Manual of the TYPO Type Checker

J.M.G.G. de Nivelle

Information Systems (INS)

INS-R9804 February 28, 1998

Report INS-R9804
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Manual of the TYPO Type Checker

H. de Nivelle
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
TYPO is a type checker which implements typed lambda calculus with the purpose of formalizing mathematical proofs. TYPO has been constructed with the following design objectives: **(1)** Easy installation. In order to obtain this goal the program has been written in portable C. The program has modest memory demands. **(2)** Access to the proof terms. A formalized theory is represented as an ASCII file containing the definitions/theorems and proofs as lambda terms. **(3)** Maintainability. In order to obtain this goal the program has not been optimized for speed.

1991 Mathematics Subject Classification: 03B15, 03B40, 16B70
1991 Computing Reviews Classification System: F.4.1
Keywords and Phrases: Implementation of Type Theory, Formal Verification
Note: Supported by NWO Grant 612-32-007

1. Introduction

The term type theory is used for a family of logical calculi, that have been designed with the purpose of verifying mathematical proofs. Type theory originates in the AUTOMATH project, which took place in Eindhoven in the 70's. The AUTOMATH project was very successful from the practical point of view. The complete work of ([Land30]) was verified, and a large part of analysis. Also theoretically it has been very succesfull, as it inspired a whole generation of researchers ([ML75], [ML84], [CQ85], [Luo89], [Geuv93])

Type theory can be seen as a logic with few rules, but these rules are strong. This makes it possible to define other logics in type theory. Because of this type theory is sometimes called a *logical framework*, or a *meta logic*. This makes that when a mathematical theory is verified, the logical axioms that are used have to be stated, in the same manner as the other axioms, and this makes them equally transparant.

It is well-known that mathematics can also be formalized in set theory, for example ZF set theory. For the AUTOMATH project it was chosen not to use ZF, because the principles of set theory are too strong, and too dubious. With type theory one can choose the principles that are used, and allow them to be weak if they can be weak.

Another reason why type theory was chosen was the fact that it is closer to the mathematical practice. The designers of the AUTOMATH system did not perceive type theory as a logic ([NGV94]), and they believed that mathematical reasoning was distinct from logical reasoning. This motivation is not valid anymore, as type theory has been greedily absorbed by the logic community.

However something remains of the argument since the strength of the system makes it possible to derive theorems about the logic that is used, on an equal level with theorems about the field that is studied. After that the logical theorems can be used on the same level as the logical axioms. This shifting between object level and metalevel is closer to what is done in mathematical text books, than the strict disctinction that is enforced by classical logic with set theory.

1.1 An Informal Description of the Calculus

The calculus of type theory is based on *typed λ-calculus.* λ-calculus can be seen as a formalism that is able to describe computation. The types in typed λ-calculus are not so much different from the types in programming languages. They constrain the programs that can be formed, so that functions can only be applied on meaningful arguments. The types in λ-calculus have as a side-effect that they make all computations terminating.
The objects that the λ-calculus knows off, are λ-terms. They are used to represent objects and programs. Later they will also be used to represent proofs, and statements.

Definition 1.1 We assume an infinite set of variables V. The set of λ-terms is recursively defined as follows:

- Every variable is also a λ-term.
- If x is a variable, X is a type, and a is a λ-term, that possibly contains x, then $\lambda x{:}X a$ is a λ-term. The intended meaning is a function dependent on x.
- If f is a λ-term, and t is a λ-term, then $f \cdot t$ is a λ-term. The intended meaning is the application of function f on argument t.
- If x is a variable, and X is a type, and A is a type, possibly depending on x, then $\Pi x{:}X A$ is a type. The intended meaning is a variable type, dependent on x. If x does not occur in X, then it is possible to write $X \rightarrow A$.

We will write just term instead of λ-term. We write $x{:}X$ for the statement

- Term x has type X.

We write $a_1{:}A_1, \ldots, a_n{:}A_n \vdash b{:}B$, for the statement

- If each a_i has type A_i, then b has type B.

We have been unclear at some points. We did not specify what are the types, and we did not use the types at all in forbidding ill-typed terms. We want first to explain the ideas, and later to explain the details.

Example 1.2 We have $3{:}\mathrm{Nat}$, $0{:}\mathrm{Nat}$, and $s{:}\mathrm{Nat} \rightarrow \mathrm{Nat}$. One can write

$$x{:}\mathrm{Nat} \vdash (s \cdot x){:}\mathrm{Nat}.$$

$$x{:}\mathrm{Nat} \vdash s \cdot (s \cdot x){:}\mathrm{Nat}.$$

Then

$$\vdash \lambda x{:}\mathrm{Nat}\ \ (s \cdot (s \cdot x)){:}\mathrm{Nat} \rightarrow \mathrm{Nat}.$$

One can write $\sin \cdot x$ for the sine of x. (The application of the sine function on x.) Similarly x times y can be written as $* \cdot x \cdot y$. This last term is equal to $(* \cdot x) \cdot y$. The subterm $(* \cdot x)$ corresponds to the function that takes a number and multiplies it with x.
The type $\mathrm{Nat} \rightarrow \mathrm{Nat}$ denotes the type of functions from the natural numbers to the natural numbers. If $R \cdot n$ denotes the vector space R^n, then $R{:}\mathrm{Nat} \rightarrow \mathrm{Type}$. If $Z \cdot n$ denotes the zero vector of dimension n, then the function Z has type $\Pi n{:}\mathrm{Nat}(R \cdot n)$. One could use $+ \cdot n$ to denote vector addition in an n dimensional space. Then

$$\vdash +{:}\,\Pi n{:}\mathrm{Nat}(R \cdot n \rightarrow R \cdot n \rightarrow R \cdot n).$$

Example 1.3 Analysis could benefit quite a lot from the notation of λ-calculus.([NGV94]) In analysis there is no good manner to write functions. Usually functions are written as formulae, and there is a convention that letters x, y are intended to denote parameters. For example the derivative of ax^n equals anx^{n-1}, because a function in x is intended. However if one wants to differentiate xy^z the situation is not so clear. In order to overcome this ambiguity the derivative could seen as a binary operator. The first argument is a formula, and the second is a variable. So xy^z can be differentiated either after x, y, or z.
One can write $D(t, x)$ for the derivative of term t after x. An alternative notation is:

$$\frac{\partial t}{\partial x}.$$

However what does the x mean?
Even worse is that there is no good notation for the derivative of the sine function, as there is no variable in the sine-function.
What the derivative operator really is, is an operator with type (Real $\to$ Real) $\to$ (Real $\to$ Real).
In order to denote which expressions in xy^z are intended as constants one should write $\lambda y{:}\text{Real}\ xy^z$. This term has type Real $\to$ Real and it can be an argument to the differential operator.

There are many more such expressions in mathematics:

Example 1.4
- The following expressions use a variable, where in fact they are operators on functions:

$$\int_a^b x^2 dx. \qquad \Sigma_{x=a}^b x^2 \qquad \lim_{x\to 0} \frac{\sin x}{x} \qquad \min_x f(x).$$

  Using the λ-notation these expressions could be written as:

$$\int_a^b \lambda x{:}\text{Real}\ x^2 \qquad \Sigma_a^b \lambda x{:}\text{Nat}\ x^2 \qquad \lim(0, \lambda x{:}\text{Real}\ \frac{\sin x}{x}) \qquad \min(\lambda x{:}\text{Real}\ f(x)).$$

- When the induction principle $\forall P[P(0) \to \forall x(P(x) \to P(s(x))) \to \forall y(P(y))]$. is applied on $x + y = y + x$, using the variable x. The induction hypothesis is applied on the predicate $\lambda x{:}\text{Nat}(x + y = y + x)$.

