Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/175979

Please be advised that this information was generated on 2019-11-12 and may be subject to change.

Technische Universiteit
Eindhoven
University of Technology
Where innovation starts

An interactive viewer for mathematical content based on type theory

Oostdijk, M.D.

Published: 01/01/2000

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):
Oostdijk, M. D. (2000). An interactive viewer for mathematical content based on type theory. (Computing science reports; Vol. 0015). Eindhoven: Technische Universiteit Eindhoven.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 25. Sep. 2017

Technische Universiteit Eindhoven
Department of Mathematics and Computing Science

An Interactive Viewer for Mathematical Content based on Type Theory

by

M. Oostdijk

00/15

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 00/15
Eindhoven, September 2000

An Interactive Viewer for Mathematical Content based on Type Theory

Martijn Oostdijk

September 11, 2000

Abstract

This report describes the CoqViewer tool, a system for presenting mathematical content encoded in typed lambda calculus. With mathematical content we mean mathematical theories including assumptions, definitions, theorems, and proofs. The presentation of the content is interactive and resembles informal mathematical documents, without giving up the formal nature which allows symbolic manipulation. This is possible because the presentation and formal content are kept separate. The tool takes as input a context developed in the Coq theorem prover. It then displays the context in interactive views resembling informal mathematics. The views allow the user to change certain aspects of the presentation. The resulting presentation can be exported in the OpenMath Document format.

1 Introduction

Traditionally computers are useful for mathematicians in three ways. First, in type-setting mathematical texts, computers are used since they allow easy editing of structured formulas. Second, computers take care of many computations needed by mathematicians, both concrete "number crunching" and symbolic computations. Third, more recently, computers are also employed to assist mathematicians in developing and proving theorems.

There are differences in how these uses represent the mathematical content. If one implements a system for correctly type-setting mathematical formulas, such as the LaTeX system [Lam94], it suffices to store only superficial presentation information. For example, type-setting the expression $(x^y \cdot x^z)$ requires knowledge within the system about bounding boxes and baselines, but not about the meaning of the symbols or even their arity or types.

More structure is needed when one wants to implement symbolic manipulation systems such as computer algebra systems. Consider for example a computer algebra system transforming the expression from the above example $(x^y \cdot x^z)$ into $x^{(y+z)}$. In order to perform this operation there is no need to know the exact semantics of the symbols, except for the rewrite rule that is applied here. It is, however, important to know the exact syntactic structure of the expression. Internally the manipulations take place on a tree-like data-structure which captures the syntactic structure and which can be pretty-printed in a more appealing format. Note that there is a distinction between content and presentation here.

Still more structure is needed in theorem proving systems. Not only the exact syntactic structure of expressions, but also some semantical properties need to be specified before one can prove anything about an object. In many theorem provers this is achieved by allowing the user to completely define objects in the logical language of the system. This has the advantage that these systems are very general and able to deal with any mathematical theory. It also means that the presentation stays very close to the mathematical content. This is a drawback of many theorem provers, the content is not presented well to the end user.

The CoqViewer tool described in this paper is an attempt to interactively present formal mathematical content as used in theorem provers. To be more specific, formalizations developed in the the type theoretical theorem prover Coq [BBC+99] are used as input for the tool and views are created displaying the individual elements of the development. The views present the content in a way that resembles informal mathematics. The reader can interact with the views for example by changing the level of detail of the displayed proofs. The views can access the formal mathematical content so that in principle it can be exported to computational engines and be verified or manipulated.

Since the tool is still work in progress, what is described here is the core functionality needed for the presentation. Some elementary activities for changing the presentation are possible but the tool is not an editor for mathematical content yet. The implementation language is Java [GJS96], which makes it easy to create graphical user interfaces and to reuse the code in other systems.

The next section introduces some of the main ideas behind theorem proving systems based on type theory. We elaborate on the differences between the three mathematical languages used in type theoretical theorem proving: The tactics language used to communicate with such systems, the language of formal objects used inside the theorem prover engine, and the informal natural language used by mathematicians. Section 3 presents a high level overview of the architecture of the tool we are building. Sections 4, 5, and 6 provide more details on the design and implementation of the tool. Section 7 sums up the main results and draws some conclusions from them.

2 Type Theoretic Theorem Proving

This section describes theorem proving based on type theory. It introduces some of the basic concepts and gives examples. All examples in this section are in Coq notation.

An important idea connected to type theoretical theorem proving is the notion of *context*. A context is a list of items that are either *assumptions* or *definitions*. An assumption introduces a name N and states a type τ for the new symbol. It is of the form:

$$N : \tau$$

A definition also states a defining lambda term T for the symbol. It is of the form:

$$N := T : \tau$$

Here is an example of a context that assumes a set A and a binary relation R on A, defines the notions of reflexivity, symmetry and transitivity, and assumes

that R is transitive. The example is in Coq notations, which denotes lambda-abstracted variables between square brackets and Pi-abstracted variables (a Pi-abstraction is a generalized Cartesian product) between parentheses.

```
A: Set
R: A->A->Prop
isReflexive =
   [T:A->A->Prop](x:A)(T x x):
   (A->A->Prop)->Prop
isSymmetric =
   [T:A->A->Prop](x,y:A)(T x y)->(T y x):
   (A->A->Prop)->Prop
isTransitive =
   [T:A->A->Prop](x,y,z:A)(T x y)->(T y z)->(T x z):
   (A->A->Prop)->Prop
R_trans:
   (isTransitive R)
```

So, a context consists of a list of names with a type and optionally a defining term of that type. The fact that a term is typeable means that it is meaningful in a mathematical sense.

The items in the context represent the usual elements one finds in a mathematical document, i.e. assumptions, definitions, axioms, theorems, and proofs. Theorems and their proofs are encoded as definitions, where the term represents the proof and its type the statement of the theorem. The underlying principle is called the Curry-Howard-DeBruijn correspondence of propositions as types [How80]. Lambda terms representing proofs are called *proof-objects*. As an extension to the context from the above example, consider the definition of *Leibniz' equality*, and theorems with proof-objects that prove the relation reflexive, symmetric, and transitive. Readers unfamiliar with typed lambda calculus should not try to decode the proof-objects. Instead, note that theorems are treated exactly like definitions.

```
leibniz =
   [x,y:A](P:(A->Prop))(P x)->(P y)
   : A->A->Prop
leibniz_refl =
   [x:A; P:(A->Prop); H:(P x)]H
   : (isReflexive leibniz)
leibniz_sym =
   [x,y:A; H:((P:(A->Prop))(P x)->(P y)); P:(A->Prop)]
      (H [z:A](P z)->(P x) [H2:(P x)]H2)
   : (isSymmetric leibniz)
leibniz_trans =
   [x,y,z:A; H:((P:(A->Prop))(P x)->(P y));
      H0:((P:(A->Prop))(P y)->(P z));
      P:(A->Prop); H1:(P x)]
      (H0 [a:A](P a) (H [a:A](P a) H1))
   : (isTransitive leibniz)
```

The fact that mathematical objects and propositions about those objects can be dealt with in a similar way, makes theorem proving based on type theory

so attractive. The kernel of the theorem prover engine can remain small, since all that needs to be done is type checking within a relatively small formal typing system. A small kernel means that it is trustworthy. That is, if one does not trust the results a type theoretical theorem prover produces, one can easily implement an independent type checker to check the proof-objects. This principle is called the *DeBruijn criterion* in [BB97].

The drawback of the DeBruijn criterion is that proof-objects tend to be very large and are hence difficult to read and construct. Proof-objects can be translated to natural language to make them more readable [CKT95]. To assist the human mathematician in constructing proof-objects, a language of *tactics* is used. Instead of requiring that the user inputs concrete lambda terms, the theorem prover builds the proof-objects according to high level tactics scripts given by the user.

