A Natural Language Explanation for Formal Proofs

Yann Coscoy*

INRIA-Sophia-Antipolis

Introduction

We study formal proofs conceived to be checked automatically. These proofs are rarely used. We think that it is partly due to the fact that it is very difficult for a human to understand the connection between what he considers a mathematical proof and its formal representation. We are interested in automatically producing text explaining these formal representations. We have chosen to produce text in natural language using the traditional vocabulary of mathematics.

More precisely, we work on a formalism called *Calculus of Inductive Constructions*, which uses a functional representation of proofs. The objective of this formalism is to allow an automatic proof checking by a simple machine. Each aspect, simplicity and automation, imposes its own specificity to the proof representation. On the one hand, the proofs contain a lot of trivialities that would be considered useless by human readers. On the other hand, some information that would be very useful to these same human readers are omitted.

In previous work [3], we have developed a machine that translates proofs of the *Calculus of Inductive Constructions* into natural language. The text produced by this machine has the same structure as the formal proof. These texts are not satisfactory. They are in general too verbose, and sometimes they omit important information. In this article, we will present an improvement of this translation based on a more precise choice of the information we include in the text.

1 A brief survey of *Calculus of Constructions*

The *Calculus of Inductive Constructions* is a formalism used to represent proofs. It is an extension of *Calculus of Constructions*, which belongs itself to a family of functional formalisms called λ-calculi. We will briefly present the pure λ-calculus, then we will introduce the λ-calculus with dependent types, and then the *Calculus of Constructions*. (We will present the *Calculus of Inductive Constructions* later in section 4.)

* Supported by *Direction de la Recherche et de la Technologie, (DGA France)*

1.1 Pure λ-calculus

The terms of λ-calculus are called "λ-terms" or "λ-expressions". They are defined by the grammar:

$$t \;:=\; v \mid c \mid (t\ t) \mid \lambda v.\, t$$

The "v" and "c" represent variables and constants. They are simply denoted by names. The "$(t\ t)$" represents applications. The term $(F\ A)$ denotes the application of the function F to an argument A. This corresponds to the more traditional notation $F(A)$. The "$\lambda v.\, t$" represents functions definitions. The term $\lambda x.\, T$ represents the function that maps the variable x to the term T. In mathematical texts, the more traditional notation is $x \mapsto T[x]$. The square brackets are used to make clear that T can depend on x. In λ-calculus this dependency is implicit.

There exist three rewriting rules which can be used to transform a λ-term into an equivalent λ-term.

$$(\alpha\text{-conversion})\ \Gamma \vdash \lambda x.\, M \longmapsto \lambda y.\, M\{x/y\}\ \texttt{if not}\ y \in \mathit{Free}(M)$$

$$(\beta\text{-reduction})\ \Gamma \vdash (\lambda x.\, M\ \ N) \longmapsto M\{x/N\}$$

$$(\delta\text{-reduction})\ \Gamma \vdash c \longmapsto D\ \texttt{if}\ c{:}= D \in \Gamma$$

The α-conversion corresponds to the renaming of local variables. The β-reduction specify how to compute the results of functions. The notation $M\{x/N\}$ means that the variable x must be substituted by the argument N within the body M of the function. The δ-reduction means that a constant can be substituted by its definition.

If the function F is defined by $F{:}= \lambda x.\, (x + a)$, then, $(F\ 3)$ can be δ-reduced in $(\lambda x.\, (x + a)\ 3)$. And, this expression can be α-converted in $(\lambda y.\, (y + a)\ 3)$ or β-reduced in $3 + a$.

This system, called pure λ-calculus, gives very few guarantees about the correctness of the defined functions. To avoid this disadvantage, several extensions of this formalism attribute *types* to λ-expressions. The *Calculus of Constructions* is one of these extensions. It is a λ-calculus with "dependent" types.

1.2 λ-calculus with dependent types

To give a mathematical meaning to λ-expressions, a *type* is attributed to each of them. The notation $M{:}T$ is used to express "M has type T". Types can be constructed with other types or other expressions. We can for example define a type Nat to designate natural numbers. So, we can for any natural n define the type $(\mathit{list}\ \mathit{Nat}\ n)$ to designate the lists of length n of natural numbers.

Consider now the function which maps a natural n to the list of length $2 * n$ that contains a repetition of 0. The domain of the function is Nat and its range is $(\mathit{list}\ \mathit{Nat}\ (2 * n))$ where n is the argument of the function. The type of this function is a *dependent* type because the range depends of the effective argument of the function. We denote the type of this function $\forall n{:}\mathit{Nat}.\, (\mathit{list}\ \mathit{Nat}\ (2 * n))$.

Consider now another function on naturals and lists that maps the natural n to the list of length 3 that contains three times n. This list has formally type $\forall n\colon Nat.(list\ Nat\ 3)$. In this case, the range $(list\ Nat\ 3)$ does not depend on the argument n. We can use a simpler notation $Nat \rightarrow (list\ Nat\ 3)$.

1.3 Proof representation

Some types, e.g., Nat or $(list\ Nat\ n)$, represent sets, but others represent propositions. To distinguish them, the *Calculus of Constructions* defines two special symbols, Set and $Prop$, called *sorts*. The types which represent sets are themselves of type Set whereas the types which represent propositions are themselves of type $Prop$.

The Curry-Howard isomorphism defines a direct correspondence between types and propositions. The symbols "$\forall$" and "$\rightarrow$" of types actually correspond to the symbols "$\forall$" and "$\Rightarrow$" of logic. This correspondence is such that if the type T represents a proposition, then any λ- term of type T represents a proof of this proposition. Checking that the λ-term follows the typing rules guarantees that the proof is valid.

2 Background

The system of translation from λ-terms into natural language presented in this section has been described in a earlier publication. A more complete description can be found in [3].

2.1 The main idea of translation

We use the word "proof" to designate a λ-term whose type is a proposition. Each proof is constructed by an operator of λ-calculus with some subterms. The subterms whose type is a proposition are also proofs. This construction of a proof with subterms can be considered as an elementary reasoning step. We determine the intuitive meaning of each of these steps studying the types of the subexpressions. Then, we produce a sentence expressing this step. This sentence contains the type of the constructed λ-term because this type represents the proposition that is proved by this part of the proof. All these sentences are grouped together in a text whose structure is precisely the structure of the λ-term.