We did not describe how the λ-calculus can model computation. There are two basic mechanisms for this:

1. Expansion of definitions: If a variable f has been defined as a term t, then f can be replaced by its definition. This corresponds to replacing names of functions by their defininition during the evaluation of a functional program.
2. Expansion of λ-terms: The term $(\lambda x{:}X a) \cdot t$ can be replaced by $a[x := t]$, where $a[x := t]$ is the result of replacing x by t everywhere in a. This corresponds to evaluating the term a in the context $x := t$, during the evaluation of a functional program.

In the next section we show how proofs can be embedded in the calculus.

1.2 The Curry/Howard/De Bruyn isomorphism

The CHdB isomorphism consists of the following identifications:

Formula = Type,

Proof = Term.

Using these equations formulae and their proofs can be introduced into the typed λ-calculus. The statement $a{:}A$ can be interpreted as a is a proof of A, using these equations.

1. Formulae of the form $A \rightarrow B$, (A implies B) are encoded as the type $A \rightarrow B$.
2. Formulae of the form $\forall x{:}XP$ are encoded as the type $\Pi x{:}XP$.

This embedding is meaningful because the standard rules for type derivation correspond to the rules for natural deduction. We do not prove this, but we give a set of examples that illustrates all of the rules of natural deduction:

1. If there is a term a with $a{:}A$ and there is a term f with $f{:}A \rightarrow B$, then $(f \cdot t){:}B$ This corresponds to $\rightarrow$-elimination.
2. If assuming that $a{:}A$ one can find a term f with type B, then the term $\lambda a{:}Af$ has type $A \rightarrow B$. This corresponds to $\rightarrow$-introduction.
3. If there is a term f with $f{:}\Pi x{:}XA$, and $a{:}X$, then the term $(f \cdot a){:}(X[x := a])$. This corresponds to $\forall$-elimination.
4. If assuming that $x{:}X$ one can find a term $a[x]$ of type $A[x]$, then the term $\lambda x{:}Xa[x]$ has type $\Pi x{:}Xa[X]$. This corresponds to $\forall$-introduction.

So the logic that is embedded in λ-calculus has only implication and universal quantification. Nevertheless it is very strong, because it allows second order quantification, and with second order quantification the other operators can be expressed.

1.3 Formal Description of the Calculus

In this section we will formally describe the typing rules, as they are used by the TYPO system. We begin by repeating the definition of a term. At this point there is no type checking yet.

Definition 1.5 We assume a fixed set of variables V. The set of terms is finitely generated by the following rules:

- A variable v is term.
- If f and g are terms, then both $f \cdot g$ and $f \Rightarrow g$ are terms. The intended meaning of $f \Rightarrow g$ is f *rewrites* to g.
- If X and A are terms, and if x is a variable, then both $\lambda x{:}XA$ and $\Pi x{:}XA$ are terms.

A *context* is a finite list $\Gamma_1, \dots, \Gamma_n$, where each Γ_i has one of the following three forms:

1. Either Γ_i is of the form $y{:}Y$, in which y is a variable. In that case Γ_i is called a *declaration.*
2. Or Γ_i is of the form $x := y{:}Y$, with x a variable. In that case Γ_i is called a *definition.*
3. Γ_i can also be a *rewrite rule* definition $x{:}\lambda x_1{:}X_1 \cdots \lambda x_n{:}X_n(a \Rightarrow b)$, with x a variable. In fact the only place where $\Rightarrow$ is allowed to occur is in the rewrite rule definitions.

Definition 1.6 We define when a variable v is *free* in term t.

- If $v = t$, then v is free in t.
- If t has form $f \cdot g$ then v is free in t, if either v is free in f, or v is free in g.
- If t has form $\lambda x{:}Xa$ or $\Pi x{:}XA$, then v is free in t if either
  1. v is free in X, or
  2. $v \neq x$, and v is free in a. (A in the case of Π)

Definition 1.7 We define recursively when two terms t and u are *α-equal*, notation $u \equiv_\alpha v$:

- $t \equiv_\alpha u$ if $t[\,] \equiv_\alpha u[\,]$.
- $t[v_1, \ldots, v_n] \equiv_\alpha u[w_1, \ldots, w_n]$ if both t and u are variables, and the following is the case: Let i be the largest integer for which $v_i = t$, or -1 if there exists no such v_i. Let j be the largest integer for which $w_j = u$, or -1 if there exists no such w_j. Then it must be the case that $i = j$, and if $i = j = -1$, then $t = u$.
- $(t{\cdot}a)[v_1, \ldots, v_n] \equiv_\alpha (u{\cdot}b)[w_1, \ldots, w_n]$ if $t[v_1, \ldots, v_n] \equiv_\alpha u[w_1, \ldots, w_n]$ and $a[v_1, \ldots, v_n] \equiv_\alpha b[w_1, \ldots, w_n]$.
- $(\lambda a{:}Ax)[v_1, \ldots, v_n] \equiv_\alpha (\lambda b{:}By)[w_1, \ldots, w_n]$ if $A[v_1, \ldots, v_n] \equiv_\alpha B[w_1, \ldots, w_n]$ and $x[v_1, \ldots, v_n, a] \equiv_\alpha y[w_1, \ldots, w_n, b]$.
- $(\Pi a{:}AX)[v_1, \ldots, v_n] \equiv_\alpha (\Pi b{:}BY)[w_1, \ldots, w_n]$ if $A[v_1, \ldots, v_n] \equiv_\alpha B[w_1, \ldots, w_n]$ and $X[v_1, \ldots, v_n, a] \equiv_\alpha Y[w_1, \ldots, w_n, b]$.

The following notion of normality avoids a lot of technical problems with bound variables.

Definition 1.8 Let Γ be a context. Γ is *normal* if all variables occurring on the left hand sides in the Γ_i are distinct.

A term t is normal if it has no subformula $\lambda x{:}Xa$ or $\Pi x{:}XA$, such that in the a (or A) there is a subformula $\lambda y{:}Yb$ or $\Pi y{:}YB$, for which $x = y$.

A term t is normal in a context Γ if Γ is normal, t is normal, and there is no subformula $\lambda x{:}Xa$, or $\Pi x{:}XA$ of t, such that x occurs as a left hand side in one of the Γ_i.

A term t is normal in another term g if t is normal, and there is no subformula $\lambda x{:}Xa$, or $\Pi x{:}XA$ of t, such that x is free in g.

The following makes sure that we can focus our attention on normal terms without losing generality.

Lemma 1.9

1. For every term t, there is a term u such that $t \equiv_\alpha u$, and u is normal.
2. For every normal context Γ, and for every term t, there is a term u such that $t \equiv_\alpha u$, and u is normal in Γ.
3. For every term pair of terms t and g, there is a term u, such that $t \equiv_\alpha u$, and u is normal in g.

We do not normalize contexts. Instead we make sure that non-normal contexts are never generated. We define substitution, using normality.

