mizar language aims at being a formal language close to ordinary mathematical language, so it should also be (human) readable as such. The underlying formal logic is Tarski-Grothendieck set theory with classical logic.
- **Nqthm** Boyer and Moore (Austin, Texas). Nqthm [Boyer and Moore 1998], also known as the "Boyer-Moore theorem prover" is geared towards *automated theorem proving*. The first version originates from 1973 and was very much inspired by the work of McCarthy [McCarthy 1962]. The logic of Nqthm is quantifier-free first order logic with equality, basically primitive recursive arithmetic, which makes the automation very powerful. The disadvantage is that one often has to rewrite a formula (e.g. getting rid of the quantifiers via Skolemization) before being able to prove it. The system Nqthm has evolved into the system ACL2 [Kaufmann, Manolios and Moore 2000], which is used for hardware and software verification.
- **LCF** Milner (Stanford; Edinburgh) LCF [Gordon et al 1979, Gordon 2006] stands for *Logic for Computable Functions*, the name Milner gave to a theory defined by Scott in 1969 (see [Scott 1993]) to prove properties of recursively defined functions. The basic idea of Milner's so called *LCF approach* is to have an abstract data type of theorems `thm`, where the only constants of this data type

> are the axioms and the only functions to this data type are the inference rules. In this way, the only terms of type `thm` are derivable sequents $\Gamma \vdash \varphi$, because there is no other way to construct a term of type `thm` then via the inference rules. Another advantage of the LCF approach is that a user can program tactics by writing more sophisticated functions that produce terms of type `thm`. Due to the abstract data type approach, these will still be derivable with the original set of inference rules, so one cannot compromise the consistency of the system. The systems Isabelle [Wenzel et al 2008], HOL [Gordon & Melham 1993] and HOL-light [HOL-light, Harrison 2009] are all descendants from LCF that use the LCF approach.

Around the same time, there was also a lot of development in type theory, due to the work of Girard [Girard 1972] and Tait [Tait 1967], originating from proof-theoretic studies into higher-order logic and higher-order arithmetic, and the work of Martin-Löf ([Martin-Löf 1984]) on intuitionistic type theory, to lay a foundation for intuitionistic mathematics.

The connection between logic and type theory goes via the so called *propositions-as-types* embedding, which goes back to Curry ([Curry and Feys 1958]), who stated it for minimal implicational logic. The propositions-as-types embedding was extended to predicate logic and arithmetic by Howard in 1969 (but the paper was only officially published in 1980, [Howard 1980]) and to higher-order logic and arithmetic by Girard [Girard 1972]. The embedding identifies a proposition A with the type of its proofs, and so proofs (logical derivations) become first class objects that can be manipulated and checked, via type-checking. De Bruijn basically reinvented this idea in his own way, as we shall discuss in Section 4.

Martin-Löf used the propositions-as-types concept to give a formal account of the so called *Brouwer-Heyting-Kolmogorov interpretation* of proofs as constructions. In his intuitionistic type theory, the inference rules are motivated as being object constructions. Inductive types and functions defined by structural recursion then naturally become basic principles [Martin-Löf 1984, Nordström et al 1990]. This goes also back to work of Scott [Scott 1970], who had noticed that the Curry-Howard isomorphism could be extended to incorporate induction principles.

The addition of inductive types and structural recursion as basic principles (but also the addition of higher order logic) increases the expressivity and the logical power considerably. Modern implementations of proof assistants based on type theory use a combination of inductive types and higher order logic. Examples are Coq, [Coq], Agda [Agda] and NuPrl [Constable et al 1986]. It is noteworthy that de Bruijn has always been an advocate of *weak frameworks* where the logical or computational power is introduced by the user and not built-in to the system. We will come back to this in Section 5.

3 The aim of Automath

One of the first writings by de Bruijn on Automath is [De Bruijn 1968], which starts with

> AUTOMATH is a language for expressing detailed mathematical thoughts. It is <u>not</u> a programming language, although it has several features in common with existing programming languages. It is defined by a grammar, and every text written according to its rules is claimed to correspond to correct mathematics. It can be used to express a large part of mathematics, and admits many ways of laying the foundations. The rules are such that a computer can be instructed to check whether texts written in the language are correct. These texts are not restricted to proofs of single theorems; they can contain entire mathematical theories, including the rules of inference used in such theories.

Already here we see de Bruijn's focus on *language* as the crucial aspect of computer formalization. We also see the idea that one can formalize any kind of mathematics, independent of the foundational view one has, because the system should admit "many ways of laying a foundation". The language should allow a computer to check correctness of a text, which then implies the correctness of the mathematics described therein. Finally, deduction rules need not be part of the system, because they can be added as part of the text to be checked.

De Bruijn distinguished two roles of proofs in mathematics:

1. A proof explains: why? The goal is to get an *understanding* of the mathematics.
2. A proof argues: is it true? The goal then is the *verification* of the mathematics, or phrased differently, the purpose of the proof is to be *convincing*.

Notably for (2), computer support can be helpful, according to de Bruijn. In [De Bruijn 1994], he explains one of the origins of his idea of proof checking, which was really meant to have the possibility to verify long and ugly proofs, e.g. with many case distinctions and repetitions.

> In a short paper by E.W. Dijkstra on a number of processes that might sometimes block one another, the correctness of the algorithm was explained in a paragraph that ended with the remarkable sentence: "And this, the author believes, completes the proof". Indeed, the argument was a bit intuitive. I took it as a challenge and tried to build a proof that would be acceptable for mathematicians. What I achieved was long and very ugly. It might have been improved by developing efficient lemmas for avoiding the many repetitions in my argument, but I left it as it stood. Instead of improving the proof I got the idea that one should be able to instruct a machine to verify such long and tedious proofs.