The tactics language constructs are inspired by informal top-down mathematical proofs. For example, look at the following script which generates a proof-object for the statement that Leibniz-equality is transitive:

```
Lemma leibniz_trans: (isTransitive leibniz).
Proof.
   Unfold isTransitive leibniz.
   Intros x y z H H0.
   Apply H0.
   Apply H.
   Assumption.
Qed.
```

In fact, most users of type theoretical theorem provers consider tactics scripts, instead of proof-objects, to be the real proofs.

There are some disadvantages to using tactics scripts to represent proofs. One drawback of is that tactics scripts represents only the user's half of a dialog. Starting with the theorem to be proved as initial goal, the user guides the system, replacing the current goal by new goals, until it can be reduced to some known tautology or assumption. The output of the system during such an interactive session is not recorded in the script.

Another, related, drawback of tactics scripts is that it is hard to reconstruct intermediate goals. A normal top-down proof provides the reader with hints about what is the current statement that is proved. This can be compared to the notation used to report on chess games. All moves are stated, but from time to time also a diagram is printed to indicate the current state of the board. The diagram is helpful for the reader but is not necessary to reconstruct the game. Both tactics scripts and proof-objects lack intermediate goals. However, it is fairly easy to retrieve from a proof-object the intermediate goals, all that is needed is the type checking kernel of the theorem prover which is relatively small by the DeBruijn criterion. Reconstructing the intermediate goals from a tactics script is much harder, access to the complete theorem prover is needed.

This is why the tool described in this paper will represent proofs with proof-objects and not tactics scripts. However, the other option is also possible, see [HMBC99].

3 Overview of the System

This section gives a high level overview of the the CoqViewer presentation tool. Figure 1 shows the architecture of the system. A Coq context is parsed, which

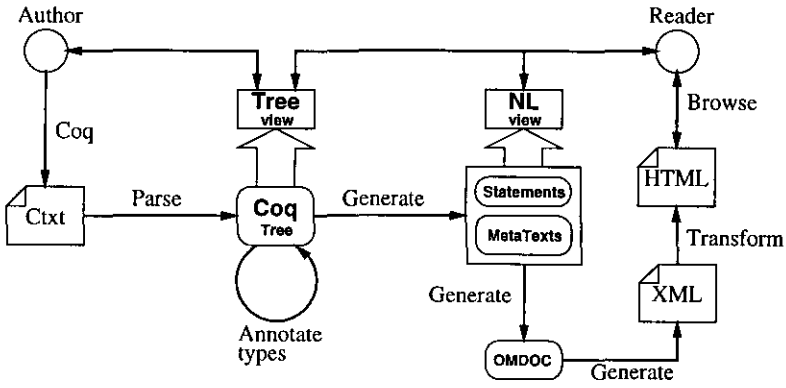


Figure 1: Architecture of the system.

results in an instance of the CoqTree data-type. This data-type is described in detail in Section 4. Next, the tree is annotated with type information using the algorithm in Section 5.8. Type inference requires many more operations on CoqTree. These operations are described in Section 5. Each node in the tree is annotated with the type of the term beneath that node. This type information is used by the *views* which present the CoqTree to the end user. Some of the views try to verbalize statements and proofs in the context. This is done by first translating the CoqTree versions of the proof-object to instances of the MetaText data-type. A MetaText consists of Statements. This data-type is described in Section 6.2. The verbalization to natural language is defined on this intermediate level so that the methods can be reused in multiple views. Currently there are two views available. The TreeView, see Figure 2, is the most basic view and shows the structure of the individual lambda terms. The NLView, see Figure 3, renders proofs as Fitch style natural language proofs. In this view assumptions are indicated by displaying them inside flags with the flag pole showing the scope of the assumption. Another possibility for displaying the content is exporting it to an OMDOC document [Koh99], which can then be viewed in an appropriate browser. More details on the views can be found in Section 6.

The creation and presentation of interactive mathematical documents using the CoqViewer, as we envision it, consists of three phases. First, during the formalization phase, a formal context is built using Coq. Next, after loading this context into the tool, during the authoring phase, the author can add presentation information to it. Currently this is done in the TreeView. Finally, during the presentation phase, the context is presented to the reader through one of the views. The reader can then interact with the document, by browsing through it and for example changing the level of detail in some of the proofs. If the reader has opened multiple views on the same object, the results from interaction are made visible consistently across all views.

3.1 Authoring

During the authoring phase, the formal objects stored in the CoqTree are altered. However, the author is not allowed to change the mathematical structure of the formal objects, only the way they will be presented to the reader. By authoring we mean refining the presentation and not the mathematical content.

Adaptation of presentation is implemented by extending the CoqTree datatype with attributes containing presentation information. Currently only the TreeView can be used to add presentation information to CoqTree objects. This view presents to the author a tree-like representation of the document, comparable to the folder tree a file browser provides. Figure 2 presents the TreeView. The panel on the left shows the context. The panel on the right shows the context item that is selected. This view is described in more detail in Section 6.1. In this view, the author is allowed to change the following information in each

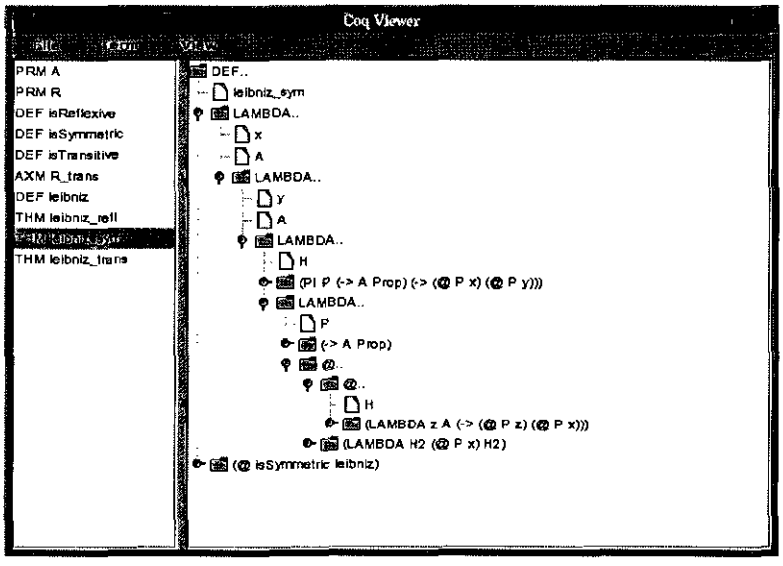


Figure 2: Screenshot of the Tree View.

node:

- If the node is a variable (variables are the leaf nodes), the author can change its name. The variable name will be changed consistently throughout the document. All variables are bound, since the input is a complete theory development.
- Each node which is not a variable can be either collapsed or expanded. This will affect the way this node is presented in the views. It depends on the view how this is done, but in general collapsed nodes are displayed concisely e.g. the translation will not recursively translate the subtrees of a collapsed node, expanded nodes are translated verbosely.
- Nodes representing certain mathematical objects can have a preferred view. For example the author might assign the natural language view as default view for proof-objects.

More attributes might be added in the future. Perhaps even an extension mechanism so the author can add annotations to suit new view specific properties of nodes.

One might envision an editing environment based on the natural language view described in the next subsection. Currently the TreeView is the only view which allows this kind of authoring.