2.2 A little development on relations

Here, we introduce the formal development on relations that we use to illustrate this article. A relation is a binary predicate on a set. This is formally defined as a function that takes as arguments a set and two elements of this set and then gives a proposition. So, we define the constant Rel by:

$$Rel\colon= \forall U\colon Set. U \rightarrow U \rightarrow Prop$$

The inverse of a relation can be formally defined with a function Inv that takes as argument a set and a relation on this set and then gives another relation. This function should have the type $\forall U{:}\, Set. (Rel\ U) \rightarrow (Rel\ U)$. It is defined by:

$$Inv := \lambda U{:}\, Set. \lambda R{:}\, (Rel\ U). \lambda x,y{:}\, U. (R\ y\ x)$$

Transitivity is a predicate on relations. It can be defined by a function $Trans$ that takes as arguments a set, a relation on this set, and then gives a proposition. This function has type $\forall U{:}\, Set. (Rel\ U) \rightarrow Prop$. Its definition is:

$$Trans := \lambda U{:}\, Set. \lambda R{:}\, (Rel\ U). \forall x,y,z{:}\, U. (R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)$$

We can notice that the proposition $(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)$ that could be understood as $(R\ x\ y) \Rightarrow ((R\ y\ z) \Rightarrow (R\ x\ z))$ is logically equivalent to the proposition $((R\ x\ y) \wedge (R\ y\ z)) \Rightarrow (R\ x\ z)$. It is a curryfied form that is more natural in λ-calculus.

Next, we want to prove that if a relation is transitive, then its inverse relation is also transitive. This assertion is formally expressed by:

$$\forall U{:}\, Set. \forall R{:}\, (Rel\ U). (Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$$

The proof that we have chosen starts by introducing an arbitrary set U and a transitive relation R. It continues by introducing three elements x, y, and z such that $(Inv\ U\ R\ x\ y)$ and $(Inv\ U\ R\ y\ z)$. Then it uses the hypothesis of transitivity and the definition of Inv to prove $(Inv\ U\ R\ x\ z)$. The λ-term that formally represents this proof is:

$$\begin{array}{l} \lambda U{:}\, Set. \lambda R{:}\, (Rel\ U). \lambda trans{:}\, (Trans\ U\ R). \\ \quad \lambda x,y,z{:}\, U. \lambda h_1{:}\, (Inv\ U\ R\ x\ y). \lambda h_2{:}\, (Inv\ U\ R\ y\ z). \\ \quad\quad (trans\ z\ y\ x\ h_2\ h_1) \end{array}$$

2.3 A first translation

The λ-abstractions correspond to proofs of propositions with universal quantifiers. The term $\lambda x{:}T.M$ (with $M{:}P$ and $P{:}\,Prop$) is a proof, in three steps, of $\forall x{:}T.P$. The proof starts by introducing an arbitrary variable x of type T. It continues with the proof M of P with this variable. Then it concludes by generalizing the proposition P to any variable x. It is always possible to translate one of these proofs by:

Consider an arbitrary x of type T.
«M»
We have proven $\forall x{:}T.P$.

Such a translation is always correct, but it is too general. It expresses correctly the structure of the proof, but it does not express its intuitive nature. The conciseness of the formalism allows an overloading of λ-abstractions, this means that they are used to introduce terms of several natures. We determine the nature of the variable x by studying its type T, then we can generate an appropriate sentence.

If T is one of the sorts *Prop* or *Set*, then x represents respectively a proposition or a set. When T is not a sort, we study its type. If T has type *Prop*, it is itself a proposition, and the λ-abstraction corresponds to the introduction of a hypothesis. But if T has type *Set*, the λ-abstraction corresponds to the introduction of an arbitrary element x of set T. We can translate the beginning of our little example with these specific sentences:

```
Consider an arbitrary set $U$.
  Consider an arbitrary $R$ of type $(Rel\ U)$.
    Assume $(Trans\ U\ R)$ $(trans)$.
      Consider an arbitrary element $x$ of $U$.
        Consider an arbitrary element $y$ of $U$.
          Consider an arbitrary element $z$ of $U$.
            Assume $(Inv\ U\ R\ x\ y)$ $(h_1)$.
              Assume $(Inv\ U\ R\ y\ z)$ $(h_2)$.
                «$(trans\ z\ y\ x\ h_2\ h_1)\colon (R\ z\ x)$»
              We have proven $(Inv\ U\ R\ y\ z) \Rightarrow (R\ z\ x)$.
            We have proven $(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (R\ z\ x)$.
          We have proven $\forall z\colon U.(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (R\ z\ x)$.
        We have proven $\forall y,z\colon U.(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (R\ z\ x)$.
      We have proven $\forall x,y,z\colon U.(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (R\ z\ x)$.
    We have proven $(Trans\ U\ R) \Rightarrow \forall x,y,z\colon U.\cdots$.
  We have proven $\forall R\colon (Rel\ U).(Trans\ U\ R) \Rightarrow \forall x,y,z\colon U.\cdots$.
We have proven $\forall U\colon Set.\forall R\colon (Rel\ U).(Trans\ U\ R) \Rightarrow \forall x,y,z\colon U.\cdots$.
```

We now translate the central part of the proof *i.e.* $(trans\ z\ y\ x\ h_2\ h_1)$. We also proceed by type analysis, we can determine that this application corresponds to the mathematical application of hypothesis *trans* to two subproofs. These two subproofs are the variables h_2 and h_1 that represent hypotheses. We can complete the text by:

```
* With hypothesis $trans$ we have $(Trans\ U\ R)$.
* With hypothesis $h_2$ we have $(Inv\ U\ R\ y\ z)$.
* With hypothesis $h_1$ we have $(Inv\ U\ R\ x\ y)$.
Applying the first result to the two others we get $(R\ z\ x)$.
```

2.4 A shorter translation

The text that we obtain by grouping these two parts is obviously too long and too repetitive. To avoid this, we use more complex sentences that express several proof steps at a time. For λ-abstractions, we use a sentence that introduces simultaneously several variables. For applications, we take advantage of the fact that all subproofs are hypotheses to produce one single sentence. The text that we finally get is:

Theorem: *Trans_R_imp_Trans_Inv_R*.
Statement: $\forall U{:}\,Set.\forall R{:}\,(Rel\ U).(Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
Proof:
Consider a set U and a R of type $(Rel\ U)$ such that $(Trans\ U\ R)$ $(trans)$; then consider three elements x, y and z of U such that $(Inv\ U\ R\ x\ y)$ (h_1) and $(Inv\ U\ R\ y\ z)$ (h_2).
Applying hypothesis *trans* to hypotheses h_2 and h_1 we get $(R\ z\ x)$.
Q.E.D.