De Bruijn did not make any claim that the machine would "understand" the mathematics. Quite the opposite: the machine had no understanding or creativity. De Bruijn did not identify the formalization of mathematics with doing mathematics, nor did he identify the framework of formalized mathematics with mathematics itself: he considered formalized mathematics just (a poor) part of mathematics. Nevertheless, through the ages, mathematicians have put down their ideas in linguistic formalisms, to free their mathematics from inclarities or uncertainties, to communicate it with each other and to maintain it. So a formally checked text is a very important asset, but this final text is the closing of the process of mathematical thought, it is not the mathematical thought process itself.

4 PAT: proofs-as-terms

The propositions-as-types and proof-as-terms concept was originally due to Curry, who noticed the following.

- The correspondence between the axioms of minimal implicational logic and the types of the combinators **I**, **K** and **S**.
- The correspondence between modus ponens (in logic) and application (of terms).

De Bruijn wasn't aware of this and reinvented it in his own way. To pinpoint at his contribution and to make this paper self-contained for those not familiar with the propositions-as-types concept, we briefly summarize the ideas of Curry, which were later extended by Howard.

4.1 Lambda calculus and combinators

The idea of λ-calculus is that *abstraction* and *application* are the basic mechanisms for dealing with functions. The untyped λ-calculus does that in its most primitive form: there is *only* abstraction and application and there are no restrictions (e.g. types) that limit the application. So the syntax is, in grammar format

$$\Lambda ::= \mathcal{V} \mid (\lambda \mathcal{V}.\Lambda) \mid (\Lambda\Lambda),$$

where $\mathcal{V}$ is a set of variables, $x, y, z, \ldots$, λ is a special symbol to denote *abstraction* and *application* is denoted by juxtaposition. The idea is that, if M is a term (i.e. $M \in \Lambda$) and x is a variable, then $\lambda x.M$ is the function f with $f(x) = M$, i.e. the function usually denoted by $x \mapsto M$. One peculiarity of the λ-calculus is that one doesn't write $f(x)$ for the application of f to x, but $f\,x$. The λ is a *variable-binder* which means that we identify e.g. $\lambda x.x$ and $\lambda y.y$, because they can be obtained from each other by a simple renaming of the bound variable x into y. Variables that are not bound are called *free*, so in the term $\lambda x.(xy)$, the variable y is free[1]. A term without free variables is also called a *combinator*. Here are three well-known examples of combinators.

$$\begin{array}{rcl} \mathbf{I} & := & \lambda x.x, \\ \mathbf{K} & := & \lambda x.\lambda y.x, \\ \mathbf{S} & := & \lambda x.\lambda y.\lambda z.xz(yz). \end{array}$$

Having a basic syntax for functions, one wants to compute with them, which is done via the *β-rule*:

$$(\lambda x.M)N =_\beta M[x := N],$$

where $M[x := N]$ denotes the *substitution of N for x in M*. This is what one would expect, given the intuitive understanding of $\lambda x.M$. Some care has to be taken when substituting N for x: we have to make sure that a free variable in N doesn't become bound after the substitution: $(\lambda x.\lambda y.x)(zy) =_\beta \lambda y.zy$ is not correct. To solve this problem we first have to *rename the bound variable y* before substituting zy for x. The following is correct.

$$(\lambda x.\lambda y.x)(zy) = (\lambda x.\lambda v.x)(zy) =_\beta \lambda v.zy$$

The relation of *β-equality*, also denoted by $=_\beta$, is defined as the least equivalence relation on Λ that contains the β-rule and is compatible with abstraction and application. Put differently, the β-rule can be applied anywhere inside a λ-term. Computationally, we can give it a direction from left to right, and then it can be seen as a reduction relation. In this paper we will not go any deeper into the λ-calculus, the standard references being [Curry and Feys 1958, Barendregt 1984].

As examples of the β-rule, we can apply the combinators to some terms:

$$\begin{array}{rcl} \mathbf{K}MN & = & (\lambda x.\lambda y.x)MN \\ & =_\beta & (\lambda y.M)N \\ & =_\beta & M \end{array}$$

So, the combinator $\mathbf{K}$ can take two arguments and returns the first, while discarding the second.

$$\mathbf{SKK} \quad = \quad (\lambda x.\lambda y.\lambda z.xz(yz))\mathbf{KK}$$

[1] We will suppress some brackets inside terms. This is done in λ-calculus by giving application preference over abstraction and by letting the omitted brackets associate to the left in an application. So $\lambda x.xyx$ denotes $(\lambda x.((xy)x))$.

$$\begin{array}{ll} =_\beta & (\lambda y.\lambda z.\mathbf{K}\,z\,(y\,z))\,\mathbf{K} \\ =_\beta & \lambda z.\mathbf{K}\,z(\mathbf{K}\,z) \\ =_\beta & \lambda z.(\lambda x.\lambda y.x)\,z\,(\mathbf{K}\,z) \\ =_\beta & \lambda z.(\lambda y.z)\,(\mathbf{K}\,z) \\ =_\beta & \lambda z.z \\ = & \mathbf{I} \end{array}$$

So we see that $\mathbf{S}\,\mathbf{K}\,\mathbf{K} =_\beta \mathbf{I}$, showing that the combinator $\mathbf{I}$ can be defined in terms of $\mathbf{S}$ and $\mathbf{K}$.