Definition 1.10 Let v be a variable, and let u and t be terms, such that u is normal in t. (Otherwise replace u) We define the result of *substituting* t for v in u, notation $u[v := t]$ as follows:

- If u is a variable, and $u \neq v$, then $u[v := t] = u$.
- If u is a variable, and $u = v$, then $u[v := t] = t$.
- If u has form $f \cdot g$, then $(f \cdot g)[v := t]$ equals $(f[v := t]) \cdot (g[v := t])$.
- If u has form $\lambda x{:}Xa$, then $(\lambda x{:}Xa)[v := t]$ equals $\lambda x{:}(X[v := t])(a[v := t])$.
- If u has form $\Pi x{:}Xa$, then $(\Pi x{:}XA)[v := t]$ equals $\Pi x{:}(X[v := t])(A[v := t])$.

The substitution function as we have defined it now is quite inefficient, because before the substitution can be made, the terms u and t have to be searched for non-normality, after that possibly u has to be replaced by an α-variant u', and only after this the actual substitution can be made.
This problem could be avoided by using De Bruyn variables: Replace bound variables by special symbols V_i, denoting a reference to the λ or Π that can be found i-positions upward. Using De Bruyn variables a substitution $u[v := t]$ can be performed in a single recursion on u, without having to go into t.
We have considered using De Bruyn variables, and chosen not to do so, because of the fact that in most cases the names of bound variables carry some information. Even if the variable name are of the form P_i or n_i then they still contain some type information because most users use different variable types for integers, propositional symbols etc. With De Bruyn indices all this information would be lost. Normality conflicts are rare enough to make it worth making at least an attempt to preserve the original variable names.
We can now describe the reductions:

Definition 1.11 We define the following reductions: Let Γ be a normal context, let t be a term that is normal in Γ. We define:

β-reduction Term t β-reduces to u if u can be obtained from t by replacing one subterm of the form $(\lambda x{:}X f) \cdot g$ by $f[x := g]$.

γ-reduction Term t γ-reduces to u if there is a rewrite rule of the form

$$\lambda x_1{:}X_1 \cdots \lambda x_n{:}X_n(a \Rightarrow b) \in \Gamma,$$

(assume that the rewrite rule is normal) and a sequence of terms $u_1, \ldots, u_n$, such that $a[x_1 := u_1, \ldots, x_n := u_n]$ occurs in t and u is obtained by replacing one occurrence of $a[x_1 := u_1, \ldots, x_n := u_n]$ by $b[x_1 := u_1, \ldots, x_n := u_n]$.

δ-reduction Term t δ-reduces to u if u is obtained by replacing one occurrence x by y, and there is a definition $x := y{:}Y$ in Γ.

η-reduction Term t η-reduces to u if u is obtained by replacing one subterm of the form $\lambda x{:}X(f \cdot x)$, for which x is not free in f, by f.

The definition of γ reduction is complicated. In a rewrite rule $\lambda x_1{:}X_1 \cdots \lambda x_n{:}X_n(a \cdot x_1 \cdot \ldots \cdot x_n \Rightarrow b \cdot x_1 \cdot \ldots \cdot x_n)$, the x_i should be interpreted as variables which are to be unified with terms. If the unification succeeds, then the term has form $a \cdot u_1 \cdot \ldots \cdot u_n$. In that case it can be replaced by $b \cdot u_1 \cdot \ldots \cdot u_n$. The normality ensures that we don't have to worry about variables in $b \cdot u_1 \cdot \ldots \cdot u_n$ being caught by λ's or Π's. If the rewrite rule introduces a new variable, then this variable must occur somewhere in Γ as a left hand side. Then by normality this new variable is not caught.
In order to define a calculus one needs some primitive types. The primitive types are usually called sorts.

Definition 1.12 A *sort typing* $\mathcal{S}$ is a set of the form $\{(\sigma_1, \tau_1), \ldots, (\sigma_n, \tau_n)\}$. where all σ_i and τ_i are variables, s.t. $\sigma_i = \sigma_j$ implies $\tau_i = \tau_j$. The σ_i and the τ_j are together called *sorts*.

Example 1.13 An example of a sort typing is

$$\mathcal{S} = \{(\mathbf{Prop}, \mathbf{Type_0}), (\mathbf{Type}, \mathbf{Type_0})\}.$$

Prop, **Type**, and $\mathbf{Type_0}$ are sorts.

It can be shown that **Type** cannot be a type by itself. ([Gir72]).

Definition 1.14 Let $\mathcal{S} = \{(\sigma_1, \tau_1), \ldots, (\sigma_n, \tau_n)\}$ be a sort typing. We define the typing rules dependent on $\mathcal{S}$, and the rules for well-formed contexts.

- The empty context is well-formed.
- If Γ is well-formed, and $\Gamma \vdash Y{:}\sigma$, where σ is a sort, and the variable y does not occur in Γ and is not a sort, then $\Gamma, y{:}Y$ is well-formed.
- If Γ is well-formed, and $\Gamma \vdash y{:}Y$ and $\Gamma \vdash Y{:}\sigma$ for a sort σ, and x does not occur in Γ and is not a sort, then $\Gamma, x := y{:}Y$ is well-formed.
- If Γ is well-formed, and $\lambda x_1{:}X_1 \ldots \lambda x_n{:}X_n(a \Rightarrow b)$ is a rewrite rule, and for each X_i there is a sort σ_i, such that $\Gamma \vdash X_i{:}\sigma_i$, and there is an A such that $\Gamma \vdash (\lambda x_1{:}X_1 \ldots \lambda x_n{:}X_n a){:}A$, and $\Gamma \vdash (\lambda x_1{:}X_1 \ldots \lambda x_n{:}X_n b){:}A$, and there is a sort σ, such that $\Gamma \vdash A{:}\sigma$, then $\Gamma, \lambda x_1{:}X_1 \ldots, \lambda x_n{:}X_n(a \Rightarrow b)$ is well-formed.

The typing rules:

sort If $(\sigma, \tau) \in \mathcal{S}$ and Γ is well-formed then $\Gamma \vdash \sigma{:}\tau$.

decl If Γ is well-formed and $y{:}Y$ in Γ, then $\Gamma \vdash y{:}Y$.

def If Γ is well-formed, and $(x := y{:}Y)$ is in Γ, then $\Gamma \vdash x{:}Y$.

appl If Γ is well-formed,
$\Gamma \vdash f{:}(\Pi x{:}X A)$, and $\Gamma \vdash (\Pi x{:}X A){:}\sigma_1$ for a sort σ_1, and $\Gamma \vdash t{:}X$, and $\Gamma \vdash X{:}\sigma_2$ for a sort σ_2, then $\Gamma \vdash (f \cdot t){:}A[X := t]$.

lambda If $\Gamma, x{:}X$ is well-formed, and $\Gamma, x{:}X \vdash a{:}A$, and $\Gamma, x{:}X \vdash A{:}\sigma$ for a sort σ, then $\Gamma \vdash (\lambda x{:}X a){:}(\Pi x{:}X A)$.

pi If $\Gamma, x{:}X$ is well-formed, and $\Gamma, x{:}X \vdash A{:}\sigma$ for a sort σ, then $\Gamma \vdash (\Pi x{:}X A){:}\sigma$.

conv If $\Gamma \vdash a{:}A_1$, $\Gamma \vdash A_1{:}\sigma$, and $\Gamma \vdash A_2{:}\sigma$ for a sort σ, and $A_1 \equiv_{\alpha\beta\gamma\delta\eta} A_2$ in the context of Γ, then $\Gamma \vdash a{:}A_2$.

The typing rules are complicated. As they stand here they are not appropriate for implementation due to the presence of the rule **conv**. In Section 3.4, there will be an alternative set of typing rules, that is more fit for implementation.