3.2 Presentation

The reader is not allowed to change any attribute, formal or presentation, of the CoqTree objects in the context. Figure 3 presents the NLView. This view presents proof-objects as Fitch style natural language proofs. The panel on the left shows the context. The panel on the right shows the context item that is selected. This view is described in more detail in Section 6.2. Statements

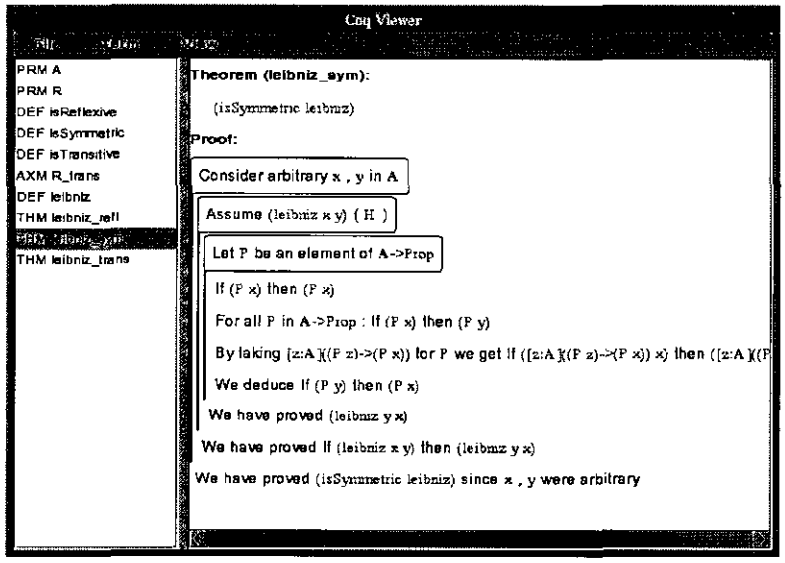


Figure 3: Screenshot of the Natural Language View.

are presented in natural language, and the reader can change the level of detail of certain parts of the proof text. The data-structure allowing this form of interaction is described in Section 6.2. It uses the type inference algorithm defined in Section 5.8.

The views on mathematical content that the tool generates can also be exported as an OMDOC document and then presented to the reader using an OMDOC browser. For more details, see Section 6.

4 Terms

This section describes the data-type CoqTree and its implementation in Java. All Coq terms are encoded as CoqTree nodes. The parser generates a CoqTree containing as root a *context* node.

On the first level under the context node are the *context item* nodes. There are two possibilities for a context item node: An *assumption* node or a *definition* node. Both of these are described in Section 4.1. A special kind of definition node is the *inductive definition* node, which is described in Section 4.4. The context item nodes have CoqTree terms as subtrees describing mathematical objects, statements about mathematical objects, or proofs of those statements.

Although, by the DeBruijn criterion, the set of nodes for terms can be limited to basic lambda calculus, the parser recognizes primitive nodes for many more notions, such as the logical connectives and the natural numbers. All of these notions can be defined in terms of lambda calculus, and this is how they are implemented in Coq. However, from a presentation point of view, since we want a presentation close to informal mathematics, it is a good idea to treat them as primitives.

A CoqTree consists of nodes connected by pointers. There are a number of different kinds of pointers. The foremost one, subtree, connects a parent node to the root nodes of its subtrees. Then there are the bindvar and bindsym pointers, which are used to indicate formal binding of variables. Furthermore, some temporary pointers copylink, alphalink, and typelink are needed during some of the operations described in Section 5.

Since the tool is implemented in the object oriented language Java, there are two distinct options for the representation of trees. The first option, in true object oriented style, is to define an abstract class CoqTree and define subclasses for every different kind of node. These classes are then organized in a hierarchy based on the inheritance relation, such that operations on similar nodes need only be specified once. For example, lambda- and Pi-nodes are both abstraction nodes and behave the same with respect to alpha-conversion, substitution, etc. The drawback of this option is that methods to manipulate the terms, such as the operations described in Section 5, are scattered throughout the different classes.

The second option, which is actually implemented, is to have one class for CoqTree of trees which has a field treekind indicating what kind of tree is represented. Most of the algorithms from Section 5 can now be specified using case distinction on this field. This style of coding is closer to the functional programming style.

4.1 Context and Context Item Nodes

What the parser gets from Coq is a context with definitions, assumption, and theorems with proofs. One context node is created, which has as subtrees all definition and assumption nodes. In Figure 4 a context node with three context items is shown. The nodes labeled BV are abstraction variables which are discussed in Section 4.3.

A definition node has three subtrees. The first one is its name (the definiendum). The second subtree is the actual value of the definition (the definiens). And the third subtree is is its type. Assumption nodes are just like definition nodes except they do not have a definiens subtree. They only declare a name with a type. Inductive definition nodes are described in Section 4.4.

Once a definition or an assumption has been declared, its name may be used in the rest of the context. These names are treated like variables, and therefore the definition and assumption nodes are in fact abstraction nodes, see

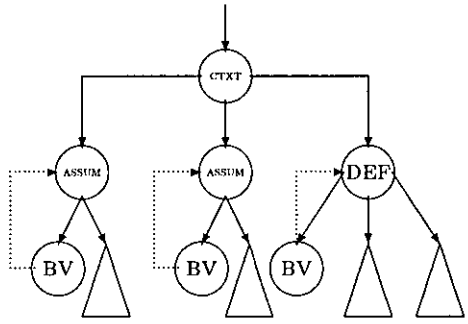


Figure 4: A context with two assumptions and a definition.

Section 4.3. A definition node differs from ordinary abstraction nodes, such as lambda nodes, in that the scope of the bound name is not a subtree of the node, but all sibling trees in the context to the right.

4.2 Basic Nodes

The simplest of all nodes is the *variable* node. It has no subtrees. Variable nodes do not have a name. Since all variables are bound the name can be stored in the formal abstraction variable node, see Section 4.3. The only relevant attribute of a variable is a bindvar pointer to this formal abstraction variable node.

Other basic nodes correspond to connectives like negation, conjunction, and application etc. These do have subtrees, but they introduce no other structure,

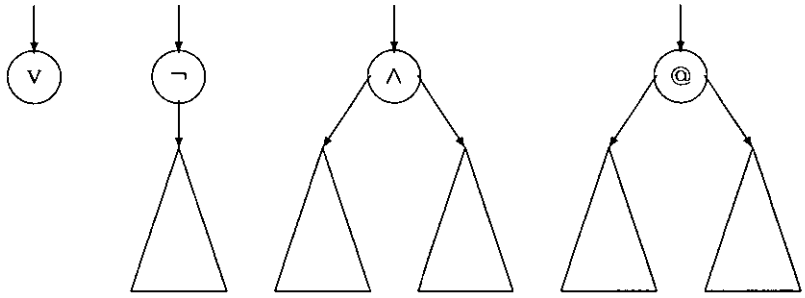


Figure 5: Variable, negation, conjunction, and application.

see Figure 5.

4.3 Abstraction Nodes

An *abstraction* node is used to introduce a formal name that binds occurrences of this name in the subtrees of the node. Usually there is only one subtree of an abstraction node where bound variable can occur called the *body*, but there are exceptions. The variable nodes occurring in the body in the tree need to be able to indicate that they are bound by this node. For example in the body (the third subtree) of the lambda abstraction node in Figure 6 a variable node

marked with V is bound by the lambda node. The binding is implemented using a bindvar pointer.

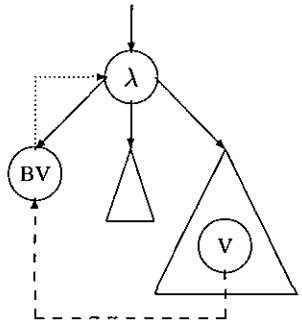


Figure 6: An abstraction node.

To be more accurate, the variable occurring in the body of the lambda term is bound by the formal *abstraction variable*, indicated by BV. It was decided to not bind variables to the abstraction node itself because we will encounter examples of abstraction nodes where multiple variables are bound at once. This decision forces us to add yet another link: The bindsym pointer provides a reference from the abstraction variable to the abstraction node. Certain operations on the tree use this bindsym reference to make decisions depending on the kind of abstraction used for some variable.

The second subtree contains the type of the abstraction variable and is called the *domain.*

Examples of abstraction nodes are lambda and Pi-nodes, but also definition and inductive definition nodes and the match nodes occurring in the cases construct.