4.2 Simple type theory and minimal implicational logic

The simplest way to give types to λ-terms is by using *simple types*. This originates from [Church 1940], but we give the presentation as it is usually done these days. The types are formed from type variables (the set $\mathscr{T}$) using only the arrow as a type constructor.

$$\text{Type} ::= \mathscr{T} \mid (\text{Type} \to \text{Type})$$

We use $\alpha, \beta, \gamma, \ldots$ to denote type variables and $A, B, C, \ldots$ to denote types. We suppress brackets by letting them associate to the right in a type, so $A \to B \to C$ denotes $A \to (B \to C)$. Now, we give types to (some) λ-terms by deriving statements of the form

$$\Gamma \vdash M : B$$

where M is a λ-term and B is a type. Here Γ is a *context*, that we need to give a type to the free variables in M. So Γ is of the form $x_1 : A_1, \ldots, x_n : A_n$, where all the x_i are distinct. The rules for deriving a judgment $\Gamma \vdash M : B$ are as follows.

$$\frac{}{\Gamma \vdash x : A}\ (\text{var}) \text{ if } x : A \in \Gamma \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ (\text{abs}) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\text{app})$$

Not all λ-terms can be typed in this system. For example, $\lambda x.xx$ cannot be typed. The combinators that we have defined above can be typed in the empty context: for any types A, B, C we have

$$\begin{array}{rcl} \mathbf{I} & : & A \to A, \\ \mathbf{K} & : & A \to B \to A, \\ \mathbf{S} & : & (A \to B \to C) \to (A \to B) \to A \to C. \end{array}$$

The types of $\mathbf{I}$, $\mathbf{K}$ and $\mathbf{S}$ correspond exactly with the axioms of minimal implicational logic, as was first pointed out by Curry (see [Howard 1980]). Minimal implicational logic is proposition logic with implication as the only connective and implication-elimination and implication-introduction as only logical rules. In case one presents the natural deduction rules with a sequent notation, the *propositions-as-types isomorphism* can be observed immediately. Here are the deduction rules of minimal implicational logic. Here Δ is a set of implicational formulas and A and B range over implicational formulas, which are the same as types from Type.

$$\frac{}{\Delta \vdash A}\ (\text{ax}) \qquad \frac{\Delta \cup \{A\} \vdash B}{\Delta \vdash A \to B}\ (\to\text{-in}) \qquad \frac{\Delta \vdash A \to B \qquad \Delta \vdash A}{\Delta \vdash B}\ (\to\text{-el})$$

The so called *Curry-Howard propositions-as-types isomorphism* ("formulae-as-types" in [Howard 1980]) maps a derivations Σ in natural deduction to a λ-term $\overline{\Sigma}$ such that the following holds.

$$\text{If } \Sigma \text{ is a derivation with conclusion } \Delta \vdash A, \text{ then } \overline{\Delta} \vdash \overline{\Sigma} A,$$

where $\overline{\Delta}$ is a context with $x : B \in \overline{\Delta}$ for some x in case $B \in \Delta$. So, the *assumptions* in Δ are translated to *variable declarations* and the use of these assumptions in the derivation Σ is mimicked by an occurrence of the associated free variable in $\overline{\Sigma}$.

There is also a mapping in the other direction, from typable λ-terms to derivations and these mappings together constitute an isomorphism between derivations and typed λ-terms. Often, the typed terms are presented with the type information explicitly given, which is called *typing à la Church* in [Barendregt 1992]. Then, instead of $\lambda x.M : A \to B$, one writes $\lambda x : A.M : A \to B$.

An important additional observation, due to [Tait 1965], is that the proof-theoretic concept of *cut-elimination* corresponds to the β-rule. In natural deduction, a *cut* is a part of a derivation where an introduction rule (say introducing $A \to B$) is immediately followed by an elimination rule (concluding B from A and the $A \to B$ that had just been introduced). It was observed by Tait that the removal of cuts corresponds to the β-rule $(\lambda x : A.M)N =_\beta M[x := N]$. This gives a computational interpretation to derivations in natural deduction. In [Howard 1980], the propositions-as-types isomorphism is extended to constructive predicate logic and Heyting arithmetic.

The propositions-as-types isomorphism can also bee seen as a concretization of the *Brouwer-Heyting-Kolmogorov interpretation* (BHK) of propositions (and proofs). Under this interpretation, a proposition A is not analyzed in terms of *truth* or *falsehood* of A, but in terms of what it means to be *a proof of* A. We can only establish that A holds by giving a proof of A, so we need to establish what it means to be a proof of A. In the BHK interpretation, a proof of $A \to B$ is a method to transform a proof of A into a proof of B. The proofs-as-terms interpretation makes the informal BHK idea formally precise by mapping a derivation to a λ-term, which indeed is a function. Martin-Löf has taken this as a starting point for his *intuitionistic type theory* which gives a foundation of mathematics based on intuitionistic principles [Martin-Löf 1984]. Martin-Löf treats propositions and sets really at the same level; there is no formal distinction between them. The collection of natural numbers forms a set ($\mathsf{nat} : \mathbf{Set}$), just like the collection of proofs of, say, $\forall x \in \mathbf{N}(P(x) \to P(x))$ (viz. $\Pi x : \mathsf{nat}.\, Px \to Px : \mathbf{Set}$).

4.3 De Bruijn's view on propositions-as-types and proofs-as-terms

De Bruijn reinvented the propositions-as-types concept, where he was somewhat influenced by Heyting. They were professors at the University of Amsterdam at the same time. De Bruijn writes in [De Bruijn 1994]:

> An important thing I got from Heyting is the interpretation of a proof of an implication $A \to B$ as a kind of mapping of proofs of A to proofs of B. Later this became one of the motives to treat proof classes as types.

It should be noted that de Bruijn did not view a proposition *itself* as a type, but, given a proposition A one can consider *the type of proofs of* A, which he denotes by $T(A)$. For de Bruijn, the key issue was not to interpret a proposition as a type but to interpret a *proof as a term*. According to him, interpreting a proof, which is an informal notion in mathematics, as a formal term is yet another example of the advancement of mathematics, where notions that are in the meta-language are transferred to the formal object language, to make them amenable for further mathematical research. Another example of this, according to de Bruijn, is the notion of 'function', which has long been a meta-theoretic notion, but was transferred to

the language with the introduction of λ-abstraction. De Bruijn regretted that Bourbaki hadn't used λ-notation as an explicit notation for functions, because he found it very fruitful [De Bruijn 1994]. The *proofs-as-terms* concept is what de Bruijn calls 'PAT'.