2. Use of the TYPO System

The TYPO system is a straightforward type checker, which supports little automation. It has been designed with the following design goals in mind:

1. The technical problems/limitations in installing TYPO should be as small as possible. In order to achieve this the program must be portable to as many computers as possible. For this reason the program is written in C. This made it possible to keep the use of resources modest, and to keep the system independent of ill-defined, non-portable, resources consuming AI languages.

2. The proofs, i.e. the λ-terms that the program creates must be visible, and the user must have good acess to them. It is always claimed an advantage of type theory that the proofs are simple λ-terms, and that λ-terms can be easily checked by another type checker. In order to achieve this goal the terms must be accessible.

3. Although the program is written in C, the code must be readable and maintainable. For this reason the program should not be optimized for speed/low memory use.

The following things did not have high priority:

1. Sophisticated Calculi. Most of mathematics can be formalized in relatively weak calculi. At this moment it is more important to have the patience to make some formalizations, then to design stronger calculi.

2. Facilities for constructing large proofs. We do not expect the appearance of very large databases very soon. Such facilities are not important. For example LATEX has no structuring commands, and complete books are written in it, without problems.

3. Perfection of the proof editor. It is under certain conditions possible to produce incorrect proofs in the proof editor. Because of this the fact that a proof has been entered is not a guarantee that it is correct. Such a guarantee exists only when the proof has been scanned completely in the Readfile-mode.

The purpose of the system is to develop formal proofs in type theory. This is done by constructing an ASCII text file, that consists of the declarations, definitions, definitions of rewrite rules, and the theorems, together with their proofs. With the exception of the proof editor, and facilities for experimenting with λ-terms, the system is non-interactive.
A typical developement of a proof goes as follows: First outside the TYPO system, in a text editor, a file containing necessary definitions and declarations is constructed. If the user wants it is possible to develop some complicated terms inside TYPO, and then to add them into the file.
When the file is read by TYPO, and if it has been accepted, TYPO contains a context in which proofs can be developed. It is possible to specify a goal type, and to gradually build up a λ-term that has the desired goal type. When the term is complete, it can be saved, and added to the context file, which after that can be further extended by more definitions/declarations or theorems.
The TYPO system can be in one of two states:

1. The top level. This is the state in which the system enters after it is started. In this state it is possible to build the internal context, and to enter the proof editor.

2. The proof editor. The proof editor is entered from the top level by specifying a goal to be proven. The editor passes through a set of *proof states.* A proof state essentially consists of a λ-term with open ends that represent details which are not yet filled in. The editor commands replace the open ends by complete λ-terms.

2.1 Syntax of the Input

The syntax of the λ-terms is as follows: (See also Section 3.1)

- Variables can be represented by a string of letters, digits, and the special symbols _, ?, !, and $. Examples are:

  ```
  Drink_Coca_Cola!, should_I_really?, _$123_, _!, 0.
  ```

  There is no condition that variables should start with a letter, since numbers have no special status in TYPO.

- Terms of the form $\lambda x{:}X A$ are represented as

  ```
  [ x : X ] A.
  ```

  Iterated λ's can be grouped together. $\lambda x_1{:}X \ldots \lambda x_n{:}X A$ can be represented as

  ```
  [ x1, ..., xn : X ] A.
  ```

- Terms of the form $\Pi x{:}X A$ are represented as

  ```
  { x : X } A.
  ```

  Iterated Π's can also be grouped together. $\Pi x_1{:}X \ldots \Pi x_n{:}X A$ can be represented as

  ```
  { x1, ..., xn : X } A.
  ```

- Terms of the form $A \to B$ are represented as

  ```
  A -> B.
  ```

  There is no internal format for terms of type $A \to B$, since they are replaced by $\Pi x{:}AB$, with a fresh variable x.

- Terms of the form $A \Rightarrow B$ are represented as

  ```
  A => B.
  ```

- Terms of the form $A \cdot B$ are represented as

  ```
  A B.
  ```

- Every λ-term should be ended by a dot (.).

As it stands now the representation of the λ-terms is ambigious. For example we did not specify how the following strings should be read:

```
A B -> C.
[ x : Nat ] P x -> Q x.
```

For this reason there are the following scoping rules:

1. In the following expressions B is as large as possible:

   ```
   A -> B.   A => B.   [ x : X ] B.   { x : X } B.
   ```

2. In the following expression B is as small as possible.

   ```
   A B,
   ```

Rule 1 is applied from left to right. Rule 2 is applied after rule 1 has been applied. In this way the rules resolve all ambiguity. For example the following expressions

```
a -> b -> c.   a b c.   a -> b c.   a b -> c.
```

are read as

```
a -> ( b -> c ).  ( a b ) c.  a -> ( b c ).  ( a b ) -> c.
```

Parentheses (and) can be used to group operators different, for example as in:

```
( a -> b ) -> c.   a ( b c ).    ( [ x : Nat ] x ) three.
```

All input to TYPO is given in the form of λ-terms. All commands have form

```
Command A1 A2 ... An.
```

If the arguments are non-variable, then they (except the last), need to parenthesized.

2.2 The Top Level
In the top level it is possible to build the context, and to check proofs. It is also possible to enter the proof editor. On the top level there are the following commands:

Readfile F Start reading from file F. Files are read in a nested fashion. When from file F another file F_1 is opened the reading will continue in file F when file F_1 is finished. If the filename is complicated, it should be quoted using

’"’

End This command, when it appears in a file, stops the reading of this file.

Halt Leave the TYPO system. This command is possible only on top level.

Edit G Enter the proof editor, with goal G.

2.3 The Proof Editor
The proof editor is entered by the command **Edit** A, where A is a type of which a proof is to be found. The proof editor can be entered from the top level. If the proof editor is entered while reading a file, it will not continue reading the file after the editing is finished.
The proof editor always is in a *state*. A state of the proof editor consists of the following components:

- A goal type. This is the theorem to be proven, or the type of which an inhabitant is being constructed.
- A proof term. This is the λ-term that is being constructed. The proof term may contain so called *metavariables* that act as parts that are not yet complete.
- A list of the types of the meta variables. When the list is empty, the proof is complete.
- The main metavariable. This indicates the part of the proof that the user is currently working on.
- The proofstatus. This is one of two possibilities. Finished, or Unfinished. The proof is finished if there are no metavariables in the proof term.

In the initial state, the proof term equals a metavariable, and the list of metavariable types consists of one element declaring the proof term having the goal type. The main metavariable is the unique metavariable. The proofstatus is unfinished. The following commands control the states of the editor:

Allgoals Show all the goals that belong to the current proof state.

Goal Show the current goal, together with its context.

Goal M Make the metavariable M the current goal.

Back Go back one step in the list of proof states

Forward Move back one step forward in the list of proof states.

Proofterm Show the proof term under construction.

Forget Forget the current goal, and leave the proof editor.

Restart Restart editing the current goal.

Save Save the proof term. TYPO gives a warning if the proof term is not complete.

The following commands do the actual editing:

Variables Close all goals that can be replaced by a variable term.

Addgoal A Add a goal A to the list of goals.

Lambda Try to obtain the current goal as a term of the form $\lambda x{:}X a$, where x is a fresh variable, chosen by the system.

Lambda x Try to obtain the current goal as a term of the form $\lambda x{:}X a$, where x should be a fresh variable.

Apply f Try to find an n and types $A_1, \ldots, A_n$, such that when $a_1{:}A_1, \ldots, a_n{:}A_n$, the term $f \cdot a_1 \cdot \ldots \cdot a_n$ has the current goal type. (See also Section 3.5)

2.4 Trace

The TYPO system has two trace options. They have been added because it is sometimes interesting to see the type checker at work, and to find out why a certain proof attempt does not work. There is a trace flag for the type checker, and one for the Apply-command.