4.4 Inductive Definition Nodes

Inductively defined sets are often used in mathematics. For example the set of natural numbers can be defined as the smallest set which contains 0 and which is closed under the successor operation S. In Coq this type is introduced with:

```
Inductive nat: Set :=  0: nat | S: nat->nat
```

Although all inductive definitions can be encoded in the calculus of Coq as second order types, inductive definitions were explicitly added to the calculus both for convenience and for efficiency reasons. The extension is done by introducing a new sort of definition called *inductive definition.* Both inductive sets and inductive propositions may be defined.

An inductive definition introduces a new name and states a type for the object it defines. It also introduces a number of constructors. Each constructor introduces a new name with a type. The type of a constructor may contain a reference to the type we are defining. There are some restrictions on the position where this name may occur, ensuring that the type defined is well-founded. A description of these restrictions is beyond the scope of this report, see for example [Gim94], and the CoqViewer does not check them since the context will be fully checked by Coq.

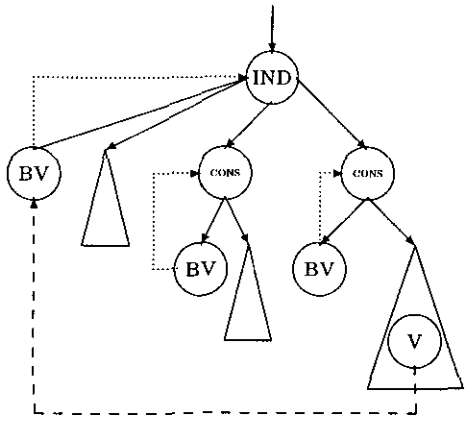


Figure 7: An inductive definition node with two constructors.

4.5 Cases and Match Nodes

The *cases* construct was added to the language to allow case distinction on values of inductive type. Every concrete object of an inductive type is constructed by repeated application of the type's constructors. Cases can be used to determine which constructor was applied last. See Section 4.6 for some examples of expressions involving cases.

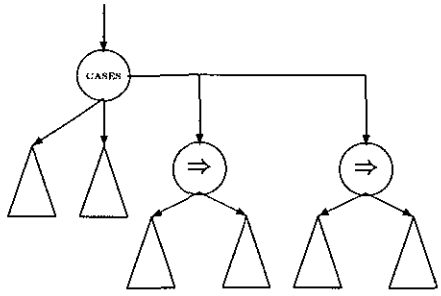


Figure 8: A cases node with two match nodes.

A cases node, Figure 8, has as subtree the term on which the case distinction is applied and the type of the expression itself. In addition there are several *match nodes*, marked with ⇒.

Each match node, Figure 9, contains a pattern in the left subtree and a body in the right subtree. During reduction, the term on which case distinction is applied is compared to each of the patterns using the matching algorithm described in Section 5.6. The matching algorithm returns a substitution for variables occurring in the corresponding body. The result of the reduction is this body after applying the substitution.

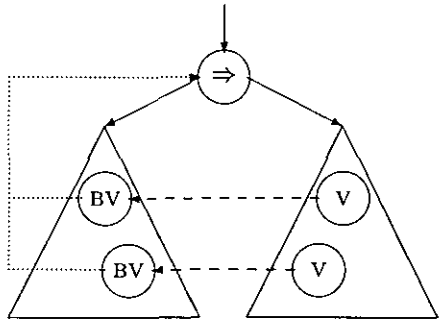


Figure 9: A match node with a pattern and a body.

4.6 Fixpoint Nodes

Fixpoints are used to define recursive functions. In Coq a recursive function may only be specified over inductively defined types. A fixpoint introduces a temporary name, the *fixpoint variable*, which may be used again in the body of the fixpoint construct.

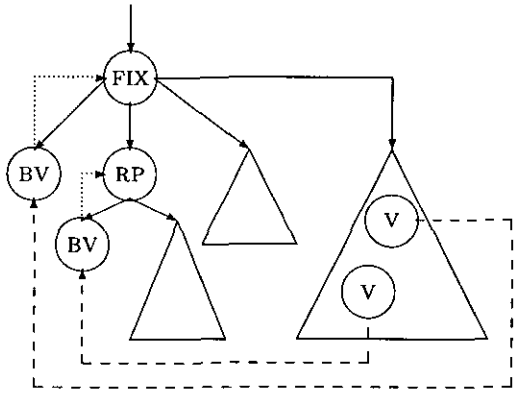


Figure 10: A fixpoint node with one recursion parameter.

A general recursion scheme like this would allow non-terminating functions, therefore some restriction is necessary. In Coq this is solved by demanding that a *recursion parameter* is mentioned explicitly. This is a variable whose value gets structurally smaller with every recursive application of the function. Early versions of Coq used a positive integer to indicate which variable plays the role of recursion parameter. For example the definition of the addition function would be printed as:

```
plus = fix f/1: nat->nat->nat :=
       { [n,m:nat]
         <nat>Cases n of
            0     => m
          | (S k) => (f k m)
```

```
    end
}
```

The number 1 indicates that the first lambda abstracted variable in the body, n, is the recursion variable. This is (roughly) the same notation used in [Gim94]. In newer versions of Coq a slightly friendlier but equally powerful notation is used:

```
plus = fix f [n:nat]: nat->nat :=
       { [m:nat]
         <nat>Cases n of
            0     => m
          | (S k) => (f k m)
         end
       }
```

The name of the recursion parameter n is mentioned explicitly in the type of the fixpoint variable. The square brackets in the type act as a lambda abstraction. If the recursion parameter is not the first variable, more variables need to be abstracted in this way.

The combination of fix and cases allows the Coq user to specify recursive functions using an intuitive looking syntax. The recursion check (to ensure recursive calls are applied to a smaller term than the value of the recursion parameter) and the positive occurrence check (on the definition of the inductive type) ensure that we get a terminating function.

5 Operations

Several operations can be defined on the CoqTree data-type. Ultimately what is needed for the NLView, described in Section 6.2, is a type inference algorithm. Type inference requires operations such as reduction and copying of terms. Because of the representation of the terms described in Section 4 these operations may not be very standard anymore. This section describes some of the problems encountered in implementing them.

5.1 Copy

For various purposes, for example type inference, it is useful to be able to make a copy of a term. The obvious way to copy a tree is to go top-down from the root to the leafs, copying all information in this node to a newly created one, and applying the copy method recursively to all subtrees. A problem arises here because we do not want to copy the binding links of bound variables literally, as this would bind the variables to abstraction nodes in the source tree. We would rather bind those variables to the copy of this abstraction node in the destination tree.

The solution is easy. While copying top-down, temporary *copy links* are made, connecting abstraction nodes, marked with BV in Figure 11, in the source tree to the corresponding abstraction nodes in the target tree. When a variable is encountered the copylink of the abstraction node in the source tree now points to the new abstraction node in the target tree.

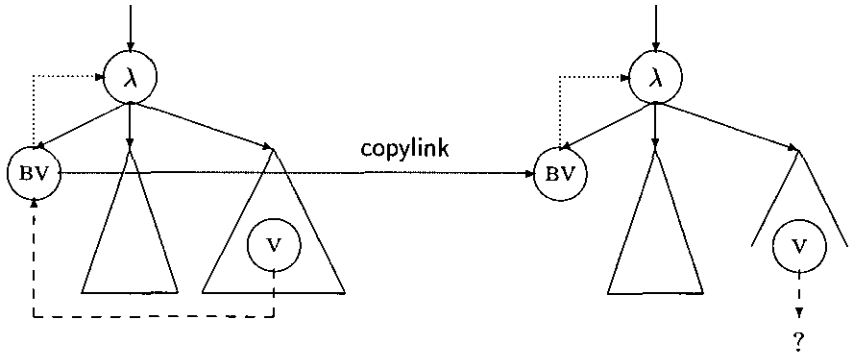


Figure 11: The copy method in action.