However, we are not satisfied with this text, some information is lacking. For example, to understand this proof, it is necessary to known that the final result $(R\ z\ x)$ is equivalent to $(Inv\ U\ R\ x\ z)$. This equivalence is not expressed in the text because it is not in the λ-term. However, it can be found by studying the type expressions of the subproofs.

3 The types expressions

The type expressions we include in the text are to allow human readers to understand the development of the proof. We have not yet specified how these expressions are obtained. There exist several classical algorithms that can compute then. But, while in the *Calculus of Constructions* a λ-term has only one type, this type can be represented by different expressions. The expression computed depends on the algorithm used. All these expressions are formally equivalent, but we want to obtain the one which is easiest to understand for human readers.

3.1 Motivation for computing types

We actually do not compute one but two type expressions for each subproof. The first expression, called the *expected* type, corresponds to the proposition this subproof has to prove. This expression does not depend on the subproof itself but only on its location within the global proof. The second expression computed for a subproof is called the *constraint* type. This expression depends of the structure of the subproof. It indicates the direction the proof will follow. When the proof is correct, these two expressions are formally equivalent. This equivalence can be checked automatically, but this in no way implies that it is easy to understand for human readers.

3.2 The algorithm

The main algorithm:

The algorithm consists on a recursive top-down traversal of the λ-term that annotates each subproof with its two type expressions T_e and T_c. The algorithm has to be initialized with the expected type of the proof we want to annotate. When the proof concerns a theorem, this type is simply the statement of the theorem. The algorithm then computes the constraint type of the proof and the expected type of its subproofs. At this point it can recursively annotate the subproofs. The judgment $\Gamma, T_e \vdash M \rightsquigarrow M^{T_e}_{T_c}$ must be understood as "In an environment Γ, for an expected type T_e, the proof M must be the annotated with T_e and T_c". The following inference rules determine the way in which these annotations are computed.

$$\text{(VAR)}\ \Gamma, T_e \vdash v \rightsquigarrow v^{T_e}_{T_c}\ \texttt{if}\ (v\colon T_c) \in \Gamma$$

$$\text{(CONST)}\ \Gamma, T_e \vdash c \rightsquigarrow c^{T_e}_{T_c}\ \texttt{if}\ (c\colon T_c) \in \Gamma$$

$$\text{(LAMBDA)}\ \frac{\Gamma \vdash T_e \rhd \forall x\colon T_2.\,T_3 \qquad (x\colon T_1) + \Gamma, T_3 \vdash M \rightsquigarrow M'}{\Gamma, T_e \vdash (\lambda x\colon T_1.\,M) \rightsquigarrow (\lambda x\colon T_1.\,M')^{T_e}_{\forall x\colon T_2.\,T_3}}$$

$$\text{(APP1)}\ \frac{\begin{array}{c}\Gamma \vdash M\colon T_1 \qquad \Gamma \vdash T_1 \rhd \forall x\colon T_2.\,T_3 \\ \Gamma, \forall x\colon T_2.\,T_3 \vdash M \rightsquigarrow M' \qquad \Gamma, T_2 \vdash N \rightsquigarrow N'\end{array}}{\Gamma, T_e \vdash (M\ N) \rightsquigarrow (M'\ N')^{T_e}_{T_3\{x/N\}}}\ \begin{array}{c}\\ \\ \texttt{if}\ \Gamma \vdash T_2\colon Prop\end{array}$$

$$\text{(APP2)}\ \frac{\Gamma \vdash M\colon T_1 \qquad \Gamma \vdash T_1 \rhd \forall x\colon T_2.\,T_3 \qquad \Gamma, \forall x\colon T_2.\,T_3 \vdash M \rightsquigarrow M'}{\Gamma, T_e \vdash (M\ N) \rightsquigarrow (M'\ N)^{T_e}_{T_3\{x/N\}}}\ \begin{array}{c}\\ \\ \texttt{if not}\ \Gamma \vdash T_2\colon Prop\end{array}$$

The rule APP2 corresponds to the case where the type of N is not a proposition. In this case, N is not a proof and we do not compute annotations for it. The algorithm uses two auxiliary algorithms. The types inference algorithm computes the types of the applications' heads. The weak head normal-form algorithm is used when a type must be put in the form $\forall x\colon T_1.\,T_2$.

A simple type inference algorithm:

We have chosen the simplest type inference algorithm because it does not reduce the constants. This way, the type we obtain is quite abstract. The judgment $\Gamma \vdash M\colon T$ must be understood as "In the context Γ, the term M has type T".

$$\text{(VAR)}\ \Gamma \vdash v\colon T\ \texttt{if}\ (v\colon T) \in \Gamma$$

$$\text{(CONST)}\ \Gamma \vdash c\colon T\ \texttt{if}\ (c\colon T) \in \Gamma$$

$$\text{(APP)}\ \frac{\Gamma \vdash M\colon T_1 \qquad \Gamma \vdash T_1 \rhd \forall x\colon T_2.\,T_3}{\Gamma \vdash (M\ N)\colon T_3\{x/M\}}$$

$$\text{(LAMBDA)}\ \frac{(x\colon T_1) + \Gamma \vdash M\colon T_2}{\Gamma \vdash (\lambda x\colon T_1.\,M)\colon \forall x\colon T_1.\,T_2}$$

$$\text{(PROD)}\ \frac{(x\colon T_1) + \Gamma \vdash T_2\colon s}{\Gamma \vdash (\forall x\colon T_1.\,T_2)\colon s}$$

The calculus of weak head normal-form:

The weak head normal form of an expression has the property that if a type is the type of a λ-abstraction or the type of the head of an application, its weak head normal-form has the form $\forall x{:}T_1.T_2$. We have chosen this algorithm because it reduces a minimum number of constants and we want to keep the type expressions as abstract as possible. The algorithm we present is simpler than the classical one. It is because we only use it to reduce types into the form $\forall x{:}T_1.T_2$.

