

# Connecting Logical Representations and Efficient Computations

Martin Pollet

*Fachbereich Informatik, Universität des Saarlandes, Germany*  
pollet@ags.uni-sb.de, www.ags.uni-sb.de/~pollet

Volker Sorge

*School of Computer Science, The University of Birmingham, UK*  
V.Sorge@cs.bham.ac.uk, www.cs.bham.ac.uk/~vxs

---

## Abstract

When combining logic level theorem proving with computational methods it is important to identify both functions that can be efficiently computed and the objects they can be applied to. This is generally achieved by mappings of logic level terms and functions to their computational counterparts. However, these mappings are often quite ad hoc and fragile depending very much on the particular logic representations of terms. We present a method of annotating terms in logic proofs with their computational properties. This enables the compact representation of computational objects in deduction systems as well as their connection to functions that can be easily computed for them. This eases the identification of deduction problems that can be treated efficiently by computational methods and also abstracts from trivial properties that are artefacts of a particular representation. We ensure logical correctness of our concepts by providing the possibility to replace terms by their logical representation and by expanding computational procedures by tactic application that can be rigorously checked.

*Key words:* Computation and Reasoning, Mathematical Knowledge Representation, Annotated Terms

---

## 1 Introduction

The representation of mathematical objects in deduction systems is often dictated by the requirements of the formalism and logic of a particular system. For instance, natural numbers are often represented in terms of successors of zero or lists of numbers are recursively concatenated via constructor functions. While these representations are generally suitable for reasoning about properties of the abstract mathematical concept they are often a hindrance when dealing with concrete objects, i.e., instances of the abstract concept, and

*This is a preliminary version. The final version will be published in  
Electronic Notes in Theoretical Computer Science  
URL: [www.elsevier.nl/locate/entcs](http://www.elsevier.nl/locate/entcs)*

their computational properties. Not only are the representations often rather detached from the informal mathematical vernacular but also from a representation that is suitable for direct computational manipulation. Moreover, it is often already difficult to automatically identify these objects inside complex formulas.

When we take a look at typical representations in mathematics it seems that information about the objects is attached to the object itself. It starts with the choice of letters:  $a$  seems to be a better notation for an element of the set  $A$  than any other letter, capital letters denote sets,  $G$  stands for a group in the context of group theory,  $n$  and  $m$  are the ‘typical’ arbitrary natural numbers. Formal systems are able to attach this kind of information to objects by using types, and make it possible to identify objects or properties by their type. Other representations are harder to model, e.g., associativity of an operation is remembered by forgetting the brackets, ‘+’ is used for different addition-like operations.<sup>1</sup> These observations suggest a more object oriented approach: Namely, to store information about an object at the object itself rather than in detached procedures, for instance, of the interface. This also eases the identification of certain objects, for example in complex formulae, and the reuse of information on the objects for different purposes.

To capture the information connected to certain mathematical representations we have introduced the data structure of *annotated constants* [14] (see Sec. 3). It handles particular classes of objects that are given as term in a logic language but that should be treated as constants from a mathematical point of view. Annotated constants replace terms with logical constants that contain the information on the actual object as annotation. The annotation allows on the one hand to reconstruct the original object, should it be necessary, in the formal proof. On the other hand it enables the recognition of the object by specialised proof rules as well as to perform efficient manipulations and computations on the objects represented. We will present two examples for mathematical concepts captured by annotated constants in Sec. 2.