Trace Select trace on for the type checker.

Notrace Select trace off for the type checker.

Traceapply Select trace on for apply.

Notraceapply Select trace off for apply.

These commands can be used both in the proof editor, and on top level.

2.5 Halt

The command Halt leaves TYPO. This command does not work from the proof editor, to give protection against accidental loss of the proof.

2.6 Commands for Context Maintenance

The TYPO system internally builds a context, that is used for the checking of proofs, and by the proof editor. The following commands are used to maintain the context. They can be used from the top level.

Var/Decl y Y y should be a variable that does not yet occur in the context. Y should have a type that can be a sort. The pair $y{:}Y$ is added to the context.

Define/Abbreviate x y x should be a variable that does not yet occur in the context. y should be typable in the context. The definition $x := y{:}Y$ is added to the context, where Y is a type of y.

Define/Abbreviate $x\ y\ Y$ x should be a variable that does not yet occur in the context. y should have type Y in the current context. The definition $x := y{:}Y$ is added to the context.

Theorem/Lemma $x\ y\ Y$ x should be a variable that does not yet occur in the context. First it is checked that y has type Y, i.e. that y is a proof of Y, and Y is typable as a sort. After that the declaration $x{:}Y$ is added to the context. When one refers to a theorem, one does not to refer to the proof, but only to the truth of the theorem.

Rewrite $A[x1 : X1]\cdots[xn : Xn](B_1 \Rightarrow B_2)$ A should be a variable that occurs in the context. The rewrite rule $B_1 \Rightarrow B_2$ is added to the context, as a rule that is relevant for A.

Context Show the complete internal context.

Clear Clear the internal context. This is possible only on top level.

2.7 Normalization and Reduction Commands
The following commands make it possible to compute reductions of terms. The reduce-commands make a one step reduction. The normal-commands construct a normal form. These commands store their arguments and sometimes their results in a list of terms that is called the ***history***, s.t. they can be used later without having to type them again. They can be used in the following two manners:

1. In the proof editor they can be called without argument. In that case they apply to the current goal. They will try to make the desired reduction on the current goal, and if successful, replace the current goal by the result. The commands with an exclamation mark cannot be used in the proof editor.
2. In the proof editor and on top level, the commands can be called with an argument. The argument is always stored in the history. They will try to make the reduction on the argument, and print the result. They will also try to print the type of the argument, if there exists one. The commands with an exclamation mark act as their counterparts without exclamation mark, but they store the result of the reduction in the history, instead of the argument.

Betareduce/Betareduce! Make a left most outermost β or η-reduction.

Betanormal/Betanormal! Construct a $\beta\eta$-normal form. Note that ill-typed terms do not necessarily have a normal form. In such cases TYPO does not behave well.

Gammareduce/Gammareduce! Make a left most outermost γ-reduction.

Gammanormal/Gammanormal! Compute a γ-normal form. Note that the rules in the definition of rewrite rules are very liberal, and that there may not exist a normal form. The normal form may be also not unique. No reasonable behaviour should be expected from TYPO in such situations.

Deltareduce/Deltanormal! Make a left most outermost δ-reduction, and compute a $\beta\eta$-normal form of the result.

Deltanormal/Deltanormal! Compute a δ-normal form, and compute a $\beta\eta$-normal form of the result.

Normal/Normal! Compute a $\beta\gamma\delta\eta$ normal form.

Reduce/Reduce! Make a left most outermost $\beta\gamma\delta\eta$-reduction.

The following command can be used only in the proof editor: It enables replacements that are not reductions.

Replace A If A is $\alpha\beta\gamma\delta\eta$-equivalent to the current goal, then replace the current goal by A.

The following command can be used both in the proof editor, and on top level:

Type A Compute the type of A if it exists, and store both A and the type of A in the history.

2.8 The History

The TYPO system contains a list of terms called the *history.* The purpose of the history is to save the user from the task of having to type complicated terms many times.
Some commands that are likely to have complicated arguments store their arguments in the history. These commands are the reduction commands of Section 2.7, and the Apply-command. If a term is added to the history there will appear a signal of the form $[n]$ on the screen, where n is an integer. They can be referred to by $\%n$. The following command makes it possible to see the history:

History Show the whole history.

Terms are not stored infinitely long. The makro MAXHISTORY in file decl.h determines how many terms are kept.

3. Relevant Aspects of the Implementation

3.1 Reading of λ-Terms

In this Section we formally describe the language in which the λ-terms are represented. We begin by defining the token set. After that we will define the rewrite rules for the language.

Definition 3.1

- An *identifier* (used to represent variables) consists of a string of length $\{1, 2, 3, \ldots\}$ taken from the following character set:

  ```
  '0'..'9', 'a'..'z', 'A'..'Z', '?', '!', '_', '$'
  ```

  Numbers are not treated specially. The program cannot calculate. Identifiers also can have another form: A string starting and ending with

  ```
  '"'
  ```

  and with in between all characters except

  ```
  '"'
  ```

  and the end-of-line character, is also an identifier. The only purpose of identifiers of this type is the representation of filenames. They are accepted by the type checker, but we do not recommend their use in proofs.
- A *metavariable* consists of the symbol

  ```
  '?',
  ```

  followed by $\{0, 1, 2, 3, 4\}$ characters from the following character set:

  ```
  '0'..'9'
  ```

- A *history reference* consists of the symbol

```
'%',
```

followed by $\{1, 2, 3, 4\}$ characters from the following character set:

```
'0'..'9'
```

- The following strings each are one token:

  ```
  '->'    '=>'
  ```

  The first is used for $\rightarrow$, the second for $\Rightarrow$,.
- The following characters each are one token:

  ```
  '['  ']'  '{'  '}'  '('  ')'  '.'  ','  ':'  EOF.
  ```

  EOF is the end of file symbol.
- *Comment* has form

  ```
  /* ... */
  ```

  On the place of ... is allowed every character string not containing

  ```
  */
  ```

  Comment does not result in a token.
- *Whitespace* consists of $\{1, 2, 3, \ldots\}$ occurrences of one of the following characters:

  ```
  ' ', '\t', '\n'
  ```

  Whitespace does not result in a token.

We will now give the grammar of the λ-terms. The binding rules of Section 2.1 cannot be handled by usual operator priorities because the operators have different binding strength on the left and on the right. For example

```
[ x : Nat ] P x -> Q x.
```

means $\Pi x{:}\mathrm{Nat}(Px \rightarrow Qx)$, where

```
P x -> [ x : Nat ] Q x
```

means $Px \rightarrow (\lambda x{:}\mathrm{Nat}\ Qx)$. In the first term the $\rightarrow$ was stronger, in the second the λ was stronger. Another problem arises when application terms appear in a context:

```
A B -> C D.
```

Both AB and CD are applications, and one might decide to make one non-terminal symbol App, standing for an possible application term, but this would be not correct. Left of the $- >$, B cannot be replaced by

```
[x:Nat] P x
```

without adding parentheses, while right of the $->$, D can be replaced. So we need two non-terminals for product. (In the grammar this will be Term2 and Term3). Keeping all this in mind we arrive at the following grammar. The grammar is LALR, and has been processed by the LALR parser generator Maphoon.