The judgment $\Gamma \vdash T_1 \triangleright T_2$ should be understood as "In a context Γ, type T_1 has the weak head normal-form T_2.

$$\text{(PROD)}\ \Gamma \vdash \forall x{:}T_1.T_2 \triangleright \forall x{:}T_1.T_2$$

$$\text{(LAMBDA)}\ \Gamma \vdash \lambda x{:}T_1.T_2 \triangleright \lambda x{:}T_1.T_2$$

$$\text{(CONST)}\ \frac{\Gamma \vdash D \triangleright M}{\Gamma \vdash c \triangleright M}\quad \texttt{if } (c{:=}\,D) \in \Gamma$$

$$\text{(BETA)}\ \frac{\Gamma \vdash M\{x/N\} \triangleright M'}{\Gamma \vdash (\lambda x{:}T.M\ N) \triangleright M'}$$

$$\text{(APP)}\ \frac{\Gamma \vdash M \triangleright M' \quad \Gamma \vdash (M'\ N) \triangleright M_1}{\Gamma \vdash (M\ N) \triangleright M_1}\quad \texttt{if not } M = \lambda x{:}T.M_2$$

Dealing with α-conversion:

With these algorithms, it is possible to apparently get an incoherence between the names of the variables used in types and the names used in λ-terms. For example, the type of the proof $\lambda x{:}T.M$ can be $\forall a{:}T.P$. This is not a formal incoherence but it can lead to some dislinking text. We prefer to use in the types the same names that are in the proof. So, if the computed type for proof $\lambda x_1{:}T_1.\cdots\lambda x_n{:}T_n.M$ is $\forall a_1{:}T_1'.\cdots\forall a_n{:}T_n'.T$, we rename it in $\forall x_1{:}T_1'.\cdots\forall x_n{:}T_n'.T$.

3.3 The complete translation

We now have a proof where each subproof is annotated by two type expressions. We can use the sentences of the previous translation briefly described in section 2. We only have to adapt them when the expected and constraint types differ. In this case, we use a sentence of the form "We have proven T_c which is equivalent to T_e." The new translation in its complete form is:

Consider a set U.
Consider a R of type $(\mathit{Rel}\ U)$.
Assume $(\mathit{Trans}\ U\ R)$ (trans).
Consider an element x of U.
Consider an element y of U.
Consider an element z of U.
Assume $(\mathit{Inv}\ U\ R\ x\ y)$ (h_1).
Assume $(\mathit{Inv}\ U\ R\ y\ z)$ (h_2).
* With hypothesis *trans* **we have** $(\mathit{Trans}\ U\ R)$ **which is equivalent to** $\forall x, y, z{:}\,U.\,(R\ x\ y) \Rightarrow \cdots$.
* With hypothesis h_2 **we have** $(\mathit{Inv}\ U\ R\ y\ z)$ **which is equivalent to** $(R\ z\ y)$.
* With hypothesis h_1 **we have** $(\mathit{Inv}\ U\ R\ x\ y)$ **which is equivalent to** $(R\ y\ x)$.
Applying the first result to the two others **we get** $(R\ z\ x)$ **which is equivalent to** $(\mathit{Inv}\ U\ R\ x\ z)$.
We have proven $(\mathit{Inv}\ U\ R\ y\ z) \Rightarrow (\mathit{Inv}\ U\ R\ x\ z)$.
We have proven $(\mathit{Inv}\ U\ R\ x\ y) \Rightarrow (\mathit{Inv}\ U\ R\ y\ z) \Rightarrow (\mathit{Inv}\ U\ R\ x\ z)$.
We have proven $\forall z{:}\,U.\,(\mathit{Inv}\ U\ R\ x\ y) \Rightarrow \cdots$.
We have proven $\forall y, z{:}\,U.\,(\mathit{Inv}\ U\ R\ x\ y) \Rightarrow \cdots$.
We have proven $\forall x, y, z{:}\,U.\,(\mathit{Inv}\ U\ R\ x\ y) \Rightarrow \cdots$ **which is equivalent to** $(\mathit{Trans}\ U\ (\mathit{Inv}\ U\ R))$.
We have proven $(\mathit{Trans}\ U\ R) \Rightarrow (\mathit{Trans}\ U\ (\mathit{Inv}\ U\ R))$.
We have proven $\forall R{:}\,(\mathit{Rel}\ U).\,(\mathit{Trans}\ U\ R) \Rightarrow (\mathit{Trans}\ U\ (\mathit{Inv}\ U\ R))$.
We have proven $\forall U{:}\,\mathit{Set}.\forall R{:}\,(\mathit{Rel}\ U).\,(\mathit{Trans}\ U\ R) \Rightarrow (\mathit{Trans}\ U\ (\mathit{Inv}\ U\ R))$.

This text is now complete. It should be compared with the text presented in section 2.3. All the information that could be useful to the reader are exposed. For example the equivalence between $(R\ z\ x)$ and $(\mathit{Inv}\ U\ R\ x\ z)$ was lacking in the first text. It is now expressed in the sentence "$\cdots$ **we get** $(R\ z\ x)$ **which is equivalent to** $(\mathit{Inv}\ U\ R\ x\ z)$".

3.4 More complex sentences

Again, the text we obtain is excessively verbose. As for the previous translation, we want to use more complex sentences. We also want to avoid giving too many type expressions. Nevertheless, we must be sure to keep all the useful information in the text.

λ-abstractions

It is not useful to translate the λ-abstractions one by one. It is often possible to introduce several variables in a single sentence. However, we cannot do this in every case because this contraction makes the types of intermediate λ-abstractions disappear. We do not describe this contraction directly on the text but on the λ-term with a conditional rewriting rule. We denote a multiple λ-abstraction with the notation $\lambda x_1{:}\,T_1; x_2{:}\,T_2.\,M$.

$$\left(\lambda x_1\!:T_1;\cdots x_i\!:T_i.\,(\lambda x_{i+1}\!:T_{i+1};\cdots x_{i+j}\!:T_{i+j}.\,M)^{T'_e}_{T'_c}\right)^{T_e}_{T_c}$$
$$\downarrow \; if\ T'_e = T'_c$$
$$(\lambda x_1\!:T_1;\cdots x_{i+j}\!:T_{i+j}.\,M)^{T_e}_{T_c}$$

This rule makes the type annotations T'_e and T'_c disappear. Nevertheless, by construction, we know how to easily derive T'_e from T_c. When T_c remains present, the disappearance of T'_e (and of T'_c if it is equal to T'_e) does not deprive the reader of information.