We have implemented several classes of annotated constants in the Omega proof development environment [13] to ease automatic proof construction, mainly in proof planning scenarios. Besides simple objects like numbers, lists and sets, we have also experimented with more complex objects such as matrices [15] and permutations [4]. In particular, when automatically certifying computer algebra algorithms in Omega a large number of concrete mathematical objects can be handled efficiently using annotated constants. Unfortunately, the spectrum of objects that can be handled by annotated constants in their current form is restricted to concrete terms, that is, terms that do not contain variables. However, it is often desirable to also identify a term containing variables as a particular mathematical object. For instance, we would like to distinguish objects representing finite sets even though they

---

<sup>1</sup> Overloading allows to reuse symbols but does not help to reuse the knowledge about the symbols.

contain variables, in order to perform efficient set manipulation on them. We therefore extend our notion of annotated constants to that of *annotated terms* (Sec. 4) that allows for terms with variables and show some of the impacts this has on computations that can be carried out on them. Since our concrete implementation is within the logical framework of the Omega system — a simply typed lambda calculus (cf. [1]) — we will present our examples in this formalism. However, the general concept of annotated constants and terms is not restricted to a particular logic system.

## 2 Two Examples

We first examine two examples to motivate our concept of annotated constants. Commonly deduction systems depend on the use of a finite signature, i.e., a finite set of constants, functions, and predicate symbols. Therefore, infinite sets of constants are generally recursively constructed, which means that the individual objects are given in terms of their construction rather than as the constants they actually are from a mathematical point of view. This fact makes these objects not only cumbersome to handle but often difficult to identify. While some objects such as integers or lists can still be fairly easily identified in their formal logic representation even when embedded in complex terms, for other constructs this is not so obvious.

For instance, in lambda calculus finite sets are usually represented as lambda terms containing a disjunction of equalities. A set of the form  $\{a, b, c\}$  is represented as  $\lambda x.(x = a \vee x = b \vee x = c)$ . Now it is not necessarily obvious whether a lambda term actually represents a finite set or not. Moreover, equality between sets is independent of the order of its elements. However, the (syntactic) equality on lambda expression depends on the order.

Another example is the formal representation of matrices. A mathematical definition for matrix is for instance given in [11, p.441]:

*“By an  $m \times n$  **matrix** in  $R$  one means a doubly indexed family of elements of  $R$ ,  $(a_{ij})$ , ( $i = 1, \dots, m$  and  $j = 1, \dots, n$ ), usually written in the form*

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ & \cdots & \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

*We call the elements  $a_{ij}$  the **coefficients** or **components** of the **matrix**.”*

Translating this definition into a formal representation is straightforward. Depending on the exact logical language, one would consider a matrix as a tuple consisting of a double indexed function, number of rows, number of columns, and the underlying ring. For an instance of a concrete matrix one can then give the function in the following way:

$$a : i, j \longrightarrow \begin{cases} 3, & \text{if } i = j \wedge i \in [1, 2] \\ 1, & \text{if } i = 1 \wedge j = 2 \\ 1, & \text{if } j = 1 \wedge i = 2 \end{cases} \quad \text{with } (a_{ij}) = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}.$$

While the functional representation on the left hand side is sufficient to describe a matrix, it has already lost information implicitly given by the actual matrix representation on the right hand side: The definition introduces the representation as a rectangular form in which the elements of a matrix are ordered with respect to their indices to make relevant information directly accessible and ease reasoning. However, if we look at one representation of the above matrix in lambda calculus, using an if-then-else construct  $[\lambda i, j. \text{if } (i = j \wedge (i = 1 \vee i = 2)) \text{ then } 3 \text{ else } 1]$  it is no longer obvious that this even suffices to define a rectangular structure. Other obvious information, such as symmetry, are also less accessible in the lambda term. And even accessing components of the matrix will require considerable reasoning.

While some problems arising from the sketched formal representations of concrete mathematical objects could still be handled by adding some syntactic sugar and elaborate translation and display facilities, the handling on the term level would still remain difficult. In particular, for an automated system (e.g., an automated theorem prover, a proof planner or a computer algebra system) it becomes a problem to automatically distinguish which part of the formula constitutes concrete mathematical objects and which not. This is especially important when we want to avoid semantically incorrect expressions that can arise from manipulating the functional expression without adhering to constraining conditions. While a sublist can usually be substituted without violating additional properties, manipulations of matrix expressions need to explicitly preserve the rectangular nature of the object. Therefore, it is of help if certain terms or sub-terms can be explicitly marked as concrete mathematical objects that have to adhere to certain side-conditions. We achieve this by using annotated constants.