Nowhere in his writings, does de Bruijn refer to a computational interpretation of proofs, in the sense of Tait and Howard and later being promoted by Martin-Löf, where cut-elimination corresponds to β-reduction. For de Bruijn, λ-abstraction, application and β-reduction were needed for handling variable binding, instantiation and substitution as meta-operations of the framework.

5 Logical Framework and Dependent Types

Two very important contributions of de Bruijn are:

- The insight that one can define a *logical framework* that only encompasses basic mathematical operations, like abstraction, instantiation and substitution, in which a user can do his/her 'own' mathematics
- The introduction and use of so called *dependent types* as a vehicle to make this technically work.

The concept of a logical framework can be explained abstractly by contrasting the "direct proofs-as-terms (PAT) embedding" (due to Curry and Howard) with the "logical frameworks proofs-as-terms (PAT) embedding", which is due to de Bruijn. (These should not be confused with so called "deep" and "shallow" embeddings of logic in type theory, see e.g. [Garillot and Werner 2007][2]). In a *direct PAT embedding*, there is an embedding T of a logic L into a type theory "typetheory(L)", which maps a proposition to the type of its proofs. One has

$$\Gamma \vdash_L A \text{ iff } \overline{\Gamma} \vdash_{\text{typetheory}(L)} M : \mathsf{T}(A)$$

where

- typetheory(L) is the (fixed) type theory associated with the logic L; $\vdash_L$ denotes derivability in the logic L; $\vdash_{\text{typetheory}(L)}$ denotes derivability in typetheory(L).
- M codes, as a λ-term, the derivation of A.
- $\overline{\Gamma}$ contains
  - declarations $x : \sigma$ of free variables that occur in propositions,
  - assumptions, of the form $y : \mathsf{T}(B)$,
  - definitions of objects: $x := t : \sigma$,
  - proven lemmas, which are also definitions: $y := p : \mathsf{T}(B)$ (y is a name for the proof p of B).

The embedding T is often an isomorphism and is often even the identity, for example in the case of minimal implicational logic and simple type theory that we have discussed in Section 4.2. In that simple case, there are no first-order variables and we don't have definitions, but one could easily add those. We use a special font to denote T to emphasize that the embedding is outside the type theory. The (implicit) goal of the direct PAT embedding is to find a type theory typetheory(L) that is *isomorphic* to a logic L, and therefore this embedding is often referred to as a *PAT isomorphism*. A least requirement is that the

[2] In a deep embedding one encodes logic as a data type, so one can do meta-reasoning about the logic in type theory, whereas in a shallow embedding one uses the constructions of the type theory itself to represent logical mechanisms. In this respect both the direct and the logical frameworks PAT embedding are shallow.

map T constitutes a *conservative embedding*: for every formula A of the logic, if $\mathsf{T}(A)$ is *inhabited in* typetheory(L) (i.e. there is a closed term $M : \mathsf{T}(A)$), then A is provable in L.

De Bruijn's version of this is a *logical frameworks PAT embedding* in which we have the following.

$$\Gamma \vdash_L A \text{ iff } \Gamma_L, \overline{\Gamma} \vdash_{\mathrm{TT}} M : T\,A$$

where L is a logic, Γ_L is the *context in which the constructions of the logic L are declared* and $\overline{\Gamma}$ is as before. The main difference is that there is *one type theory* that acts as the logical framework in which a logic L can be encoded by choosing an appropriate context Γ_L.

We can make this precise for the case of minimal implicational logic, that we have seen in Section 4.2. The first observation is that simple type theory can be used to describe a term-language. (This was already known to Church [Church 1940], who used it to describe the language of higher order logic, but de Bruijn wasn't familiar with this work.) The language of implicational formulas can be described by the context

$$\begin{array}{rcll} \mathrm{prop} & : & \textbf{type} & (1) \\ \supset & : & \mathrm{prop} \to \mathrm{prop} \to \mathrm{prop} & (2) \end{array}$$

where **type** denotes the collection of types (the set Type in Section 4.2), which has now been 'internalized' inside the type theory, so we have as one of the rules

$$\frac{\Gamma \vdash A : \textbf{type} \qquad \Gamma \vdash B : \textbf{type}}{\Gamma \vdash A \to B : \textbf{type}}\,(\to)$$

Γ contains declarations of type variables: $\alpha \in \mathscr{T}$ of Section 4.2 is now internalized as $\alpha : \textbf{type} \in \Gamma$.

With this initial bit of context plus some declarations of variables of type prop, we can make propositions:

$$\begin{array}{lcl} \mathrm{prop} : \textbf{type}, \supset : \mathrm{prop} \to \mathrm{prop} \to \mathrm{prop}, \alpha : \mathrm{prop}, \beta : \mathrm{prop} & \vdash & \supset\,(\supset\,\alpha\,(\supset\,\beta\,\alpha)) : \mathrm{prop} \\ \mathrm{prop} : \textbf{type}, \supset : \mathrm{prop} \to \mathrm{prop} \to \mathrm{prop}, \alpha : \mathrm{prop}, \beta : \mathrm{prop} & \vdash & \supset\,(\supset\,(\supset\,\alpha\,\beta)\,\alpha)\,\alpha : \mathrm{prop}. \end{array}$$

These propositions are usually written infix: $\alpha \supset \beta \supset \alpha$ and $((\alpha \supset \beta) \supset \alpha) \supset \alpha$. We will also use this infix notation for readability. We observe that the *propositions-as-types* embedding is not the identity, but a function that encodes a proposition as a term of type prop (in an appropriate context).