Applications

The application of a function to several arguments is formally decomposed into several applications each with a single argument. But it is often useful to consider this set of applications as a single application. This is put in concrete form with the usual rule that allows to denote the applications $((f\ a_1)\ a_2)$ with the notation $(f\ a_1\ a_2)$. As with λ-abstractions, this rule makes the type of the intermediate application disappear. So, we use a conditional rewriting rule.

$$\left((M\ a_1\cdots a_i)^{T'_e}_{T'_c}\ a_{i+1}\cdots a_{i+j}\right)^{T_e}_{T_c}$$
$$\downarrow \; if\ T'_e = T'_c$$
$$(M\ a_1\cdots a_{i+j})^{T_e}_{T_c}$$

By construction, T_c is the specialization of T'_e to the arguments $a_{i+1}\cdots a_{i+j}$. With this condition, it is not useful for a human reader, to keep T'_e (or T'_c if it is equal to T'_e) when T_c remains present in the text.

Variables and constants

When all the subproofs of an application are variables or constants, it is more natural to produce a unique sentence which translates at the same time the application and all of its subproofs. We use this rule with the condition that, for each subproof, the expected type must be equal to the constraint type. Another rule, used to reduce the verbosity of the text, is to omit the constraint type of variables or constants because these types are immediately accessible.

Our example

The text after the contractions is:

Theorem: *Trans_R_imp_Trans_Inv_R.*
Statement: $\forall U{:}\,Set.\forall R{:}\,(Rel\ U).\,(Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
Proof:
Consider a set U and a R of type $(Rel\ U)$ such that
$(Trans\ U\ R)$ $(trans)$.
  Consider three elements x, y and z of U such that $(Inv\ U\ R\ x\ y)$ (h_1)
  and $(Inv\ U\ R\ y\ z)$ (h_2).
    * With hypothesis $trans$ we get $\forall x,y,z{:}\,U.\,(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)$
    * With hypothesis h_2 we get $(R\ z\ y)$.
    * With hypothesis h_1 we get $(R\ y\ x)$.
    Applying the first result to the two others we get $(R\ z\ x)$ which is
    equivalent to $(Inv\ U\ R\ x\ z)$.
  We have proven $\forall x,y,z{:}\,U.\,(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (Inv\ U\ R\ x\ z)$
  which is equivalent to $(Trans\ U\ (Inv\ U\ R))$.
Q.E.D.

This text should be compared with the text obtained in section 2.4. This text represents the totality of the proof. It is very precisee. A human reader doesn't need so many details. Next, we look into suppressing information that the human reader deems useless.

3.5 The *well-known* simple definitions

To obtain shorter text, we want to suppress some information. This information will be type information. More precisely it will be information on equivalence between expected and constraint types.
We want allow the rules of contraction of λ-terms defined in section 3.4 to permit more type annotations to disappear. To do this, we change the conditions $T'_e = T'_c$ of these rules by conditions $T'_e \approx T'_c$. This new notation means that T'_e and T'_c does not need to be equal but simply *close*.

It is simply a restriction of the definition of equivalence. To determine if two propositions are equivalent, we compute their normal form. This normal form is obtained by a complete β-reduction (by computing the results of all applications of functions) and by a complete δ-reduction (by substituting all constants by they definitions). Then we check if both normal forms are α-convertible (equal after an adequate renaming of binded variables).

The definition of closeness is parameterized by a list of constants. This list contains the constants that the reader considers as *well-known*. To determine if two expressions are *close*, we compute some weak normal form. This weak normal form is obtained by a complete β-reduction and a partial δ-reduction. We reduce only the *well-known* constants. Then we also check if the two weak normal forms are α-convertible. We have defined an initial list of well-known constants, but the reader can modify this list to adapt the translation to his knowledge.

In our example, if the reader adds the constant *Trans* to this list, he will obtain a shorter text. This text will not explain the proof steps concerning definition of *Trans*. However, he will continue to explain the ones concerning definition of *Inv*:

Theorem: *Trans_R_imp_Trans_Inv_R.*
Statement: $\forall U{:}\,Set.\forall R{:}\,(Rel\ U).\,(Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
Proof:
Consider a set U and a R of type $(Rel\ U)$ such that $(Trans\ U\ R)$ $(trans)$; and three elements x, y and z of U such that
$(Inv\ U\ R\ x\ y)$ (h_1) and $(Inv\ U\ R\ y\ z)$ (h_2).
* With hypothesis h_2 we get $(R\ z\ y)$.
* With hypothesis h_1 we get $(R\ y\ x)$.
Applying hypothesis *trans* to these two results we get $(R\ z\ x)$ which is equivalent to $(Inv\ U\ R\ x\ z)$.
Q.E.D.

4 The *inductive* extension of *Calculus of Constructions*

In this section, we describe briefly the extension of the *Calculus of Constructions* known as the *Calculus of Inductive Constructions*. Then, we present the adaptation of our translation system to this new formalism.

4.1 Inductive definitions

Inductive definitions have been added to the *Calculus of Constructions* to increase its expressivity. They allow one to define types by constructors and allow to introduce induction. For example, in the *Calculus of Inductive Constructions* the disjunction is not a primitive of the language. It is inductively defined for any arbitrary propositions A and B. The notation used to define it in the *Coq* system is:

$$\begin{array}{l}\text{Inductive } or\ [A,B{:}\,Prop] : Prop := \\ \quad or_{intro}l{:}\,A \Rightarrow (or\ A\ B) \\ \quad |\ or_{intro}r{:}\,B \Rightarrow (or\ A\ B).\end{array}$$

This definition defines an inductive constant $or{:}\,Prop \rightarrow Prop \rightarrow Prop$ used to build propositions. It also defines two constructors:
$or_{intro}l{:}\,\forall A,B{:}\,Prop.\,A \Rightarrow (A \vee B)$ and $or_{intro}r{:}\,\forall A,B{:}\,Prop.\,B \Rightarrow (A \vee B)$, which allow one to *introduce* this constant. It also defines an *elimination* rule of this constant. All the uses of an inductive definition are made by an *introduction* with one of its constructors, or by an *elimination* with the `Elim` operator. They are always explicit in the λ-term.