### 3 Annotating Constant Objects

Annotated constants are a mechanism that provides a representational layer that can both capture the properties of the intuitive mathematical representation of computational objects, as well as connect these objects to their corresponding representation in a formal logic framework. Annotated constants are implemented in the Omega system [13] and offer special treatment for simple objects, such as numbers and lists, but also for more complex structures like permutations [14], matrices, block matrices and matrices containing ellipses [15]. For the sake of clarity we explain the idea in the following using the much simpler example of finite sets.

#### 3.1 Annotated Constants

Given a logic language  $\mathcal{L}$ , a constant symbol  $c \in \mathcal{L}$  and a ground term  $t \in \mathcal{L}$ , such that  $c = t$ . We then define an annotated constant to be a triple  $(c, t, \mathbf{a})$ , where  $\mathbf{a}$  is the annotation. The *annotation*  $\mathbf{a}$  is any object (making use of an arbitrary data structure) from which  $c$  and  $t$  can be reconstructed. Think

of  $c$  as the name of the object,  $t$  as the representation within logic, and  $\mathbf{a}$  as a representation of the object outside logic. The idea is that  $\mathbf{a}$  is some computational object we are interested in, which should be regarded as some constant object. However, its actual formulation in the logic language is the  $t$ , which is in general not a constant. Thus in order to distinguish  $t$  as a computational object inside the logic language, it is replaced by the constant  $c$ . The annotated constant  $(c, t, \mathbf{a})$  then allows us to efficiently compute with  $c$  by using the annotation  $\mathbf{a}$  as well as to guarantee that we can regain the full logical formalisation by replacing  $c$  with  $t$ .

As an example of one type of annotated constants we consider finite sets. Finite sets have a special notation in the mathematical vernacular, for example, the set consisting of the three elements  $a$ ,  $b$ , and  $c$  is denoted by  $\{a, b, c\}$ . We can define this by giving the set a name, e.g.,  $A$ , and a definition in logic as a ground term. Important knowledge about sets with which it is appropriate to reason efficiently is: sets are equal if they contain the same elements regardless of their order, or the union of two sets consists of the elements which are a member of one of the sets and so on. This type of set manipulation has not so much to do with logical reasoning as it has with computation. The union of two sets, for instance, can be very efficiently computed and should not be part of the process of search for a proof.

Annotated constants for finite sets are defined with the attributes

**Constant symbol:** We give the set a name such as  $A$ . Even more appropriate for our purpose is to generate an identifier from a duplicate free ordering of the elements of the set, for the example  $A_{\{a,b,c\}}$ .

**Definition:** The definition of the set corresponds to a lambda term in higher-order logic, e.g.,  $\lambda x.(x=a \vee x=b \vee x=c)$ . In order to normalise such terms it is useful to order the elements of the set, that is, we wouldn't write the term as  $\lambda x.(x=b \vee x=a \vee x=c)$ . Since the annotation has to represent the object completely the formal definition can be constructed from the annotation.

**Annotation for finite sets:** The data structure of sets of the underlying programming language is used as annotation and the elements of the set are restricted to closed terms, e.g., the set containing the three constants  $a$ ,  $b$ , and  $c$  in the concrete example.

We use the annotation for the presentation of the concept in a standard mathematical notation, and the annotation is given as input by the user. The constant symbol and its definition are chosen according to the annotation. This means that the sets  $\{a, b, c\}$  and  $\{c, b, a\}$  would be represented by the same constant symbol because the annotations are equal. With this mechanism it is possible to implement trivial equalities on annotated constants as syntactic equality within our object logic.

The basic functionality for handling annotated constants is implemented on the term level of the Omega system. In first approximation, an annotated constant is a constant with a definition and has the type of its defining term  $t$ . As such it could be replaced by its defining term during the proof or when

expanding the proof to check formal correctness. Typically, this is not done, but annotated constants are manipulated via their annotations. The defining term of an annotated term is used only when necessary.