The next step is to let a proof of a proposition A be a term of type A, so we have a real *proofs-as-terms* embedding. This cannot be achieved now, because an $A : \mathrm{prop}$ is a term and not a type. The type prop is best viewed as the type of *names of propositions*, and not as the type of propositions, because an $A : \mathrm{prop}$ cannot yet be inhabited. So, de Bruijn introduces a map $T : \mathrm{prop} \to \textbf{type}$ that transforms names of propositions into the types of their proofs. In the type theory, this is achieved by allowing $\mathrm{prop} \to \textbf{type}$ as a type[3] and declaring $T : \mathrm{prop} \to \textbf{type}$ in the context Γ_L:

$$T \quad : \quad \mathrm{prop} \to \textbf{type} \qquad (3)$$

The simple addition of $T : \mathrm{prop} \to \textbf{type}$ paves the way for *dependent types*. A dependent type is a type of the form $\Pi x : A.B$, where x can occur as a free variable in B. This type is a generalization of the function

[3] One can put $\mathrm{prop} \to \textbf{type} : \textbf{type}$, but also put another "sort" (or "universe"), **kind** and put $\mathrm{prop} \to \textbf{type} : \textbf{kind}$; the relevant thing is that $\mathrm{prop} \to \textbf{type}$ can be inhabited.

type $A \to B$, where the "range type" B can depend on the input $a : A$. Informally, $\Pi x : A.B$ denotes the type of terms f such that, for all $a : A$, one has $f\,a : B[x := a]$. With $T : \text{prop} \to \mathbf{type}$ we can now form the types $\Pi x : \text{prop}.T\,x \to T\,x$ and $\Pi x, y : \text{prop}.T\,x \to T\,y \to T\,x$, by the following generalization of the type formation rule for $\to$ to Π:

$$\frac{\Gamma \vdash A : \mathbf{type} \qquad \Gamma, x : A \vdash B : \mathbf{type}}{\Gamma \vdash \Pi x : A.B : \mathbf{type}}\ (\Pi)$$

NB. $\Pi x, y : \text{prop}.T\,x \to T\,y \to T\,x$ above abbreviates $\Pi x : \text{prop}.\Pi y : \text{prop}.T\,x \to (T\,y \to T\,x)$.

Together with the Π-types, one also needs adapted rules for abstraction and application:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Pi x : A.B}\ (\text{abs}) \qquad \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}\ (\text{app})$$

Given the interpretation of $T\,\varphi$ as the type of proofs of proposition φ (for $\varphi : \text{prop}$), $T\,\varphi \to T\,\varphi$ seems to represent the type of proofs of $\varphi \supset \varphi$, but it does not: the type of proofs of $\varphi \supset \varphi$ is $T(\varphi \supset \varphi)$. To represent proofs, the rules of the logic need to be declared in Γ_L as well. In our simple example we just have an introduction and elimination rule for the implication. So we add the following to Γ_L:

$$\supset\text{–in} \quad : \quad \Pi x, y : \text{prop}.(T\,x \to T\,y) \to T(x \supset y) \tag{4}$$

$$\supset\text{–el} \quad : \quad \Pi x, y : \text{prop}.T(x \supset y) \to T\,x \to T\,y \tag{5}$$

These declarations are enough to encode proofs in minimal proposition logic as lambda terms. Now one can form terms that encode proofs of $\alpha \supset \alpha$ and of $\alpha \supset \beta \supset \alpha$ as follows:

$$\begin{array}{rcll} \Gamma_L, \alpha : \text{prop} & \vdash & \supset\text{–in}\,\alpha\,\alpha\,(\lambda \text{x} : \text{T}\,\alpha.\text{x}) & : \quad T(\alpha \supset \alpha) \\ \Gamma_L, \alpha, \beta : \text{prop} & \vdash & \supset\text{–in}\,\alpha\,(\beta \supset \alpha)\,(\lambda \text{x} : \text{T}\,\alpha.\supset\text{–in}\,\beta\,\alpha\,(\lambda \text{y} : \text{T}\,\beta.\text{x}) & : \quad T(\alpha \supset \beta \supset \alpha) \end{array}$$

The important idea to get across here is that the type theory acts as a logical framework, in which one encodes ones favorite logic – or any formal system, as long as it has an inductively defined language and notion of derivability – by choosing an appropriate "start context" Γ_L. As an example, the context Γ_L can be extended to predicate logic over a base domain A, say with a constant a and a unary function f, by adding a universal quantifier with introduction and elimination rules as follows.

$$A \quad : \quad \mathbf{type} \tag{6}$$

$$a \quad : \quad A \tag{7}$$

$$f \quad : \quad A \to A \tag{8}$$

$$R \quad : \quad A \to A \to \text{prop} \tag{9}$$

$$\forall \quad : \quad (A \to \text{prop}) \to \text{prop} \tag{10}$$

$$\forall\text{–in} \quad : \quad \Pi P : A \to \text{prop}.(\Pi x : A.T\,(P x)) \to T(\forall P) \tag{11}$$

$$\forall\text{–el} \quad : \quad \Pi P : A \to \text{prop}.T(\forall P) \to \Pi x : A.T\,(P x) \tag{12}$$

A proof of a statement in predicate logic, e.g. $(\forall x, y.R x y) \to \forall x.R x (f x)$, by giving a term of the associated type, $T(\forall(\lambda x : A.\forall(\lambda y : A.R x y)) \supset \forall(\lambda x : A.R x (f x)))$, in the above context.

Proof assistants like Coq and NuPrl are based on the direct PAT embedding, but also the idea of building a proof assistant on the logical framework concept has been used a lot. After de Bruijn, this was first made popular by the Edinburgh Logical Framework (ELF) [Harper, Honsell and Plotkin 1987],

which is heavily based on ideas from Automath, notably the PAT embedding that we have just described. (As a matter of fact, the presentation and notation used above is close to ELF.) The system Twelf [Pfenning and Schürmann 1999] is a direct descendant of ELF. Also Martin-Löf type theory [Martin-Löf 1984] and the Extended Calculus of Constructions [Luo 1994], which were originally defined via the direct PAT embedding, are now defined as formal logics *inside* a logical framework.