Definition 3.2 The following grammar defines the input format of the TYPO system. $-->$ used for the rewrite rule. $->$ is a token.

```
Usercommand --> Term1245 .

Term1245 --> Term1
         --> Term2
         --> Term4
         --> Term5

Term35   --> Term3
         --> Term5

Term45   --> Term4
         --> Term5


Term1    --> Term35 -> Term1245
         --> Term35 => Term1245

Term2    --> Term35 Term45

Term3    --> Term35 Term5

Term4    --> [ Vars : Term1245 ] Term1245
         --> { Vars : Term1245 } Term1245

Term5    --> ( Term1245 )
         --> variable
         --> metavariable
         --> history


Vars     --> variable
         --> Vars , variable
```

3.2 Output of λ-Terms

In this section we give an algorithm for printing λ-terms in the input format of the previous section. This is not straightforward if one wants to be economical with parentheses. The algorithm must be able to decide when parentheses can be dropped. In the following λ-term, the second pair of parentheses can be dropped, but the first cannot:

```
( A -> B ) -> ( C -> D ).
```

Parentheses are necessary when the λ-term that is being printed is not a variable, and the main operator pulls terms away from the context into its own scope. This is the case with the $->$ from the first subterm. Without parentheses it would greedily pull the whole $B \to C \to D$ into its scope. We classify which contexts are possible:

C0: There is no problematic context, as in the following terms A :

```
A.    B -> A.    [ x : A ] A.   { x : A } A.
```

and

```
( A )   ( B -> A )   ( [ x : A ] A )   ( { x : A } A )
```

C1: There is right context which makes that terms of the form

```
A -> B     { x : A } B     [ x : A ] B
```

need to be parenthesized, but no problematic left context, as is the case with the A in the following terms:

```
A B       A -> B
```

C2: There is left context which makes that terms of the form

```
A B
```

should be parenthesized, but no problematic right context. This is the case with the A in the following terms:

```
B A.
```

C3: There is both left and right context, which makes that all non-variable terms need to be parenthesized. This is the case with the A in the following term:

```
B A -> C.
```

The following table describes how the contexts proceed, and whether or not parentheses are needed: We omitted the terms built by { and }, because they behave the same as the terms built by [and]. Each row starts with the type of context. After that the possible types of terms are listed, with or without parentheses, with the type of context that the subterms will be in:

C0:	[x : C0] C0	C1 -> C0	C1 C2
C1:	([x : C0] C0)	(C1 -> C0)	C1 C3
C2:	[x : C0] C0	(C1 -> C0)	(C1 C2)
C3:	([x : C0] C0)	(C1 -> C0)	(C1 C2)

So a term of type $A \to B$ should be parenthesized in context $C2$, and A has context $C1$, and B has context $C0$.
When the output function encounters a term of the form $\Pi x_1{:}X_1 \cdots x_n{:}X_n A$, it has to check up to what i,

$$X_1 = \cdots = X_i,$$

because then the notation

```
{ x1, ..., xi : X1 } ...
```

can be used. The same check has to be made for the λ. Additionaly it has to check in the case of Π, which x_i are not free in $\Pi x_{i+1}{:}X_{i+1} \cdots x_n{:}X_n A$, because these terms have to be replaced by $X_{i+1} \to \cdots$. The print function prefers using the $\to$ over combining types. That means that the term $\Pi x{:}X y{:}X z{:}X (p \cdot x \cdot z)$ will be printed as

```
{ x : X } X -> { z : X } p x z.
```

and not as

```
{ x, y, z : X } p x z.
```

3.3 Internal Data Format and the Garbage Collector
As all computer programs that deal with tree structures, TYPO has a garbage collector. It can be found in the file **memory.c**, and has a size of app. two pages.
The system uses two heaps. The first is called **termspace**, and the other is called **termspace2**. Both are arrays of integers. The array **termspace2** is used only by the garbage collector. The other functions use only **termspace**. The garbage collector simultaneously marks and copies the terms that are in use from **termspace** to **termspace2**. (Function **markandcopy**) By having two heaps the garbage collector can be linear in the amount of data, (as opposed to linear in the size of **termspace**). When the garbage collector is finished it copies the data back from **termspace2** to **termspace**. The main disadvantage of this method is that there is a factor two increase in the memory that is used. The advantages are the simplicity of the garbage collector, and the gain in speed. We decided that the additional use of memory is acceptable, because it is low enough. We now describe the internal data format, and give an outline of the algorithm of the garbage collector.

Definition 3.3 We define how λ-terms are internally represented. All λ-terms are stored in the array **termspace**. **termspace** is of type integer. The integer **freetermspace** indicates the first position that is not in use. The integer MAXTERMSPACE equals the size of **termspace**.

Every variable is replaced by a unique integer, that marks its place in the name table. We call this integer an *identifier*. Every term has a position in **termspace**. We call this position the *index*.

- Terms of the form $\lambda x{:}X a$ are represented by the following sequence of integers:
  - The special integer OP_LAMBDA (= 256).
  - The identifier belonging to x.
  - The index of X.
  - The index of a.
- Terms of the form $\Pi x{:}X A$ are represented in the same manner as terms built by λ. The only difference is that the special integer OP_PI (= 272) is used.
- Terms of the form $f \cdot g$ are represented by the following sequence of integers:

1. The special integer OP_APPLY. (= 288)
2. The index of f.
3. The index of g.

- Variable terms of the form v are represented by the sequence:
  1. The special integer OP_VAR. (= 304)
  2. The identifier belonging to v.
- Metavariable terms of the form $?n$ are represented by
  1. The special integer OP_METAVAR. (= 320)
  2. The integer n.
- Terms of the form $a \Rightarrow b$ are represented as:
  1. The special integer OP_REWRITE. (= 336)
  2. The index of a.
  3. The index of b.

Next we will give the algorithm of the garbage collector. It replaces the records that are used by another type of record, marked by OP_COPIED (= -1), followed by the index in **termspace2** to which the record is copied.

Definition 3.4 Procedure **markandcopy**(n) copies the term on index n in **termspace** to **termspace2** and returns the resulting index in **termspace2**.

```
Algorithm markandcopy ( n )

if termspace [n] = OP_COPIED then
   return termspace [ n + 1 ];

if termspace [n] = OP_LAMBDA or termspace [n] = OP_PI then
begin
   termspace [ n + 2 ] = markandcopy ( termspace [ n + 2 ] );
   termspace [ n + 3 ] = markandcopy ( termspace [ n + 3 ] );
   let m be the first free position in termspace2.
   termspace2 [ m ] = termspace [ n ];
   termspace2 [ m + 1 ] = termspace [ n + 1 ];
   termspace2 [ m + 2 ] = termspace [ n + 2 ];
   termspace2 [ m + 3 ] = termspace [ n + 3 ];
   termspace [ n ] = OP_COPIED;
   termspace [ n + 1 ] = m;
   return m;
end

if termspace [b] = OP_REWRITE or termspace [n] = OP_APPLY then
begin
   termspace [ n + 1 ] = markandcopy( termspace [ n + 1 ] );
   termspace [ n + 2 ] = markandcopy( termspace [ n + 2 ] );
   let m be the first free position in termspace2.
   termspace2 [ m ] = termspace [ n ];
   termspace2 [ m + 1 ] = termspace [ n + 1 ];
   termspace2 [ m + 2 ] = termspace [ n + 2 ];
   termspace2 [ n ] = OP_COPIED;
```

```
      termspace2 [ n + 1 ] = m;
      return m;
   end

   if( termspace [b] = OP_VAR or termspace [ n ] = OP_METAVAR then
   begin
      let m be the first free position in termspace2.
      termspace2 [ m ] = termspace [ n ];
      termspace2 [ m + 1 ] = termspace [ n + 1 ];
      termspace2 [ n + 1 ] = OP_COPIED;
      termspace2 [ n + 2 ] = m;
      return m;
   end
```