### 3.2 Annotated Functions

Now that we have a concise representation for computational objects available on the logical term level, we can exploit it for efficient manipulation of the objects. In our original approach [14] annotated constants could be manipulated by special tactics that could operate on them. This, however, restricts the detection of possible operations on annotated constants to the hazards of the actual proof process. An alternative is to identify functions and predicates that can operate on particular types of annotated constants and enhance them with additional computational information. This information can subsequently be exploited for the manipulation of annotated constants and verification of their properties.

We achieve this by introducing procedural annotations as follows: Let  $f$  be a function or predicate of the logical language  $\mathcal{L}$ . A procedural annotation is a triple  $(f, T, \mathbf{p})$ , where  $\mathbf{p}$  is a procedure of the underlying programming language with the same number of arguments as  $f$ , and  $T$  is a specification (or tactic) for the construction of a formal proof for the manipulation performed by  $\mathbf{p}$ .

The definition is parallel to the one for annotated constants, that is,  $f$  is a constant function symbol from  $\mathcal{L}$ ,  $\mathbf{p}$  is the annotation, and  $T$  is the formal counterpart of  $\mathbf{p}$ . Observe, however, that the role of  $T$  is different from the role of  $t$  for annotated constants, since  $T$  is a proof tactic of the underlying calculus, rather than an element of the logic language  $\mathcal{L}$ . Manipulations using procedural annotations can be carried out by using the procedure  $\mathbf{p}$  directly. It checks its arguments, performs the simplification, and returns a simplified constant or term together with possible conditions for this operation. This, however, results in a proof step that is essentially the abbreviation of a more complex inference on the calculus level. Moreover, since  $\mathbf{p}$  can be any procedure it is not guaranteed to be correct. It is therefore necessary to formally justify the correctness of the computation in a subsequent step using the proof specification  $T$ . Thereby an annotated constant is expanded to its formal definition and the computation is reconstructed by tactic and theorem applications. This expansion will be done only when a low level formal proof is required, certainly not during proof search.

As an example, we consider the procedure for the union of concrete sets.

**Function:** The union function on sets is defined to be  $\cup \cong (\lambda S, T, x. Sx \vee Tx)$ .

This means that  $\cup$  is a constant symbol in our logic language that can be applied to any two sets regardless of the form they are given. However, we can now annotate  $\cup$ , such that when applied to annotated constants representing finite sets the union can be efficiently computed. For example, we want to be able to compute the union  $\{a, b\} \cup \{c, d\}$  when  $\{a, b\}$  and

$\{c, d\}$  are annotated constants.

**Procedural annotation:** A procedure  $\mathbf{p}$  that computes the union of two finite sets given by concrete elements.  $\mathbf{p}$  first checks whether the argument of  $\cup$  are annotated constants denoting finite sets. If this is the case  $\mathbf{p}$  computes as a duplicate free concatenation of the two sets and returns it as a new annotated constant to be inserted into the proof. For the concrete example  $\{a, b\} \cup \{c, d\}$  the procedure checks whether the arguments are annotated constants for concrete sets, and returns the annotated constant which has the concatenation of  $\{a, b, c, d\}$  as annotation.

**Tactic:** The tactic  $T$  expands the computation of  $\mathbf{p}$  by expanding it to a logic level computation. It first substitutes  $\cup$  and the annotated constants in its arguments by their definitions and then applies low level inference rules to construct the union. For the concrete example,  $T$  substitutes  $\{a, b\} \cup \{c, d\}$  by the appropriate lambda terms, which results in  $(\lambda S, T, x. Sx \vee Tx)(\lambda x. x = a \vee x = b)(\lambda x. x = c \vee x = d)$  and afterwards applies  $\beta$ -reduction and possibly reordering of the logical connectives to get the resulting set  $\lambda x. x = a \vee x = b \vee x = c \vee x = d$ .