6 Implementing a proof checker

De Bruijn's team was the first to implement an actual proof checker and to formalize a non-trivial piece of mathematics (Landau's "Grundlagen der Analysis" [Landau 1960]) inside it [Van Benthem Jutting 1977]. This can be seen as a completion of the work started by Russell and Whitehead, who formalized mathematics in Russell's ramified theory of types, on paper. The advent of computer technology now made it possible to write formalized mathematics as computer code and have it checked for correctness by a computer program, the type checker.

A major question is why a proof checked by a computer should be correct. A computer program can be faulty as well, so why should one rely on a proof checked by a computer more than on a proof checked by humans? De Bruijn had already thought about this and he pointed out that the logical framework and its type checking algorithm should be simple. In Automath, the correctness of proofs and of mathematics is not just left to the computer, but the author writes "proof terms" that are checked by a simple program. This concept was therefore coined the "De Bruijn criterion" by Barendregt [Barendregt and Geuvers 2001]: a proof assistant satisfies the De Bruijn criterion if it generates proof terms that can be checked independently from the system by a simple program that a skeptical user could write him/herself. The idea is that a powerful system may have all kinds of intelligence built in, but in the end it should create a full proof (term) in a simple formal logic that can be checked by a simple program.

De Bruijn emphasized the simplicity of the framework: "Properly speaking, the rules of Automath involve little more than the art of substitution." [De Bruijn 1968]. Therefore, he devoted a lot of attention to the precise definitions of variable binding, instantiation and substitution. This has led to various contributions that go far beyond Automath or proof assistants, and have had impact on the implementation and study of formal systems and programming languages in general.

6.1 Nameless Dummies, Explicit Substitution and Definitions

An important issue is how to deal with variable binding in an implementation. Informally, it is easy to say that one works "modulo renaming of bound variables", which means that one is not really considering terms, but *equivalence classes of terms*, where the equivalence relation is $=_\alpha$. For example $\lambda x.(\lambda y.xy) =_\alpha \lambda y.(\lambda x.yx)$. In [De Bruijn 1972], de Bruijn invented a way to have a unique representative for each equivalence class, by treating bound variables as so called "nameless dummies", now usually referred to as *De Bruijn indices*. A bound variable is represented by a natural number and its "binding lambda" is found by counting the number of λ 's under whose scope the number is. For example, the term $\lambda z.(\lambda x.\lambda y.xyz)\,z$ is represented by $\lambda\,(\lambda\,\lambda\,2\,1\,3)\,1$. The 2 is bound by the middle λ and the 3 and the rightmost 1 are bound by the left-most λ. This is hard to read for a human, also because the two occurrences of z correspond to a different number (3 and 1), but it is purely combinatorial and easy to program with.

Another operation that needs further analysis in presence of variable binding is substitution. When writing $M[x := N]$, one cannot just replace x by N in M everywhere, because free variables in N may

get bound by this process. The nameless dummies, combined with *explicit substitution* also provide a combinatorial solution here, as de Bruijn first showed in [De Bruijn 1978]. In the example term above, $\lambda z.(\lambda x.\lambda y.xyz)z$, we see a β-redex and this term can be reduced to $\lambda z.(\lambda y.xyz)[x := z]$, which is just $\lambda z.\lambda y.zyz$. The idea of explicit substitution is to make the substitution an operator which moves through the term in small steps, so with named variables:

$$\begin{array}{rcl}
\lambda z.(\lambda x.\lambda y.xyz)z & \longrightarrow_\beta & \lambda z.(\lambda y.xyz)[x := z] \\
& \longrightarrow_x & \lambda z.\lambda y.(xyz)[x := z] \\
& \longrightarrow_x & \lambda z.\lambda y.(xy)[x := z]\,z[x := z] \\
& \longrightarrow_x & \lambda z.\lambda y.x[x := z]\,y[x := z]\,z[x := z] \\
& \longrightarrow_x & \lambda z.\lambda y.zyz.
\end{array}$$

where $\longrightarrow_\beta$ denotes the contraction of the β-redex and $\longrightarrow_x$ denotes the "explicit substitution reductions". In this simple example, we need no renaming of bound variables. With nameless dummies, one needs an update of the numbers, every time one goes under a λ, and the contraction of a β-redex also requires the "high numbers" in the body to be decreased by 1. If we do the same reduction with nameless dummies and explicit substitutions, and we use S_n^t to denote the operator substituting t for n, we get

$$\begin{array}{rcl}
\lambda\,(\lambda\,\lambda\,2\,1\,3)\,1 & \longrightarrow_\beta & \lambda\,S_1^1(\lambda\,2\,1\,3) \\
& \longrightarrow_x & \lambda\,\lambda\,S_2^2(2\,1\,3) \\
& \longrightarrow_x & \lambda\,\lambda\,S_2^2(2\,1)\,S_2^2(3) \\
& \longrightarrow_x & \lambda\,\lambda\,S_2^2(2)\,S_2^2(1)\,S_2^2(3) \\
& \longrightarrow_x & \lambda\,\lambda\,2\,1\,2
\end{array}$$

where the replacement of $S_2^2(3)$ by 2 is an "index update step": all numbers higher than 2 need to be decreased to point to the right λ, because they are in scope of one λ less.

A third aspect of the formal system that de Bruijn emphasized were definitions. He recognized that defining is a major mathematical activity, so definitions were an explicit mechanism in the Automath systems, including parametrized definitions. De Bruijn and Nederpelt [Nederpelt 1973] also observed that definitions behave very much like a β-redex (or, as de Bruijn would say, an "abstraction-application-pair"): the definition $c := P$ applied to M can be seen as $(\lambda c.M)P$. The contraction of the β-redex will just replace c by its definiendum P. An important difference is that in case of a definition, one often wants to unfold only one occurrence of c. Therefore, de Bruijn and Nederpelt have also studied various notions of "generalized β-reduction", that allow one to replace one occurrence of c by P, while maintaining the redex. This looks something like $(\lambda c.M[c])P \longrightarrow_\beta (\lambda c.M[P])P$, where $M[c]$ marks one position of c in M.