4.2 Adaptation of the translation

The sentences used for introducing an inductive constant are very closed to the ones used for applying a hypothesis. They have the form:

$$(or_{intro}l\ A\ B\ M) \leadsto \boxed{\begin{array}{l} *\ \text{«}M\text{» we have } A. \\ \text{So, with constructor } or_{intro}l \text{ we have } A \vee B. \end{array}}$$

For the elimination of inductive constants we defined other sentences that are used to bring out the structure of a proof by cases. The elimination of a disjunction will give the text:

$$\begin{array}{l} <P>\ \texttt{Elim}\ M\ \texttt{with} \\ \quad \lambda h{:}\, A.\, M_A \\ \quad \lambda h{:}\, B.\, M_B \\ \quad \text{end} \end{array} \leadsto \boxed{\begin{array}{l} \text{«}M{:}\, A \vee B\text{»} \\ \text{So by definition of } or \text{ we have two cases :} \\ \text{Case 1.:} \\ \quad \text{We have } A\ (h_1). \\ \quad\quad \text{«}M_A\text{»} \\ \text{Case 2.:} \\ \quad \text{We have } B\ (h_1). \\ \quad\quad \text{«}M_B\text{»} \\ \text{So we have proven } P. \end{array}}$$

We have also defined some sentences for the inductive constants with either no constructors or one constructor.

4.3 Adaptation of type calculus

Concerning the algorithm that computes type annotations, the eliminations behave in a way similar to applications. To compute the expected types of the arguments of an application, it is necessary to compute a type expression of the application's head and reduce it to a weak head normal-form. To compute the expected types of an elimination's subproofs, it is necessary to compute a type expression of the eliminated term and reduce it to a weak normal form.

The constructors have reactions very similar to the constants. We can extend the algorithms defined in section 3.2, by adding some inference rules.

The main algorithm:

$$\text{(ELIM1)}\ \frac{\begin{array}{c} \Gamma \vdash M{:}T_1 \qquad \Gamma \vdash T_1 \rhd T_2 \qquad \Gamma, T_2 \vdash M \leadsto M' \\ \Gamma, F(\Gamma, T_2, T_c, i) \vdash N_i \leadsto N'_i \ \ \texttt{for}\ i \in \{1 \cdots n\} \end{array}}{\Gamma, T_e \vdash \begin{pmatrix} <T_c>\ \texttt{Elim}\ M\ \texttt{with} \\ N_1 \cdots N_n \\ \text{end} \end{pmatrix} \leadsto \begin{pmatrix} <T_c>\ \texttt{Elim}\ M'\ \texttt{with} \\ N'_1 \cdots N'_n \\ \text{end} \end{pmatrix}^{T_e}_{T_c}}$$

$$\texttt{if}\ \Gamma \vdash T_2{:}\, Prop$$

$$\text{(ELIM2)}\ \frac{\Gamma \vdash M{:}T_1 \qquad \Gamma \vdash T_1 \rhd T_2 \qquad \Gamma, F(\Gamma, T_2, T_c, i) \vdash N_i \rightsquigarrow N'_i \ \ \texttt{for}\ i \in \{1 \cdots n\}}{\Gamma, T_e \vdash \begin{pmatrix} < T_c > \texttt{Elim}\ M\ \texttt{with} \\ N_1 \cdots N_n \\ \texttt{end} \end{pmatrix}^{T_e} \rightsquigarrow \begin{pmatrix} < T_c > \texttt{Elim}\ M'\ \texttt{with} \\ N'_1 \cdots N'_n \\ \texttt{end} \end{pmatrix}_{T_c}} \ \texttt{if not}\ \Gamma \vdash T_2{:}Prop$$

The rule *ELIM*2 corresponds to the case where M is not a proof. The notation $F(\Gamma, T_2, T_c, i)$ represents the function of the *Calculus of Inductive Constructions* that computes the type of i^{th} case of an elimination depending on the type of the eliminated term and the constraint type of the elimination.

The inference type algorithm:

$$\text{(IND CONSTANT)}\ \Gamma \vdash C{:}T\ \texttt{if}\ (C{:}T) \in \Gamma$$

$$\text{(IND CONSTRUCT)}\ \Gamma \vdash C_{intro}i{:}T\ \texttt{if}\ (C_{intro}i{:}T) \in \Gamma$$

$$\text{(ELIM)}\ \Gamma \vdash \begin{matrix} < T > \texttt{Elim}\ M\ \texttt{with} \\ N_1 \cdots N_n \\ \texttt{end} \end{matrix} \ {:}T$$

The weak head normal form algorithm:

The weak head normal-form we compute for the expected type of eliminated terms are $(C\ a_1 \cdots a_p)$ where C is the inductive constant and where $a_1 \cdots a_p$ are its parameters.

$$\text{(IOTA)}\ \frac{\Gamma \vdash (N_i\ a_{p+1} \cdots\ a_m) \rhd M_1}{\Gamma \vdash \begin{pmatrix} < T > \texttt{Elim}\ (C_{intro}i\ a_1 \cdots\ a_m)\ \texttt{with} \\ N_1 \cdots N_n \\ \texttt{end} \end{pmatrix} \rhd M_1}\ \texttt{if}\ \Gamma \vdash (C_{intro}i\ a_1 \cdots\ a_m){:}(C\ a'_1 \cdots a'_p)$$

$$\text{(ELIM)}\ \frac{\Gamma \vdash M \rhd M_1 \qquad \Gamma \vdash \begin{pmatrix} < T > \texttt{Elim}\ M_1\ \texttt{with} \\ N_1 \cdots N_n \\ \texttt{end} \end{pmatrix} \rhd M_2}{\Gamma \vdash \begin{pmatrix} < T > \texttt{Elim}\ M\ \texttt{with} \\ N_1 \cdots N_n \\ \texttt{end} \end{pmatrix} \rhd M_2}\ \texttt{if not}\ M = (C_{intro}i\ a_1 \cdots\ a_m)$$

4.4 The adaptation of our example

In our example, to define the constants *Trans* and *Inv*, instead of simple definitions, we could have use inductive definitions with one constructor.

Inductive $Trans\ [U{:}\,Set;\ R{:}\,(Rel\ U)] : Prop :=$
$Trans_{intro}{:}\,(\forall x, y, z{:}\,U.\,(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)) \Rightarrow (Trans\ U\ R).$

Inductive $Inv\ [U{:}\,Set;\ R{:}\,(Rel\ U);\ x, y{:}\,U] : Prop :=$
$Inv_{intro}{:}\,(R\ y\ x) \Rightarrow (Inv\ U\ R\ x\ y).$