In this example the procedural annotation only replaces definition expansion and  $\beta$ -reduction steps. The important benefit of the procedure is, that the result is again an object that is immediately identified as finite set without the need for further analysis.

The procedural annotations can produce different output with respect to different types of input. They may differentiate between arguments which are terms, constants, annotated constants, annotated constants of a specified kind, or the name of a concrete constants. The procedure checks the arguments and chooses the case which is most specific, where any term is least specific, and a concrete constant is most specific. The element relation  $a_1 \in a_2$ , for example, is implemented for the following cases:

- (i) If the argument  $a_1$  is an annotated constant of kind integer, and  $a_2$  is the concrete constant for integers  $\mathbb{Z}$ , then it returns true.
- (ii) If  $a_1$  is a term and  $a_2$  is an annotated constant of kind finite set, then it checks whether  $a_1$  is equal to one of the elements in  $a_2$ .

The procedural annotations are combined in an evaluation tactic, which applies the procedural annotations connected to functions contained in a term, and to the output of the procedural annotation until no procedural annotation is applicable. When this tactic is applied to the formula  $\{1, 2, \} \cup \{2, 3\} \subset \mathbb{Z}$  it finds that the annotation for the subset relation is not applicable since it needs an annotated constant of kind finite set as first argument, but the first argument contains the union of sets for which the procedural annotation can be applied. The resulting output  $\{1, 2, 3\} \subset \mathbb{Z}$  is again evaluated. This time the annotation for the subset relation is applicable, and it returns a list containing  $1 \in \mathbb{Z}$ ,  $2 \in \mathbb{Z}$ , and  $3 \in \mathbb{Z}$ . Each of the formulas is again evaluated and the annotation of the element relation returns true for each of them. This means

the evaluation tactic is applicable and returns no further proof obligations.

### 3.3 Discussion

Firstly, annotated constants provide an intermediate representation layer between the intuitive mathematical vernacular and a formal system. With annotated constants it is possible to abstract from the formal introduction of objects, allow the identification of certain classes of objects and enable the access of relevant knowledge about an object directly. Annotations can be translated into full formal logic expressions when necessary, but make it possible to work and reason with mathematical objects in a style that abstracts from the formal construction.

Secondly, annotations allow for user friendly input and output facilities. We extended Omega's input language to provide a markup for an annotated constant to indicate the type of the object it represents. For each kind of annotated constant the term parser is extended by an additional function, which parses annotations and transforms these annotations into an internal representation. During parsing additional properties can be checked and errors in the specification can be detected. In this way it is possible to extend syntactic type checking. An additional output function for each kind of annotated constant allows to have different display forms for presenting formulas to the user.

Thirdly, procedural annotations enable an efficient manipulation of annotated constants. These procedures can access information without further analysis on (lambda) terms (which define annotated constants formally) and allows to compute standard functions and relations very efficiently. These operations and properties become a computation on the data structures of annotated constants.

## 4 Annotated Terms

Annotated constants provide a mechanism to encode concrete mathematical objects as constants for the object logic and at the same time allow the identification of special objects, the storage of relevant information, and the implementation of specialised reasoning techniques. However, since the actual term is replaced by a single constant on the logic level the term is not permitted to contain variables, as these would no longer be accessible during proof construction. Nevertheless we would also like to be able to identify terms containing variables as certain types of mathematical objects. For example, in the theorem  $\forall x, y. x \neq y \Rightarrow |\{x, y\}| = 2$ , we would like to mark  $\{x, y\}$  as a finite set, and handle it appropriately during reasoning and when applying the theorem. Since we cannot use annotated constants for this, we will extend the concept to *annotated terms* by making the components of our annotated constants accessible to our object logic while retaining most of the features of annotated constants, especially that we have efficient reasoning techniques

connected to special types of mathematical objects.

#### 4.1 Modelling ‘General’ Concrete Objects

For general objects we use a tuple  $(f(a_1, \dots, a_n), t)$ , where  $f$  is an  $n$ -ary function symbol with terms  $a_1, \dots, a_n$  as arguments and  $t$  is an  $n$ -ary term that denotes the definition of  $f$ . For annotated constants it is necessary to attach relevant information as annotation to the constant, now we use  $f$  to identify the kind of object, which has  $a_1, \dots, a_n$  as its components.