The ideas of using nameless dummies has been adopted by all proof assistants to implement variable binding. On the theoretical side, [Abadi et al. 1991] have been very influential in reviving the idea of nameless dummies (De Bruijn indices) and explicit substitutions and studying variants of these systems for various applications. [Lescanne and Rouyer-Degli 1995] gives a nice overview of the original ideas of de Bruijn on nameless dummies and explicit substitution in a present day formulation.

7 Some more comments and further reading

In our description of the contributions of de Bruijn to type theory and proof assistants, we have mostly used modern presentations and syntax. This doesn't really do justice to de Bruijn's pioneering ideas,

which were also revolutionary in introducing new notation. However, these ideas have not always caught up and using them in this paper would be a hindrance.

There is one specific feature of Automath, that we *do* want to point out and that is the focus on a linear format of the formal system. The typing rules that we have introduced thus far all produce tree like derivations of typing judgments, which is ok on the theoretical level, but impossible to deal with if one views the development of typing as an incremental process, like writing a mathematical book. The idea of Automath is not so much to derive judgments of the form

$$\Gamma \vdash M : A,$$

but to incrementally develop the context Γ, as our corpus of mathematics, consisting of

- definitions of objects $c := M : A$, where A : **type**,
- proofs of lemmas $c := M : T\,A$, where A : prop, which are also just definitions.
- variable declarations $x : A$, where A : **type**,
- axioms $x : T\,A$, where A : prop,

where of course, the axioms should be limited to the beginning of the development. In practice, Automath works with so called "lines", that together form a "book". Each line explicitly refers to an earlier line that it builds on. This mechanism also models the branching of the knowledge in a book. We refer to [De Vrijer 2013, Nederpelt et al. 1994] for details.

For further reading, there is a lot of material on Automath available, for example in the Automath archive `http://www.win.tue.nl/automath/` and in the overview volume [Nederpelt et al. 1994]. Both [Dechesne and Nederpelt 2012] and [De Vrijer 2013] give nice inside information on the history of Automath, by two prominent members of the Automath team that have witnessed its development very close by. The forthcoming book [Nederpelt and Geuvers 2014] intends to introduce the ideas of Automath in a modern setting, showing the use of type theory for formalizing mathematics to relative newcomers (students). For an extended overview of present day proof assistants and the ideas behind them, see [Geuvers 2009]. Finally, there is a working version of Automath, by [Wiedijk 2002] who has made a new implementation of it.

References

[Abadi et al. 1991] M. Abadi, L. Cardelli, P.-L. Curien and J.-J. Lévy, Explicit substitutions, *Journal of Functional Programming*, 1(4), pp. 375-416, 1991.

[Agda] Agda: An interactive proof editor. `wiki.portal.chalmers.se/agda`

[Barendregt 1984] H.P. Barendregt, *The lambda calculus, its syntax and semantics*, North-Holland 1984.

[Barendregt 1992] H.P. Barendregt, Lambda calculi with types, in *Handbook of Logic in Computer Science*, Oxford University Press, 1992.

[Barendregt and Geuvers 2001] H. Barendregt and H. Geuvers, Proof Assistants using Dependent Type Systems, in *Handbook of Automated Reasoning (Vol 2)*, eds. A. Robinson, A. Voronkov, Elsevier 2001, pp. 1149-1238 (Chapter 18).

[Van Benthem Jutting 1977] L.S. van Benthem Jutting, Checking Landau's "Grundlagen" in the AUTOMATH system, parts D2, D3, D5 of [Nederpelt et al. 1994].

[Boyer and Moore 1998] R.S. Boyer and J.S. Moore, *A Computational Logic Handbook*, Second Edition, Academic Press, 1998,

[De Bruijn 1968] N.G. de Bruijn, Automath, a language for mathematics, Department of Mathematics, Eindhoven University of Technology, TH-report 68-WSK-05, 1968. Reprinted in revised form, with two pages commentary, in: *Automation and Reasoning, vol 2, Classical papers on computational logic 1967-1970*, Springer Verlag, 1983, pp. 159-200.

[De Bruijn 1972] N.G. de Bruijn, Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem, *Indagationes Mathematicae*, 34(5), 1972, 381-392.

[De Bruijn 1978] N.G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. *Proc. of the Koninklijke Nederlands Akademie*, 81(3), pp. 1-9, September 1978.

[De Bruijn 1991] N.G. de Bruijn, A plea for weaker frameworks, in *Logical Frameworks*, eds. G. Huet and G. Plotkin, Cambridge University Press, 1991, pp. 40-67.

[De Bruijn 1998] N.G. de Bruijn, Type-theoretical checking and philosophy of mathematics, In *Twenty-Five Years of Constructive Type Theory*, Proceedings of a Congress held in Venice, Italy, October 1995, eds. G. Sambin and J. Smith, Oxford University Press, 1998, pp. 41-56.

[De Bruijn 1994] N.G. de Bruijn, Reflectins on Automath, in [Nederpelt et al. 1994].

[Church 1940] A. Church, A Formulation of the Simple Theory of Types, *JSL* 5, 1940.

[Constable et al 1986] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki and S.F. Smith, *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, NJ, 1986.

[Coq] The Coq proof assistant, `http://coq.inria.fr/`

[Curry and Feys 1958] H.B. Curry and R. Feys, *Combinatory Logic Vol. I*, Amsterdam, North-Holland, 1958.

[Dechesne and Nederpelt 2012] F. Dechesne and R.P. Nederpelt, N.G. de Bruijn (1918-2012) and his road to Automath, the earliest proof checker. *The Mathematical Intelligencer*, 34(4), pp. 4-11, 2012.