A formal proof of our assertion with these new definitions is:

```
$\lambda U\colon Set.\,\lambda R\colon (Rel\ U).\,\lambda trans\colon (Trans\ U\ R).$
  $<(Trans\ U\ (Inv\ U\ R))>$ Elim $trans$ with
    $\lambda trans'\colon \forall x,y,z\colon U.\,(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z).$
      $(Trans_{intro}\ U\ (Inv\ U\ R)$
        $\lambda x,y,z\colon U.\,\lambda h_1\colon (Inv\ U\ R\ x\ y).\,\lambda h_2\colon (Inv\ U\ R\ y\ z).$
          $(Inv_{intro}\ U\ R\ x\ z$
            $(trans'\ z\ y\ x$
              $<(R\ z\ y)>$ Elim $h_2$ with $\lambda h_2'\colon (R\ z\ y).\,h_2'$ end
                $<(R\ y\ x)>$ Elim $h_1$ with $\lambda h_1'\colon (R\ y\ x).\,h_1'$ end)))
    end
```

This proof is bigger than the previous one. It is because all uses of the definitions of $Trans$ and Inv are done explicitly with introductions (application of a constructor C_{intro}) and by eliminations (use of operator Elim). The text we obtain is:

Theorem: $Trans\_R\_imp\_Trans\_Inv\_R$.
Statement: $\forall U\colon Set.\forall R\colon (Rel\ U).\,(Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
Proof:
Consider a set U and a R of type $(Rel\ U)$ such that
$(Trans\ U\ R)$ $(trans)$.
We use definition of $Trans$ with hypothesis $trans$.
We have $\forall x,y,z\colon U.\,(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)$ $(trans')$.
Consider three elements x, y and z of U such that $(Inv\ U\ R\ x\ y)$ (h_1)
and $(Inv\ U\ R\ y\ z)$ (h_2).
  * Using definition of Inv with hypothesis h_2 we get $(R\ z\ y)$
  * Using definition of Inv with hypothesis h_1 we get $(R\ y\ x)$
  Applying hypothesis $trans'$ to these two results we get $(R\ z\ x)$
  So, applying definition of Inv, we get $(Inv\ U\ R\ x\ z)$.
We have proven $\forall x,y,z\colon U.\,(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (Inv\ U\ R\ x\ z)$.
So, using definition of $Trans$ we have $(Trans\ U\ (Inv\ U\ R))$.
Q.E.D.

This text is similar to the text of section 3.4 although $Trans$ and Inv are defined in a different way.

5 The *well-known* inductive definitions

As for simple definitions, we are now going to define a list of *well-known* inductive definitions. All the transformations we define in this section are parameterized by this list.

We had descrided in section 3.5, how to hide the use of simple definitions by suppressing type annotations. But, we cannot adapt this method to inductive definitions. Each use of an inductive definition (elimination or introduction) appears explicitly in the proof structure. To hide these proof steps, we modify directly the λ-term before producing the text.

These transformations produce λ-terms which may not be well-typed. These λ-terms can no longer be checked by the type checkers of the *Calculus of Inductive Constructions.* This is due to the fact that they represent incomplete proofs. If we want to validate the λ-terms, which we obtain after the transformations, it will be necessary to reconstruct a complete proof.

5.1 Omitting an introduction

The introductions of an inductive constant are done by applying a constructor. When the definition of the introduced constant is *well-known,* and when this application has a unique subproof, it is possible to mingle the application and its subproof. In the λ-term we syntactically substitute the application by its subproof. We use the rewriting rule:

$$(C_{intro}i\ a_1 \cdots a_p\ N^{T_{e2}}_{T_{c2}})^{T_{e1}}_{T_{c1}} \longmapsto N^{T_e=T_{e1}}_{T_c=T_{c2}}$$

For example, we can consider a proof with an *or* introduction. In this proof h as type A.

$$(C_{intro}l\ A\ B\ h^A_A)^{A\vee B}_{A\vee B} \qquad \longmapsto \qquad h^{A\vee B}_A$$

$$\boxed{\begin{array}{l}\text{With hypothesis } h \text{ we have } A.\\ \text{So, by definition of } \textit{or} \text{ we get } A\vee B.\end{array}} \longmapsto \boxed{\text{With hypothesis } h \text{ we have } A\vee B.}$$

5.2 Omitting an elimination

The rule we are going to define concerns uniquely the elimination of a variable that is introduced by a λ-abstraction and then immediately eliminated by an `Elim` operator. We use this rule if the type of the variable is a well-known constant with exactly one constructor. This is like a proof by cases with one unique case.

$$\left(\lambda x{:}T.\ < P >\ \texttt{Elim}\ x^{T_{e3}}_{T_{c3}}\ \texttt{with}\ N^{T_{e2}}_{T_{c2}}\texttt{end}\right)^{T_{e1}}_{T_{c1}} \longmapsto N^{T_e=T_{e1}}_{T_c=T_{c2}}\ \texttt{if}\ \begin{array}{l}T_{e3}\approx T_{c3}\ \text{and}\\ \texttt{not}\ x\in \mathit{Free}(N)\end{array}$$

The condition $T_{e3}\approx T_{c3}$ limits this transformation to situations where the reader can easily realize that the type of x is the well-known inductive constant. The condition not $x\in \mathit{Free}(N)$ guarantees that the variable x is not used in another part of the proof.

As an example, we can see the effect of this transformation on a proof using an *and* elimination:

$$\left(\begin{array}{l}\lambda h_1{:}A\wedge B.\\ \quad < P >\ \texttt{Elim}\ h_1\ \texttt{with}\\ \quad \left(\begin{array}{l}\lambda h_2{:}A.\,\lambda h_3{:}B.\\ \quad M\end{array}\right)^{A\Rightarrow B\Rightarrow P}_{A\Rightarrow B\Rightarrow P}\\ \quad \texttt{end}\end{array}\right)^{A\wedge B\Rightarrow P}_{A\wedge B\Rightarrow P} \longmapsto \left(\begin{array}{l}\lambda h_2{:}A.\,\lambda h_3{:}B.\\ \quad P\end{array}\right)^{A\wedge B\Rightarrow P}_{A\Rightarrow B\Rightarrow P}$$

Assume $A \wedge B$ (h_1). We use h_1 with definition of *and*. We have A (h_2) and B (h_3). «M: P» So, we have proven $A \Rightarrow B \Rightarrow P$. So, we have proven $A \wedge B \Rightarrow P$.	$\longmapsto$	Assume A (h_2) and B (h_3). «M: P» So, we have proven $A \wedge B \Rightarrow P$.

5.3 Choosing type expressions

For these two transformations, we substitute a term M with types T_{e1} and T_{c1}, by a subterm N with types T_{e2} and T_{c2}. We have to specify the type annotations T_e and T_c of the resulting term.

The expected type of a proof is defined by its location within the global proof. It corresponds to a proof obligation. we choose $T_e := T_{e1}$ as expected type of resulting term. The constraint of a proof describes what the proof actually shows. It depends on the structure of the proof. We choose $T_c := T_{c2}$ as constraint type for the resulting proof.