5.2 Syntactical Equivalence

To test for syntactical equivalence of terms, the trees in question should be isomorphic. This operation suffers from the same problem as the copy operation. To determine if two variables have the same abstraction node, it is not enough to compare the binding links. If the pointers are exactly the same, then the variables are alpha equivalent, but the two variables are also alpha equivalent if they have alpha equivalent abstraction variables.

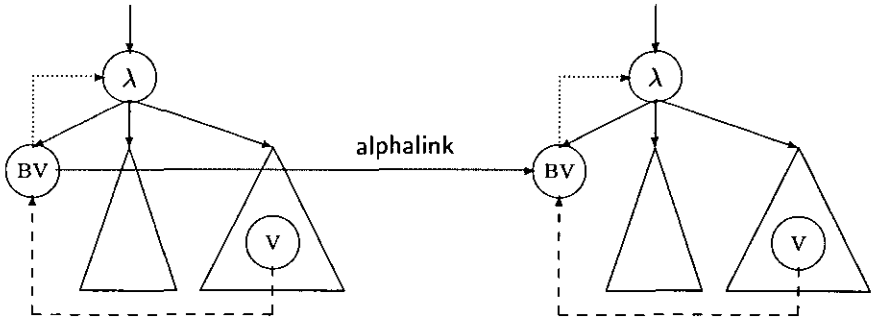


Figure 12: Testing for syntactical equivalence.

The solution is similar to the one used in the copy situation: we introduce temporary *alpha links*, connecting abstraction nodes, marked by BV in Figure 12, in the first tree to abstraction nodes in the second tree. Now we can compare the abstraction node of a variable in the second tree with the alphalink of the abstraction node of a variable in the first tree.

5.3 Currying

The application in lambda calculus takes two arguments: a function and its argument. Functions of higher arity are specified through *currying*, i.e. define the function in such a way that the result after one application is again a function which can be applied to another argument. For some of the operations described

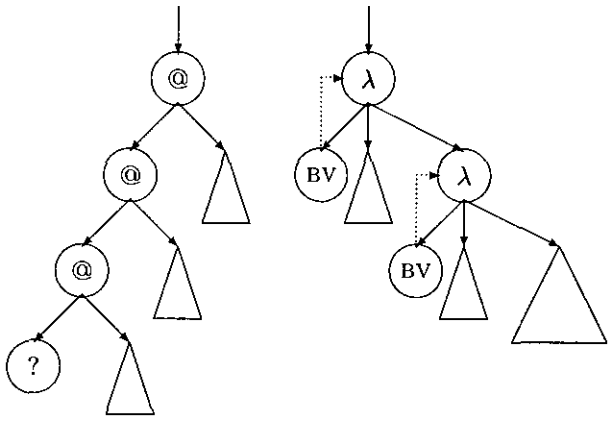


Figure 13: Currying of applications and abstractions.

in this section it is useful to be able to detect whether such a curried function is used, especially since most of the operations in this section traverse the tree recursively and can only see the kind of tree for the current node. We have operations that detect application spines such as in Figure 13 and yield pointers to the function and its arguments. This is for instance used in the reduction method to collect all the recursion parameters of a fix application.

A related problem is detecting repeated lambda and Pi abstractions. We also have operations for this. These are primarily used for presentation purposes such as pretty-printing of expressions in the views described in Section 6.

5.4 Substitution

Since bound variables cannot be identified by their name (remember that the name of a variable is stored in the abstraction variable which binds the variable) we use a pointer to their binding abstraction variable to identify them. This node is passed to the substitution method as a parameter.

The substitution method just traverses the tree top-down from root to leafs, replacing every variable bound to this parameter by a copy of the tree we want to substitute in place for it. This is done by the copy method.

5.5 Reduction

The reduction method reduces the term either to normal form or to weak head normal form. This method traverses the term leftmost outermost, reducing reducible expressions, or *redexes*. It considers three different kind of redexes: β, cases, and fix. Furthermore, if a variable node is encountered which is bound by a definition node, a so-called δ-redex, it is replaced by the corresponding

definition.

$$(\beta) \qquad (\lambda x : A.B)\, C \qquad \rightarrow \quad B[x := C]$$

$$(\text{cases}) \qquad \left(\begin{array}{lcl} \mathsf{cases}\,(\mathsf{constr}\vec{A})\ \mathsf{of} & & \\ \quad P_1 & \Rightarrow & B_1 \\ \quad\quad \vdots & & \\ \quad |P_n & \Rightarrow & B_n \\ \mathsf{end} & & \end{array}\right) \quad \rightarrow \quad B_i[\sigma]$$

$$(\text{fix}) \qquad (\mathsf{fix} f x T B)(\mathsf{constr}\vec{A}) \qquad \rightarrow \quad B[f := (\mathsf{fix} f x T B)](\mathsf{constr}\vec{A})$$

In the cases case, σ is the substitution produced by the match method and P_i is the pattern which matches with $(\mathsf{constr}\vec{A})$, see Section 5.6. In the fix case, f is the fixpoint variable and x is the recursion variable.

Weak head reduction only reduces the top level redex and does not continue to reduce the subterms. This is preferable if only the root symbol of a a term is needed. Weak head reduction is also useful in the natural language generation algorithms in Section 6.2. By only reducing a statement to weak head normal form, it remains as abstract as possible.

5.6 Matching

The matching method is only used in the cases construct to match terms against patterns that occur in the left hand side of the $\Rightarrow$ nodes. It takes as input a pattern and a tree and returns whether the tree matches the pattern. If they match, then a substitution σ is returned.

A pattern is either a new abstraction variable or a constructor (previously defined in an inductive type) or an application of patterns. Of course only well-typed patterns are allowed. Here $P \hat{=} M$ means the term M matches with the pattern P. The arrow on the left indicates that the rules should be tried from top to bottom, and the first rule that fits is used.

$$\downarrow \quad \begin{array}{rcll} (\mathsf{constr}\, c_1) & \hat{=} & (\mathsf{constr}\, c_2) & \text{if } c_1 \text{ and } c_2 \text{ are the same} \\ (\mathsf{bvar}\, x) & \hat{=} & T & \text{for all trees } T,\ \sigma := \sigma \cup \{x \mapsto T\} \\ T_1 & \hat{=} & T_2 & \text{if root symbols are equal} \\ & & & \text{and all subtrees match} \end{array}$$

5.7 Type Inference

Most typing algorithms for systems of typed lambda calculus are presented as inference rules, see for instance [Bar92]. The concept of *local context* plays an important role in this style of of presentation. Local contexts are used to keep track of free variables and their types. When the typing algorithm encounters an abstractor such as a lambda, it stores the variable with its type in the context and continues with the body of the term.

In our system, however, all variables are bound through binding links to abstraction variables in the *global context*. Our typing algorithm does not make use of local contexts. Whenever the algorithm encounters a variable, it retrieves its type by following the binding link to the abstraction node and copying the associated type.

The type of a term is built recursively. Based on the symbol of the current node, new nodes are created and the method continues recursively with the subtrees.

$$
\begin{array}{rcl}
\mathsf{type}(\mathsf{Set}) & = & \mathsf{type}(\mathsf{Prop}) = \mathsf{Type} \\
\mathsf{type}(\lambda x : A.B) & = & (\Pi x : A.\mathsf{type}(B)) \\
\mathsf{type}(FM) & = & \text{if } (\mathsf{type}(F) = (\Pi x : A.B)) \\
 & & \text{then } B[x := M] \\
 & & \text{else } \mathsf{failure} \\
\mathsf{type}(\mathsf{var}\, x) & = & \text{find the abstraction var of } x \\
 & & \text{and determine its type} \\
\mathsf{type}(A \to B) & = & \mathsf{type}(B) \\
\mathsf{type}(\Pi x : A.B) & = & \mathsf{type}(B) \\
\mathsf{type}(\mathsf{ind}\ x\ T\ \vec{C}) & = & T \\
\mathsf{type}(\mathsf{fix}\ f\ x\ T\ B) & = & T \\
\mathsf{type}(A \times B) & = & \mathsf{type}(A + B) = \mathsf{Set} \\
\mathsf{type}(\neg A) & = & \mathsf{type}(A \wedge B) = \mathsf{type}(A \vee B) = \mathsf{Prop}
\end{array}
$$

The type inference algorithm makes some assumptions about the term to be typed. For example, in the application case it is assumed that M has type A. The algorithm can be made more robust, so that it will always yield failure for terms that are not typeable. As the terms are part of a context that is checked by Coq it can safely be assumed that the terms are typeable.