[Garillot and Werner 2007] F. Garillot and B. Werner Simple types in type theory: Deep and shallow encodings. In *Theorem Proving in Higher Order Logics 2007*, Lecture Notes in Computer Science 4732, Springer, pp. 368-382, 2007.

[Geuvers 2009] H. Geuvers, Proof Assistants: history, ideas and future, *Sadhana Journal, Academy Proceedings in Engineering Sciences, Special Issue on Interactive Theorem Proving and Proof Checking*, Indian Academy of Sciences, Vol 34, part 1, February 2009, pp 3-25.

[Girard 1972] Jean-Yves Girard, (in French), *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur* (Ph.D. thesis), Université Paris 7, 1972.

[Girard et al. 1989] J.-Y. Girard, Y. Lafont and P. Taylor, *Proofs and Types*, Cambridge University Press, 1989

[Gordon 2006] M.J.C. Gordon, From LCF to HOL: a short history, *Proof, language, and interaction: essays in honour of Robin Milner* MIT Press Cambridge, MA, USA, pages 169–185, 2000.

[Gordon & Melham 1993] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gordon et al 1979] M.J.C. Gordon, R. Milner and C.P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of Lecture Notes in Computer Science. Springer, 1979.

[Harper, Honsell and Plotkin 1987] R. Harper, F. Honsell and G. Plotkin, A framework for defining logics, *Proceedings of the second Symposium on Logic in Computer Science*, Ithaca, NY, IEEE, 1987.

[Harrison 2009] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press 2009.

[HOL-light] The HOL Light theorem prover `http://www.cl.cam.ac.uk/ jrh13/hol-light/`

[Howard 1980] W.A. Howard, The formulae-as-types notion of construction, in J. Seldin and R. Hindley (eds.), *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Boston, MA, Academic Press, pp. 479-490, 1980 (original manuscript from 1969).

[Kaufmann, Manolios and Moore 2000] M. Kaufmann, P. Manolios and J.S. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June, 2000.

[Lescanne and Rouyer-Degli 1995] P. Lescanne and J. Rouyer-Degli, Explicit Substitutions with de Bruijn's Levels, in *Rewriting Techniques and Applications, 6th International Conference*, Lecture Notes in Computer Science 914, Springer, pp. 294-308, 1995.

[Luo 1994] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science, 11. Oxford University Press, 1994.

[Lyaletski et al 2004] A. Lyaletski, A. Paskevich and K. Verchinin, Theorem Proving and Proof Verification in the System SAD. In *Mathematical Knowledge Management, Third International Conference*, Białowieza, Poland, Proceedings, LNCS 3119, pp. 236-250, 2004.

[Martin-Löf 1984] P. Martin-Löf, *Intuitionistic type theory*, Napoli, Bibliopolis, 1984.

[McCarthy 1962] J. McCarthy, Computer programs for checking mathematical proofs, In *Recursive Function Theory*, Proceedings of the Symposia in Pure Mathematics, volume 5, pages 219-228, Americal Mathematical Society, 1962.

[McCarthy 1960] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, *Communications of the ACM*, Vol 3 Issue 4, April 1960, pp. 184-195. ACM New York, NY, USA

[Mizar] Mizar Home Page, http://www.mizar.org/

[Nederpelt 1973] R.P. Nederpelt, Strong normalization in a typed lambda calculus with lambda structured types, PhD. thesis, Eindhoven University of Technology, the Netherlands, 1973.

[Nederpelt et al. 1994] R.P. Nederpelt, H. Geuvers, R.C. de Vrijer, (editors), *Selected Papers on Automath*, Studies in Logic and the Foundations of Mathematics Volume 133, North-Holland, Amsterdam, 1994.

[Nederpelt and Geuvers 2014] R.P. Nederpelt, H. Geuvers, *Type Theory and Formal Proof*, to appear, CUP.

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

[Pfenning and Schürmann 1999] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction*, (CADE-16), H. Ganzinger, editor, LNAI 1632, Springer, pp. 202-206.

[Scott 1993] D. Scott, A type-theoretical alternative to ISWIM, CUCH, OWHY. *TCS*, 121:411440, 1993. (Annotated version of a 1969 manuscript.)

[Scott 1970] D. Scott, Constructive validity, *Symposium on Automatic Demonstration* (Versailles, 1968), Lecture Notes in Mathematics, Vol. 125, Springer, Berlin, pp. 237-275, 1970.

[Tait 1965] W.W. Tait, Infinitely Long Terms of Transfinite Type, *Studies in Logic and the Foundations of Mathematics*, Volume 40, 1965, Pages 176-185.

[Tait 1967] W.W. Tait, Intensional interpretation of functionals of finite type I, *Journal of Symbolic Logic*, vol. 32 (1967), no. 2, pp. 198–212.

[Verchinine et al 2008] K. Verchinine, A. Lyaletski, A. Paskevich and A. Anisimov, On correctness of mathematical texts from a logical and practical point of view. In *Intelligent Computer Mathematics, AISC/Calculemus/MKM 2008*, Birmingham, UK, LNCS 5144, Springer, pp. 583–598, 2008.

[De Vrijer 2013] R. de Vrijer, Wiskunde als taal: het Automath-project (in Dutch) *Nieuw Archief voor Wiskunde* (NAW) 5/14, pp. 36-41, 2013.

[Landau 1960] E. Landau, *Grundlagen der Analysis*, Chelsea Publ. Comp., 3rd edition, 1960.

[Wiedijk 2002] F. Wiedijk, A new implementation of Automath, .*J Autom. Reasoning*, 29(3-4), 2002, pp. 365-387.

[Wenzel et al 2008] M. Wenzel, L.C. Paulson and T. Nipkow. The Isabelle Framework. In O. Ait-Mohamed, editor, *Theorem Proving in Higher Order Logics*, TPHOLs 2008, invited paper, LNCS 5170, Springer, 2008.