When a term is being built, and **termspace** is full, the garbage collector is invoked. It copies the terms that are in use to **termspace2** by invoking **markandcopy** for every term that is in use, as follows:

```
t = markandcopy( t );
```

After that it will copy the terms back from **termspace2** to **termspace**. When the garbage collection is finished it is checked if there is enough space to build the term that was initially requested. If there is not enough space an out of memory error is generated.
The garbage collector needs to know which terms are in use. Terms that are in the array **termstack**[i], for an i with $0 \leq i <$ **freetermstack** are always assumed to be in use. The array **termstack** is used by the parser, when reading a term, and by some functions.
It is also possible to use an arbitrary variable of type integer for storing terms. In that case it is necessary to declare the variable to the garbage collector. For this purpose there is the array **gc** [ ] containing pointers to integers. All pointers between **gc**[0] and **gc**[**freegc**] are in use. A pointer can be inserted into **gc** using the call

```
k = declare ( & i );
```

where i is the integer that is being declared. The integer k must be remembered and used when i is undeclared.

```
undeclare ( k );
```

It is assumed that variables are declared and undeclared in a tree order: The sequence

```
k = declare( & i1 );
(void) declare ( & i2 );
(void) declare ( & i3 );
i1 = termnil;
i2 = termnil;
i3 = termnil;     /* declared terms should be initialized. */
be busy with ( i1, i2, i3 );
undeclare ( k );
```

declares all the i's, and disposes them after use.
There are several things that might go wrong in this delicate process, and one has to be very careful if one wants to write a function using λ-terms:

1. A variable containing a term that is to be used later has not been declared. If in between the garbage collector has been invoked, then probably the term has been moved. Since the variable did not change it does not refer to the term anymore.

2. A variable that has been declared, does not contain a term. When the garbage collector is called it will try to interpret the variable as a term.

3. A variable containing a term has been declared twice, or a variable in **termstack** has been declared. The first time that **markandcopy** is called everything goes fine. On the second call the variable does not refer to **termspace** anymore, but to **termspace2**. Unfortunately **markandcopy** assumes that the variable still refers to **termspace**.

Problems with the garbage collector can be very difficult to find because of their irreproducible nature. A function may work many times well before the garbage collector is called and things get confused. In order to make debugging possible there is a flag **DEBUG** in the file **memory.c**. When this flag is set, the garbage collector will be called every time a term is created, and it will try to move objects as much as possible. This makes it likely that bugs involving the garbage collector are caught. If one writes a new function one should always test it with this flag set.

3.4 the Type Checking Algorithm
The type checking rules, as they were given in Definition 1.14, are not well-suited for implementation. The cause is the **conv**-rule. If this rule would not exist then there would be one typing rule for each operator. With the **conv**-rule an algorithm has to backtrack, first between the **conv** rule and a structural rule, and second, when choosing for the **conv**-rule, which term to use.
The solution for this problem is to replace conversion by reduction, and to use reduction only when a conflict arises. This can only happen in the **appl**-rule. This leads to the following calculus.

Definition 3.5 We define the following functions and predicates:

- $\Phi(\Gamma, t)$ equals the canonical type of t in context Γ. It is assumed that Γ is well-formed, and t is normal in Γ.
- $\Sigma(\Gamma, \sigma)$ is true if σ is a sort, or can be reduced to a sort.
- $\text{Abcde}(\Gamma, t_1, t_2)$ is true if t_1 and t_2 are $\alpha\beta\gamma\delta\eta$-equivalent in the context Γ.
- $\Pi(\Gamma, A)$ equals a term of the form $\Pi x{:}XB$ if A can be $\beta\gamma\delta\eta$-reduced to a term of this form.

Here are the recursive definitions: First $\Pi(\Gamma, t)$.

- If A has form $\Pi x{:}XB$, or A cannot be $\alpha\beta\gamma\delta\eta$-reduced, then $\Pi(\Gamma, A) = A$.
- Otherwise let A' be obtained from A by one $\alpha\beta\gamma\delta\eta$-reduction. Then $\Pi(\Gamma, A) = \Pi(\Gamma, A')$.

Next $\text{Abcde}(\Gamma, t_1, t_2)$.

- If $t_1 \equiv_\alpha t_2$, then $\text{Abcde}(\Gamma, t_1, t_2)$ is true.
- If $t_1 \not\equiv_\alpha t_2$, and both t_1 and t_2 cannot be $\alpha\beta\gamma\delta\eta$-reduced, then $\text{Abcde}(\Gamma, t_1, t_2)$ is false.
- If $t_1 \not\equiv_\alpha t_2$, and u_1 is obtained by one $\alpha\beta\gamma\delta\eta$-reduction from t_1, then $\text{Abcde}(\Gamma, t_1, t_2) = \text{Abcde}(\Gamma, u_1, t_2)$.
- If $t_1 \not\equiv_\alpha t_2$, and u_2 is obtained by one $\alpha\beta\gamma\delta\eta$-reduction from t_2, then $\text{Abcde}(\Gamma, t_1, t_2) = \text{Abcde}(\Gamma, t_1, u_2)$.

Finally $\Sigma(\Gamma, \sigma)$ and $\Phi(\Gamma, \sigma)$.

- If σ is a sort, then $\Sigma(\Gamma, \sigma)$ is true.
- If σ is not a sort, and σ cannot be $\alpha\beta\gamma\delta\eta$-reduced, then $\Sigma(\Gamma, \sigma)$ is false.

- Otherwise, if $\tau = \Phi(\Gamma, \sigma)$, and $\Sigma(\Gamma, \tau)$, and σ_1 is obtained by $\alpha\beta\gamma\delta\eta$-normalization from σ, then $\Sigma(\Gamma, \sigma) = \Sigma(\Gamma, \sigma_1)$.

- If t is a variable, and there is a $(t, \sigma) \in \mathcal{S}$, then $\Phi(\Gamma, t) = \sigma$.
- If t is a variable, not a sort, and there is a $t{:}X$ in Γ, then
$$\Phi(\Gamma, t) = X.$$
- If t is a variable, not a sort, and there is a $t := x{:}X$ in Γ, then
$$\Phi(\Gamma, x) = X.$$
- If t is of the form $\lambda x{:}X a$, $\Phi(\Gamma, X) = \sigma$, $\Sigma(\Gamma, \sigma)$, and $\Phi(\ (\Gamma, x{:}X), a) = A$, then
$$\Phi(\Gamma, \lambda x{:}X a) = \Pi x{:}X A.$$
- If t is of the form $\Pi x{:}X A$, $\Phi(\Gamma, X) = \sigma$, $\Sigma(\Gamma, \sigma)$, $\Phi(\ (\Gamma, x{:}X), A) = \sigma'$, and $\Sigma(\ (\Gamma, x{:}X), \sigma')$, then
$$\Phi(\Gamma, \Pi x{:}X A) = \sigma'.$$
- If t is of the form $f \cdot g$, $F = \Phi(\Gamma, f)$, $\Pi(\Gamma, F)$ is of the form $\Pi x{:}X A$, $G = \Phi(\Gamma, g)$, and $\text{Abcde}(\Gamma, X, G)$, then
$$\Phi(\Gamma, f \cdot g) = A[x := f].$$