Initially the constraint and expected types of a proof are equivalent. But, after one of these transformations the two type annotations of the resulting proof are not formally equivalent. The previous definition of closeness is no more adapted. So, before the transformations we annotate each subproof with the closness of its type annotation. Then, after a transformation, we decide that $T_e := T_{e1}$ and $T_c := T_{c2}$ should be considered as closed if both T_{e1} and T_{c1}, and T_{e2} and T_{c2} are closed.

5.4 Our example

If the inductive definition of *Trans* is considered as well-known, our example becomes:

Theorem: *Trans_R_imp_Trans_Inv_R.*
Statement: $\forall U: Set. \forall R: (Rel\ U). (Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
Proof:
Consider a set U and a R of type ($Rel\ U$) such that
$\forall x, y, z: U. (R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)$ ($trans'$) and consider three elements x, y and z of U such that ($Inv\ U\ R\ x\ y$) (h_1) and ($Inv\ U\ R\ y\ z$) (h_2).
  * Using definition of *Inv* with hypothesis h_2 we get ($R\ z\ y$)
  * Using definition of *Inv* with hypothesis h_1 we get ($R\ y\ x$)
  Applying hypothesis $trans'$ to these two results we get ($R\ z\ x$)
  So, applying definition of *Inv*, we get ($Inv\ U\ R\ x\ z$).
Q.E.D.

This text is very close to the text obtained with simple definitions in section 3.5. Its formulation is a little different, but its structure and its detail level are the same. We keep this text as the final translation.

6 Other transformations of the λ-term

There is no reason to limit the transformations of the λ-term to these simple cases. While an abusive use of transformations can prevent the reconstruction of the complete proof and simply make the text incomprehensible, there are other applications for such transformations. In proof systems, the associative operators are often inductively defined with a binary representation that does not take advantage of associativity. This representation forces one to use successive introductions or eliminations. Transformations of the λ-term can group all of these proof steps into one.

Consider the logical operators of conjunction and disjunction. An elimination of the proposition $A \vee (B \vee C)$ must be done in two steps with two successive `Elim` operators. We transform the proof to use only one elimination:

```
$< P >$ Elim $M_1^{T_e=A\vee(B\vee C)}$ with
  $\lambda h_1 : A.\, M_2^{T_e=P}$
  $\lambda h_2 : B \vee C.$
    $< P >$ Elim $h_2$ with
      $\lambda h_3 : B.\, M_3^{T_e=P}$
      $\lambda h_4 : C.\, M_4^{T_e=P}$
    end
end
```

$\longmapsto$

```
$< P >$ Elim $M_1^{T_e=A\vee(B\vee C)}$ with
  $\lambda h_1 : A.\, M_2^{T_e=P}$
  $\lambda h_3 : B.\, M_3^{T_e=P}$
  $\lambda h_4 : C.\, M_4^{T_e=P}$
end
```

```
«$M_1^{T_e=A\vee(B\vee C)}$»
By definition of or have two cases.
Case 1.:
  We have $A$ ($h_1$).
    «$M_2^{T_e=P}$»
Case 2.:
  We have $B \vee C$ ($h_2$).
    Using $h_2$, we have $B \vee C$.
    By definition of or have two cases.
    Case 2.1.:
    We have $B$ ($h_3$).
      «$M_3^{T_e=P}$»
    Case 2.2.:
    We have $C$ ($h_4$).
      «$M_4^{T_e=P}$»
So, we have proven $P$.
```

$\longmapsto$

```
«$M_1^{T_e=A\vee(B\vee C)}$»
By definition of or have three cases.
Case 1.:
  We have $A$ ($h_1$).
    «$M_2^{T_e=P}$»
Case 2.:
  We have $B$ ($h_3$).
    «$M_3^{T_e=P}$»
Case 3.:
  We have $C$ ($h_4$).
    «$M_4^{T_e=P}$»
So, we have proven $P$.
```

The dual case is the introduction of the conjunction operator. The introduction of $A \wedge (B \wedge C)$ has to be done with two applications of constructor and_{intro}. We transform the λ-term to get only one application:

$$(and_{intro}\ A\ (B \wedge C)\ M_1\ (and_{intro}\ B\ C\ M_2\ M_3)) \longmapsto (and_{intro}\ A\ B\ C\ M_1\ M_2\ M_3)$$

* «$M_1^{T_e=A}$» * «$M_2^{T_e=B}$» * «$M_3^{T_e=C}$» * By definition of *and* we have $B \wedge C$. By definition of *and* we have $A \wedge (B \wedge C)$.	$\longmapsto$	* «$M_1^{T_e=A}$» * «$M_2^{T_e=B}$» * «$M_3^{T_e=C}$» By definition of *and* we have $A \wedge (B \wedge C)$.

7 Conclusion and future work

This work has been implemented and it is now distributed with the *Coq* system[2]. For future work, we still want to improve the quality of the translation. We also want to give to the user more ways to personalize the text production and to adapt it to the mathematical theory he is interested in. We want to develop the proof transformations done before the verbalization. We want to define a set of correct proof transformation and find a strategy to use them. A interesting application of these transformations can be to produce of text that does not really explain the proofs but just give the main ideas.

A possible extension can be to translate also the logic formulæ into english[1][6]. We are also studying the possibility of using natural language not only for output but also for input[5] or for formal representation of proofs[6]. This means that we must be able to generate a text from a λ-term and then to reconstruct the λ-term from this text.

References

1. D. Chester, *The translation of Formal Proofs into English*, Artificial Intelligence 7 (1976), 261-278, 1976.
2. C. Cornes, J. Courant, J-C. Filliâtre, E. Gimenez, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner, *The Coq Proof Assistant Reference Manual*, INRIA.
3. Y. Coscoy, G. Kahn, L. Théry, *Extracting Text from Proofs*, in *Typed Lambda Calculus and Applications*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag.
4. X. Huang, *Human Oriented Proof Presentation: A Reconstructive Approach*, SEKI Report SR-SR-94-07, UNIVERSITÄT DES SAARLANDES,.
5. K. Prazmowski, P.Rudnicki *Mizar-MSE Primer* Warsaw University, University of Alberta.
6. A. Ranta, *Type Theory and the Informal Language of Mathematics*, in Proceedings of the 1993 Types Worshop, Nijmegen, LNCS 806, 1994.