During type inference, some copying of subtrees is required. For example when typing a tree with a lambda as root symbol, a new Pi-node is created which also abstracts a variable. The domain of this new binding variable is an exact copy of the domain of the binding variable of the lambda-node. A problem that arises is that the domain tree to be copied might contain variables with binding links which point to places outside of the destination tree.

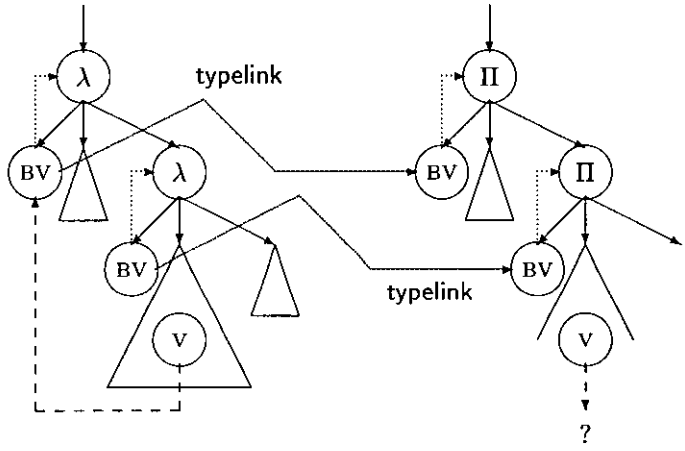


Figure 14: The type inference method in action.

For example to infer the type of the double lambda abstraction in Figure 14, a double Pi abstraction is created. In the tree on the left, the variable occurring

in the domain of the second lambda abstraction is bound by the first lambda abstraction. In the tree on the right, the corresponding variable needs to be bound by the first Pi abstraction. Therefore, the copy operations has to have knowledge about which lambda-nodes in the source tree correspond to which Pi-nodes in the destination tree.

To solve this problem, we introduce *type links* which connect the binding variables in the term to binding variables in the type. The copy algorithm uses these type links just like it uses copy links.

5.8 Computing the Expected Type

The above type inference algorithm derives a type for a term based on the derived types of its subterms. A type for a term is derived top-down.

Types for terms are not unique. A term can be assigned different types, although they will always be equivalent with respect to the reduction relation described in Section 5.5. Especially δ-redexes, i.e. definitions, can be used to make types look more abstract. For example the proof-object proving reflexivity of Leibniz' equality has derived type:

```
(x:A; P:A->Prop; H:(P x))(P x)
```

but the same term can also be assigned the more descriptive type:

```
(isReflexive leibniz)
```

In order to ensure the most abstract type for the subterms of a proof-object, the above algorithm is augmented to compute both a *derived type* and an *expected type*, similar to the algorithm described in [Cos96]. It takes as input a term and an expected type for this term, and annotates the term and all of its subterms with expected type information. At the same time the node is also annotated with it's derived type.

$$
\begin{array}{rcl}
\mathsf{exptype}(\lambda x\colon A.M, \tau) & = & \text{Annotate current term with } \tau. \\
 & & \text{Reduce } \tau \text{ to WHNF: } \tau = (\Pi x\colon A.B). \\
 & & \text{Call } \mathsf{exptype}(M, B). \\
\mathsf{exptype}(FM, \tau) & = & \text{Annotate current term with } \tau. \\
 & & \text{Let } \tau_F = \mathsf{type}(F),\ \tau_M = \mathsf{type}(M). \\
 & & \text{Reduce } \tau_F \text{ to WHNF: } \tau_F = (\Pi x\colon A.B). \\
 & & \text{Call } \mathsf{exptype}(F, \tau_F). \\
 & & \text{Call } \mathsf{exptype}(M, \tau_M). \\
\mathsf{exptype}(\mathsf{var}\, x, \tau) & = & \text{Annotate current term with } \tau.
\end{array}
$$

After parsing, each subtree of the context tree is annotated with derived and expected type attributes. The expected type of a subterm is computed from the expected type of the parent node. This means that the algorithms needs to be initialized with an expected type of the context item. Fortunately, since all definitions and proofs occur in the context with a preferred type, such an initial expected type is always available.

6 Views

This section describes the view mechanism. The idea is to have multiple views which present the underlying formal structure consistently. For example, when

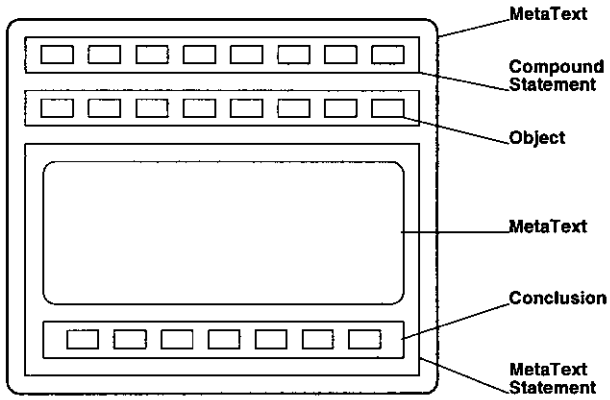


Figure 15: A MetaText.

a user changes the preferred level of detail at which a CoqTree object is viewed, this information is stored in the CoqTree itself and distributed to all views. Currently only two different views are available, the TreeView and the NLView. The context can also be exported as an OMDOC XML document, so that it can be viewed in a standard browser.

6.1 Tree View

The most basic view is called TreeView and just presents the context as a large tree, showing the nodes described in Section 4. Each node can be either collapsed or expanded. If a node is collapsed, the type of the tree starting beneath that node is displayed. If a node is expanded, the complete tree is displayed. See Figure 2 for a screenshot of this view. As described in Section 3, this view allows some simple editing of the presentation information.

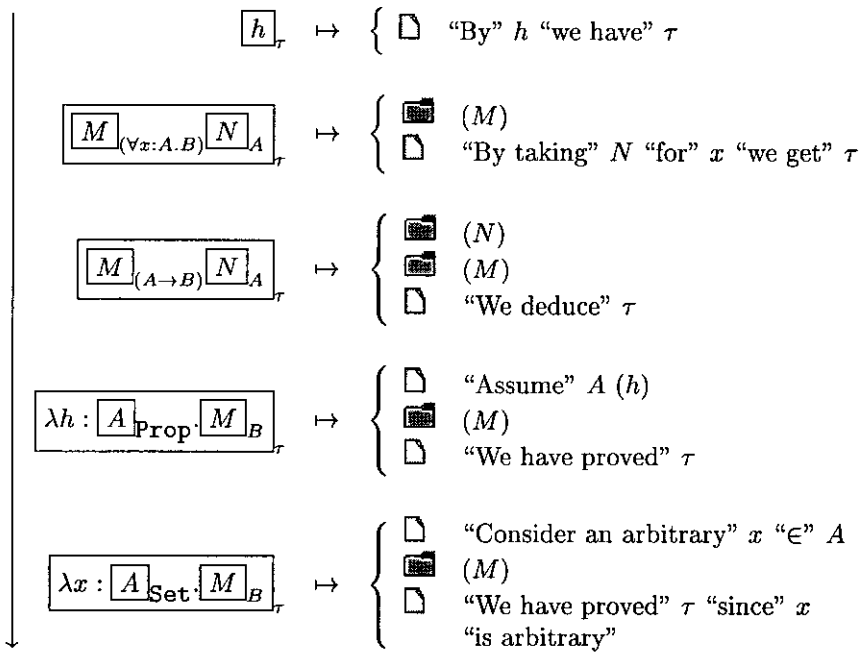
6.2 Natural Language View

The natural language view NLView is based on the standard translation algorithm presented in e.g. [CKT95] and [Cos96]. However, instead of generating flat text, objects of a new class called MetaText are generated, see Figure 15. The MetaText data-structure is intended to facilitate both the accessibility of the underlying formal proof-object, as well as the folding and unfolding mechanism of proofs in the natural language view. Moreover, assumptions and their scope can be marked which makes Fitch style presentation of proofs possible, see [Zwa98]. See Figure 3 for a screenshot of this view.