So what happend is that the **conv**-rule has been built in into the Σ, the Abcde, and the Π-predicate. It remains to give rules for well-formed contexts:

Definition 3.6

- The empty context is well-formed.
- If Γ is well-formed, if y is a variable which is not a sort and which does not occur in Γ, $\Phi(\Gamma, Y) = \sigma$, and $\Sigma(\Gamma, \sigma)$, then $\Gamma, y{:}Y$ is well-formed.
- If Γ is well-formed, if x is a variable which is not a sort and which does not occur in Γ, $\Phi(\Gamma, Y) = \sigma$, $\Sigma(\Gamma, \sigma)$, $\Phi(\Gamma, y) = Y_1$, $\Phi(\Gamma, Y_1) = \sigma_1$, $\Sigma(\Gamma, \sigma_1)$, and $\text{Abcde}(\Gamma, Y_1, Y)$, then $\Gamma, x := y{:}Y$ is well-formed.
- If Γ is well-formed, $\Phi(\Gamma, \lambda x_1{:}X_1 \cdots x_n{:}X_n a) = A$, and $\Phi(\Gamma, A) = \sigma_1$, and $\Sigma(\Gamma, \sigma_1)$, and $\Phi(\Gamma, \lambda x_1{:}X_1 \cdots x_n{:}X_n b) = B$, and $\Phi(\Gamma, B) = \sigma_2$ and $\Sigma(\Gamma, \sigma_2)$, and $\text{Abcde}(\Gamma, \sigma_1, \sigma_2)$, then $\Gamma, \lambda x_1{:}X_n \cdots x_n{:}X_n (a \Rightarrow b)$ is well-formed.

Finally we can define the $\vdash$-relation, using the Φ-predicate.

Definition 3.7 We define when $\Gamma \vdash x{:}X$. If

1. Γ is well-formed,
2. $\Phi(\Gamma, x) = X_1$, $\Phi(\Gamma, X_1) = \sigma_1$, and $\Sigma(\Gamma, \sigma_1)$,
3. $\Phi(\Gamma, X) = \sigma$, $\Sigma(\Gamma, \sigma)$, and
4. $\text{Abcde}(\Gamma, X, X_1)$, then

$$\Gamma \vdash x{:}X.$$

3.5 Type Checking with Meta Variables

Metavariables are necessary in order to make the Apply-command work well. The Apply command, when called with a functional term f and looking for a term of type T, has the task to find a list of types $A_1, \ldots, A_n$, such when there is a list of objects $a_1, \ldots, a_n$, where each a_i has the type A_i, then $f \cdot a_1 \cdot \ldots \cdot a_n$ has type T. (After that the proof editor proceeds with determining the a_i.)
In many cases this is straightforward. If for example f has type $A \rightarrow B \rightarrow C$, the goal type is C, then it is easy to guess that we need an object of type A, and an object of type B.
However if f has a dependent type, then this can only be done with metavariables. For example the rule **andintro** has type $\Pi x{:}\mathrm{Prop}\ \Pi y{:}\mathrm{Prop}\ x \rightarrow y \rightarrow (\mathrm{and} \cdot x \cdot y)$. Then if the goal type is $(\mathrm{and} \cdot a \cdot b)$, then guessing that we are looking for objects of type a and b needs a type of unification of the terms $\mathrm{and} \cdot a \cdot b$, and $\mathrm{and} \cdot x \cdot y$. In this unification the x and y are treated as metavariables.
The Apply command works as follows: If apply is trying to apply a functon f, in a context Γ, and it should produce type T, then it will introduce n metavariables $?1, \ldots, ?n$, together with their types $?n+1, \ldots, ?n+n$. After that it will calculate the type of $f \cdot ?1 \cdot \ldots \cdot ?n$ in the context $\Gamma, ?1{:}?n+1, \ldots, ?n{:}?n+n$, and after that unify the calculated type with T. This is done by calling

$$\Phi((\Gamma, ?1{:}?n+1, \ldots, ?n{:}?n+n), f \cdot ?1 \cdot \ldots \cdot ?n),$$

(call the result T') and after that calling $\mathrm{Abcde}((\Gamma, ?1{:}?n+1, \ldots, ?n{:}?n+n), T, T')$.
In order to make this work a surprisingly small modification of Abcde is sufficient:

Definition 3.8 The following rule has to be added to the definition of Φ in Definition 3.5:

- If t is a metavariable, and $t{:}X$ in Γ, then $\Phi(\Gamma, t) = X$.

The following rule replaces the 2nd rule in the definition Abcde :

- If $t_1 \not\equiv_\alpha t_2$, and both t_1 and t_2 cannot be $\alpha\beta\gamma\delta\eta$-reduced, then let π be the leftmost and undeepest position where t_1 and t_2 differ. Let $u_1 = \pi(t_1)$, and let $u_2 = \pi(t_2)$. (So (u_1, u_2) is the leftmost difference of t_1 and t_2) Then:
  - If u_1 contains a variable that is bound on position π in t_1, then $\mathrm{Abcde}(\Gamma, t_1, t_2)$ is false.
  - If u_2 contains a variable that is bound on position π in t_2, then $\mathrm{Abcde}(\Gamma, t_1, t_2)$ is false.
  - If both u_1 and u_2 are not a metavariable, then $\mathrm{Abcde}(\Gamma, t_1, t_2)$ is false.
  - If u_1 is a metavariable, then assign $u_1 := u_2$.
  - If u_2 is a metavariable, then assign $u_2 := u_1$.

The assignments have been implemented by restart. That means that at a moment an assignemnt has to be made, the type checking algorthm is aborted, the assignment is made, and then the algorithm is restarted. This is not an optimal implementation, but is turns out fast enough.

4. Acknowledgements

Part of the developement took place in Saarbrücken at the Max-Planck Instute. I also thank Jan Zwanenburg for pointing out some mistakes in the proof editor. Part of the experience in practical aspects of type theory was obtained in a project with Brian Semmes.

References

[CQ85] T. Coquand, G. Huet, Constructions: A Higher Order Proof System for Mechanizing Mathematics, pp. 151-184, in: EUROCAL'85, Springer Verlag, LNCS 203, 1985.

[Geuv93] J.H. Geuvers, Logic and Type Systems, Ph.D. Thesis, University of Nijmegen, 1993.

[Gir72] Girard, J.-Y. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur, Thèse d' Etat (Paris, Université Paris VII, 1972.

[vBJ79] L.S. van Benthem Jutting, Checking Landau's Grundlagen in the Automath System, Mathematical Centre Tracts 83, Mathematisch Centrum, Amsterdam, 1979.

[Land30] Grundlagen der Analysis, 3d Edition, New York, Chelsea Publishing Company, 1960.

[Luo89] Z. Luo, ECC: An Extended Calculus of Constructions, pp. 386-395, in Proc. of the 4th Annual Symposion on Logic in Computer Science, IEEE Computer Society Press, 1989.

[ML75] P. Martin-Löf, An Intuitionistic Theory of Types, Predicative Part, pp. 73-118, in H.E. Rose, J.C. Shepherdson, Logic Colloquium '73, North Holland Publishing Company, 1975.

[ML84] P. Martin-Löf, Intuitionistic Type Theory, Studies in Proof Theory, Bibliopolis Napoli, 1984.

[NGV94] Selected Papers on Automath, Edited by R.P. Nederpelt, J.H. Geuvers, R.C. De Vrijer, Studies in Logic and the Foundation of Mathematics, Vol. 133, North Holland Publishing Comany, Amsterdam, 1994.