##### Finite Sets:

With annotated constants we encoded the whole object  $\{a, b, c\}$  as constant of the object logic. Now we take a function symbol to identify finite sets.

**Formal term:** For the given number of elements  $n$  we use a function symbol  $S^n$  that takes the elements of the finite set as arguments. For our example the term  $S^3(a, b, c)$  is the formal representation of the finite set  $\{a, b, c\}$ .

**Definition:** In the simply typed lambda-calculus the function  $S^n$  can be defined as  $\lambda x_1, \dots, x_n, y. (y=x_1 \vee \dots \vee y=x_n)$ . The expansion of the definition yields the term  $\lambda y. (y=a \vee y=b \vee y=c)$  for our example.

As for annotated constants, the finite set is given by its elements as input by the user, a unique function symbol  $S^n$  is added to the signature during parsing. The existence of more than one function symbol is only a technical detail, as long as we can identify the  $S^n$  for arbitrary  $n$  as markup for finite sets.

The formal term and especially the function symbol do not change the underlying logic formalism. The function  $S^n$  can be seen as a place holder for the definition from the viewpoint of the object logic, but at the same time allows the identification of the category this object belongs to.

##### Concrete Matrices:

Analogous to finite sets we introduce function symbols for the identification of matrices.

**Formal term:** For a matrix of dimension  $m \times n$  there is a  $m \cdot n$ -ary function  $M^{m \times n}$  which takes the elements of the matrix as arguments. The formal term for the matrix  $\begin{pmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{pmatrix}$  is  $M^{2 \times 3}(3, 2, 7, 1, 0, 4)$ .

**Definition:** The corresponding defining term for  $M^{m \times n}$  is a double indexed function which contains all the cases for the single elements, in our example the formal term would be expanded into

$$\lambda i \lambda j. \text{ if } \quad i = 1 \wedge j = 1 \text{ then } 3 \quad \text{elseif } i = 1 \wedge j = 2 \text{ then } 2 \\ \text{elseif } i = 1 \wedge j = 3 \text{ then } 7 \quad \text{elseif } i = 2 \wedge j = 1 \text{ then } 1 \\ \text{elseif } i = 2 \wedge j = 2 \text{ then } 0 \quad \text{else } 4.$$

## 4.2 Functionality

With the change of the formal representation from constants to terms, the term can be manipulated by tactics that are not aware of the special annotation. Therefore the interpretation of the object as a data structure has to be generated from the formal term when a tactic wants to access the annotation. This is less efficient but it avoids the analysis of arbitrary terms.

For special representations containing concrete mathematical objects, we expressed trivial equalities of the annotation as syntactic equality for the formal object, because we use the same constant whenever the annotation is equal. For annotated terms we have to treat equality in the mechanism for procedural annotations as described in Sec. 3.

The motivation for the implementation of annotated constants is that certain classes of objects allow for efficient reasoning techniques, this is also true for the extended representation. Consider the simple problem  $(\{1, 2, x\} \cup \{2, 3, y\}) \subset \mathbb{Z}$  with variables  $x$  and  $y$ , which is given to the evaluation tactic. The union of finite sets is a procedure that creates finite sets consisting of the duplicate free members of the input sets. Since it is not known whether  $x = y$ , both elements appear in the resulting problem  $\{1, 2, 3, x, y\} \subset \mathbb{Z}$ . Now the subset relation for finite sets can be reduced to the element relation for all members of the finite set. This is an instance of a more general reasoning technique connected with finite sets: to show that a property holds for the elements of a finite set, check the property for every element in the set. The applicability of this technique depends on the finiteness of the set, which can be directly identified with our annotated terms. The resulting element relations  $i \in \mathbb{Z}$  for  $i \in \{1, 2, 3\}$  are trivial because integers are represented by annotated terms. However, the evaluation cannot show  $x \in \mathbb{Z}$  or  $y \in \mathbb{Z}$  and returns both as new subproblems.