A MetaText contains a number of MathStatements which either consist of Objects or contain a recursive MetaText. Recursive MetaText statements also contain a conclusion statement. An Object is either some concrete text string or it is a pointer to a CoqTree representing a mathematical object term.

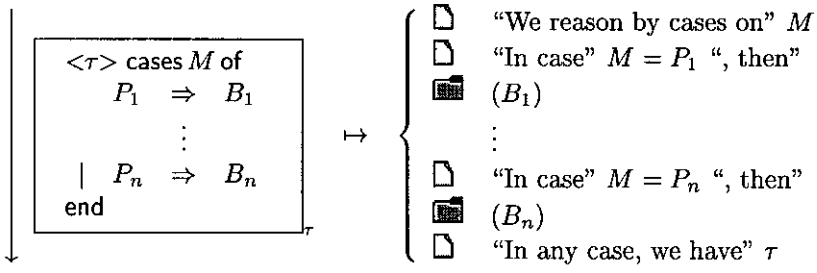
A CoqTree object is translated to a MetaText using the algorithm below. Here $\boxed{M}_\tau$ on the left hand side means "M is of type τ" and $\blacksquare(M)$ on the right hand side means "create a recursive MetaText containing M here". The

conclusion of such an embedded MetaText is the type of M.



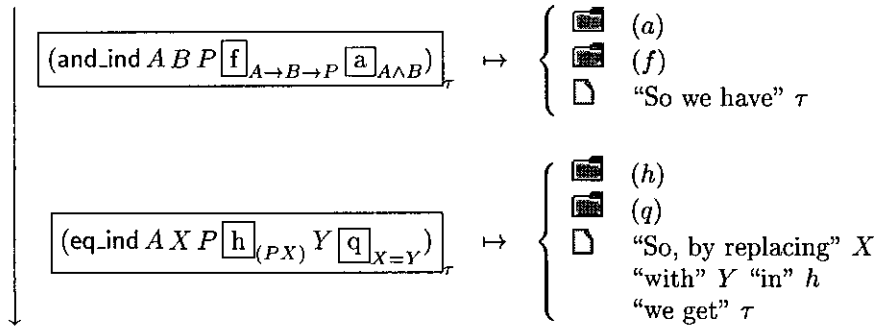
As can be seen from the many boxes on the left-hand side, unless every subterm is annotated with type information, a typing algorithm is necessary for this translation. The types τ used in the above algorithm are the expected types of the terms involved. This means that the translation is as abstract as possible: Definitions are only unfolded when necessary. The initial expected type is the type of the proof-object as it occurs in the context.

The above algorithm only specifies the translation for the basic calculus. Inhabitants of inductively defined theorems are translated as follows:



Some constructions involving defined constants such as elimination of conjunction, disjunction, equality, and natural numbers (induction) require alternative translations. Although these constants have a definition in Coq, we treat

them as primitives for the purpose of this translation.



6.3 Generating OMDOC Documents

OMDOC [Koh99] stands for OpenMath Document. It is an extension of the OpenMath [Ope99] language. OpenMath is intended as a communication medium between symbolic computation engines. Objects encoded in OpenMath can be shared by different systems. OMDOC enables the communication of complete contexts by providing constructs for assumptions, definitions, theorems and proofs. This makes it an excellent choice as a vehicle for type theoretical contexts.

A first prerequisite for converting CoqTree contexts to OMDOC documents is an OpenMath encoder which translates CoqTree object terms to OpenMath objects. In OpenMath terminology such an encoder consists of a *content dictionary* and a *phrasebook*. A content dictionary for Coq is defined in [CC99].

Using the translation for objects, it is easy to define the translation for contexts. The context items in an OMDOC document consist of a formal and an informal description of the object that is defined or assumed. For the formal description the OpenMath equivalent of the formal (proof-)object is taken. The informal description is generated by first verbalizing the object as a MetaText and then replacing the formal parts of the text by OpenMath objects.

The OMDOC document can be exported in XML format. Consider for example a step from the proof of reflexivity of Leibniz' equality.

```
<derive id="leibniz_refl-prf.2p.2p.2p.1">
  <CMP>
    By
    <OMOBJ>
      <OMV name="H"/>
    </OMOBJ>
    we have
    <OMOBJ>
      <OMA>
        <OMV name="P"/>
        <OMV name="x"/>
      </OMA>
    </OMOBJ>
  </CMP>
</derive>
```

This step contains a *commented mathematical property*, between CMP tags, which consists of natural language text with embedded OpenMath objects.

The generated XML can be transformed into for example HTML using an appropriate XSL style-sheet. The OMDOC view is not really a view in the sense described above. Since the presentation of the generated XML is done in a different tool, there is no direct connection back to the CoqTree structure a document was generated from. However, all the formal MathObjects which occur in a MetaText are translated to OpenMath objects. This means that the formal content is still present in the OMDOC document.

The use of standard export formats for mathematics like OpenMath and OMDOC potentially allows communication of objects to symbolic computation engines such as theorem provers and computer algebra systems. This may lead to true interactive mathematical documents in the sense of [CO00b, CO00a], and is object of future work.

7 Conclusions

We described the requirements and implementation of a prototype tool, called CoqViewer. The tool can be used for authoring and presenting mathematical content based on type theory. The Coq system is used to create an initial mathematical theory. Using the tool, an author can enhance the definitions and proof-objects that form a formally derived mathematical context with presentation information.

The tool can then display the content in several views allowing for interactivity by presenting the formal content on different levels of detail. The formal content does not change as a result of these interactions. One example of a view is the natural language view. Although the reasoning displayed in this view resembles informal mathematics, the underlying formal objects remain accessible. This means that they can still be communicated to symbolic computation engines.

The lambda terms representing the mathematical content are implemented inside the tool as pointer trees. References are used to indicate bound variables. This causes some non-trivialities in implementing the usual algorithms such as copying, checking syntactical equivalence and type inference.

Type inference is needed in order to implement the natural language view. Because it can be assumed that the input to the tool is a checked mathematical context, some side conditions are not checked during reduction and type inference.

The OpenMath language and its extension OMDOC may be used to connect this tool to other similar tools and makes viewing of the content in standard browsers possible. This is the subject of future work.

Acknowledgments

Tijn Borghuis, Olga Caprotti, and Rob Nederpelt read early versions of this report and gave many useful suggestions. Robert Brouwer and Roel Körvers helped implement some of the fancy features in the natural language view.

Thanks also to Herman Geuvers and Jan Zwanenburg for many discussions about presentation of mathematics.

References

[Bar92] H. Barendregt. *Lambda calculi with Types*, volume 2 of *Handbook of Logic in Computer Science*, chapter 2, pages 117–309. Oxford Science Publications, 1992.

[BB97] H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. Available from http://www.cs.kun.nl/~henk, 1997.

[BBC+99] G. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant, Reference Manual, Version 6.3.* INRIA-Rocquencourt - CNRS-ENS Lyon, July 1999.

[CC99] O. Caprotti and A. M. Cohen. A Type System for OpenMath, February 1999. OpenMath Deliverable 1.3.2b.