With the presence of variables, we can now express unification problems on our representation. It is well-known that purely syntactic theory unification procedures are undecidable. On the other hand there exist efficient algorithms for many theories. With annotated terms we are able to identify the theory for which we have to solve a unification problem. For finite sets, for example, we can use procedures for ACI-unification [6].<sup>2</sup>

## 5 Example: Permutations

A permutation is a bijective mapping of a finite set onto itself. While there are different notations in mathematics to express permutations, the cycle notation is usually preferred and used by the computer algebra system GAP. In this notation a permutation consist of duplicate-free disjoint cycles, i.e., lists  $(n_1, \dots, n_k)$  of points with  $k \geq 1$  and  $n_i \neq n_j$  for  $i \neq j$ . A cycle maps the point  $n_i$  to  $n_{i+1}$  for  $i = 1, \dots, k-1$  and  $n_k$  to  $n_1$ . A permutation is then either

---

<sup>2</sup> Finite sets can be modelled with an operation that is associative, commutative and idempotent.

a set containing disjoint cycles or the composition of permutations. A group  $G$  can be specified by a list of generating permutations  $G = \langle p_1, p_2, \dots, p_k \rangle$  with composition of permutations as group operation.

We want to take cycles as annotation for permutations, but for the formalisation of permutations one has to decide whether cycles are just a notation for permutations or whether cycles are mathematical objects of its on right. In the first case a permutation expressed via cycles corresponds to a mapping onto itself which is bijective in the object language, for the second case an object that corresponds to cycles has to be formalised in the object language. Since properties of cycles are the subject of theorems we decide choose the second alternative.

In detail, cycles are formalised as lists, permutations as sets of cycles or composition of permutations, and both cycles and permutations are interpreted as mappings by an application operator @ that takes a permutation and a point in its domain, and returns the image. We identify cycles and permutations if their application results in the same mapping. We already introduced the annotation for finite sets, for cycles we introduce the following annotation.

**Formal term:** For each  $n$  we use a function  $C^n$  which has the elements of the cycle as arguments. For example, the term  $C^3(3, 1, 2)$  is the formal representation of the cycle  $(3, 1, 2)$ .

**Definition:** A cycle is expanded to a list with *cons* as the list constructor and with *nil* as empty list, e.g.,  $C^3(3, 1, 2) = cons(3, cons(1, cons(2, nil)))$ .

An object can be identified as permutation when it is a finite which contains disjoint cycles, e.g.  $\{(1, 2)(3, 4)\}$  or as formal term  $S^2(C^2(1, 3), C^2(3, 4))$ .

### Computations.

As procedural annotation for the application operator for concrete permutations and concrete points we used GAP. For permutations containing variables the computation of the result is not possible in general, for example, the result of  $\{(1, 2)(3, x)\}@4$  depends on whether  $x = 4$  holds or not (GAP does not even allow unbound variables in expression). Nevertheless, there are cases which can be evaluated in the same way as for constants, for example,  $\{(1, 2)(3, x)\}@x = 3$ .

The identification of permutations and the property that they contain disjoint cycles can be exploited to at least approximate results by collecting constraints. For the permutation  $\{(1, 2)(3, x)\}$  the value of  $x$  is constrained by  $x \neq 1$ ,  $x \neq 2$ , and  $x \neq 3$ . Similarly for  $\{(1, 2)(3, x)\}@4$  the result is constrained to be either 3 or 4.

### Theorems.

With variables it is now possible to formulate theorems containing annotated constants, for example the following lemma.