[CKT95] Y. Coscoy, G. Kahn, and L. Thery. Extracting text from proof. In M. Dezani and G. Plotkin, editors, *Proceedings of Int. Conf. on Typed Lambda-Calculus and Applications (TLCA), Edinburgh*, volume 902. Springer-Verlag LNCS, April 1995.

[CO00a] O. Caprotti and M. Oostdijk. How to formally and efficiently prove prime(2999). In *Proceedings of Calculemus 2000, St. Andrews*, August 2000.

[CO00b] O. Caprotti and M. Oostdijk. Proofs in interactive mathematical documents. In *Proceedings of AISC 2000, Madrid*, July 2000.

[Cos96] Y. Coscoy. A natural language explanation for formal proofs. In C. Retore, editor, *Proceedings of Int. Conf. on Logical Aspects of Computational Liguistics (LACL), Nancy*, volume 1328. Springer-Verlag LNCS/LNAI, September 1996.

[Gim94] E. Giménez. Codifying guarded definitions with recursive schemes. Technical Report 95-07, Ecole Normale Supérieure de Lyon, December 1994.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, Massachusetts, 1996.

[HMBC99] A. M. Holland-Minkley, R. Barzilay, and R. Constable. Verbalization of high-level formal proofs. In *Sixteenth National Conference on Artificial Intelligence*, 1999.

[How80] W. A. Howard. *The formulae-as-types notion of construction*, pages 479–490. To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, 1980.

[Koh99] M. Kohlhase. OMDOC: Towards an Openmath Representation of Mathematical Documents. Technical report, DFKI, Saarbrücken, 1999.

[Lam94] L. Lamport. *LaTeX: A Document Preparation System, (2nd ed.).* Addison-Wesley, Reading, Massachusetts, 1994.

[Ope99] OpenMath Consortium. The Openmath Standard, August 1999. OpenMath Deliverable 1.3.3a.

[Zwa98] J. Zwanenburg. The Proof-assistant Yarrow. Technical Report 98/11, Eindhoven University of Technology, July 1998.

Computing Science Reports **Department of Mathematics and Computing Science**
Eindhoven University of Technology

If you want to receive reports, send an email to: m.m.j.l.philips@tue.nl (we cannot guarantee the availability of the requested reports)

In this series appeared:

96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.

97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.
97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28
97/15	M. Franssen	λP-: A Pure Type System for First Order Loginc with Automated Theorem Proving, p.35.
97/16	W.M.P. van der Aalst	On the verification of Inter-organizational workflows, p. 23
97/17	M. Vaccari and R.C. Backhouse	Calculating a Round-Robin Scheduler, p. 23.
97/18	Werkgemeenschap Informatiewetenschap redactie: P.M.E. De Bra	Informatiewetenschap 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie Informatiewetenschap, p. 60.
98/01	W. Van der Aalst	Formalization and Verification of Event-driven Process Chains, p. 26.
98/02	M. Voorhoeve	State / Event Net Equivalence, p. 25
98/03	J.C.M. Baeten and J.A. Bergstra	Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04	R.C. Backhouse	Pair Algebras and Galois Connections, p. 14
98/05	D. Dams	Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06	G. v.d. Bergen, A. Kaldewaij V.J. Dielissen	Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07	Proceedings of the workhop on Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98) June 22, 1998 Lisbon, Portugal	edited by W. v.d. Aalst, p. 209
98/08	Informal proceedings of the Workshop on User Interfaces for Theorem Provers. Eindhoven University of Technology ,13-15 July 1998	edited by R.C. Backhouse, p. 180
98/09	K.M. van Hee and H.A. Reijers	An analytical method for assessing business processes, p. 29.
98/10	T. Basten and J. Hooman	Process Algebra in PVS
98/11	J. Zwanenburg	The Proof-assistemt Yarrow, p. 15
98/12	Ninth ACM Conference on Hypertext and Hypermedia Hypertext '98 Pittsburgh, USA, June 20-24, 1998 Proceedings of the second workshop on Adaptive Hypertext and Hypermedia.	Edited by P. Brusilovsky and P. De Bra, p. 95.
98/13	J.F. Groote, F. Monin and J. v.d. Pol	Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.
98/14	T. Verhoeff (artikel volgt)	
99/01	V. Bos and J.J.T. Kleijn	Structured Operational Semantics of χ , p. 27
99/02	H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst	Diagnosing Workflow Processes using Woflan, p. 44

99/03	R.C. Backhouse and P. Hoogendijk	Final Dialgebras: From Categories to Allegories, p. 26
99/04	S. Andova	Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81
99/05	M. Franssen, R.C. Veltkamp and W. Wesselink	Efficient Evaluation of Triangular B-splines, p. 13
99/06	T. Basten and W. v.d. Aalst	Inheritance of Workflows: An Approach to tackling problems related to change, p. 66
99/07	P. Brusilovsky and P. De Bra	Second Workshop on Adaptive Systems and User Modeling on the World Wide Web, p. 119.
99/08	D. Bosnacki, S. Mauw, and T. Willemse	Proceedings of the first international syposium on Visual Formal Methods - VFM'99
99/09	J. v.d. Pol, J. Hooman and E. de Jong	Requirements Specification and Analysis of Command and Control Systems
99/10	T.A.C. Willemse	The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μ CRL, p. 44.
99/11	J.C.M. Baeten and C.A. Middelburg	Process Algebra with Timing: Real Time and Discrete Time, p. 50.
99/12	S. Andova	Process Algebra with Probabilistic Choice, p. 38.
99/13	K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen	A Framework for Component Based Software Architectures, p. 19
99/14	A. Engels and S. Mauw	Why men (and octopuses) cannot juggle a four ball cascade, p. 10
99/15	J.F. Groote, W.H. Hesselink, S. Mauw, R. Vermeulen	An algorithm for the asynchronous *Write-All* problem based on process collision*, p. 11.
99/16	G.J. Houben, P. Lemmens	A Software Architecture for Generating Hypermedia Applications for Ad-Hoc Database Output, p. 13.
99/17	T. Basten, W.M.P. v.d. Aalst	Inheritance of Behavior, p.83
99/18	J.C.M. Baeten and T. Basten	Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79
99/19	J.C.M. Baeten and C.A. Middelburg	Real Time Process Algebra with Time-dependent Conditions, p.33.
99/20	Proceedings Conferentie Informatiewetenschap 1999 Centrum voor Wiskunde en Informatica 12 november 1999, p.98	edited by P. de Bra and L. Hardman
00/01	J.C.M. Baeten and J.A. Bergstra	Mode Transfer in process Algebra, p. 14
00/02	J.C.M. Baeten	Process Algebra with Explicit Termination, p. 17.
00/03	S. Mauw and M.A. Reniers	A process algebra for interworkings, p. 63.
00/04	R. Bloo, J. Hooman and E. de Jong	Semantical Aspects of an Architecture for Distributed Embedded Systems*, p. 47.
00/05	J.F. Groote and M.A. Reniers	Algebraic Process Verification, p. 65.
00/06	J.F. Groote and J. v. Wamel	The Parallel Composition of Uniform Processes wit Data, p. 19
00/07	C.A. Middelburg	Variable Binding Operators in Transition System Specifications, p. 27.
00/08	I.D. van den Ende	Grammars Compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases, p. 33.
00/09	R.R. Hoogerwoord	A Formal Development of Distributed Summation, p. 35
00/10	T. Willemse, J. Tretmans and A. Klomp	A Case Study in Formal Methods: Specification and Validation on the OM/RR Protocol, p. 14.
00/11	T. Basten and D. Bošnački	Enhancing Partial-Order Reduction via Process Clustering, p. 14
00/12	S. Mauw, M.A. Reniers and T.A.C. Willemse	Message Sequence Charts in the Software Engineering Process, p. 26
00/13	J.C.M. Baeten, M.A. Reniers	Termination in Timed Process Algebra, p. 36