For all permutations  $p$  on a set  $S$  and for all pairwise disjoint points  $i, j, k \in S$  holds

$$p \circ \{(i, j, k)\} \circ p^{-1} = \{(p@i, p@j, p@k)\}.$$

For the proof we can exploit the knowledge given by our representation of permutations. To show that the permutations are equal we have to show that they return the same values on the domain  $S$ . The permutation on the right hand side only manipulates the points  $p@i$ ,  $p@j$ , and  $p@k$ . So we only have to consider the three points and a fourth case for a point in  $l \in S$  that is different from  $p@i$ ,  $p@j$ , and  $p@k$ . The cases can be shown by evaluation. For  $p@i$  the right hand side evaluates to

$$\{(p@i, p@j, p@k)\}@p@i = p@j,$$

and the left hand side to

$$p \circ \{(i, j, k)\} \circ p^{-1}@p@i = p \circ \{(i, j, k)\}@i = p@j.$$

The other cases are analogous, except for the fourth case where  $l \notin \{i, j, k\}$  has to be used.

Now that the lemma is available we can apply in other proofs, which are formulated for annotated terms without the necessity to expand our representation to its definition. For instance, in the proof that all cycles with three elements of the alternating groups  $\mathbf{A}_n$  with  $n \geq 5$  are conjugated.

## 6 Conclusion

With annotated terms we attach semantic information to terms. This allows us to distinguish between mathematical objects for which efficient computational algorithms exist and objects which have to be treated purely by deduction. The criterion for this distinction is the *form* in which the object is given and not the properties of the object. For example, we differentiate between finite sets where the elements are given explicitly from other formalisations. In the object logic it is not possible to define a predicate (i.e., a sort) that distinguishes between finite sets given in form of their elements from other representations of finite sets. So the distinction cannot be expressed inside the object logic but it is necessary to express it as extra-logical annotation.

The only other approach that does not leave the formal language is to formalise the data structure for special objects itself and its interpretation as theory in the object logic. So all manipulations on the data structure have to be explicitly performed and justified by proofs. Unless the proof system supports a high degree of automation for data structures this can be a tedious task. In our approach we only have to *reconstruct* the operation for the formal object, which is usually easier than to perform the manipulation itself. An example for a framework with a high degree of automation for data structures is the Calculus of Inductive Construction [5] implemented in the COQ system [8], where inductively defined operations can be executed without proof obligations. The advantage of our approach is that it does not depend

on a specific formal system.

There exists related work in which computation is integrated into formal reasoning, for example, the integer arithmetic in the type theory of NuPRL [9], and evaluation for functions for certain terms in the automated theorem prover Otter [12]. We applied this idea to other classes of objects and operations on these objects and think that the possibility to introduce new classes is an important feature to model the flexibility in mathematical representations. In contrast to these systems our evaluation does not extend the formal system and therefore does not influence the correctness. Our annotations are used to ease the construction of an abstract proof, but require verification on the object-level.

Other approaches for the integration of computational algorithms into theorem proving, which make use of existing systems like MAPLE, are usually focused on arithmetic operations and problems from real analysis [2,7]. The integration of results into the theorem proving system is eased by the correspondence between the formalisation in the theorem prover and the language of computer algebra systems, for example, MAPLE can deal with variables, arithmetic terms can be identified by their type, and sub-term consistency with respect to the signature represented in the computer algebra system holds.<sup>3</sup> Thus it is sufficient to consider only the correspondence between the symbols in the signature of the theorem prover and relate them to objects in the computer algebra system. This cannot be expected in general, for example, the computer algebra system GAP does not allow unbound variables, and is only able to perform computations with concrete structures. With annotations we are able to express finer categorisations of computational objects and operations which can be exploited for the communication with computer algebra systems.

A common observation for the formalisation of mathematics is, that there does not exist a single best formalisation, but that there are several possible ones, which are suitable for different purposes. With annotated terms, we don't have to make a decision. We can use a straight-forward encoding for the formal representation while having alternative representations available in form of the annotation.

## 7 Future Work

We have presented the extension of annotated constants to annotated terms and motivated this representation with examples that demonstrate how the knowledge about special kinds of objects can be exploited for computation and reasoning. We have successfully shown the usefulness of annotated constants in a large case study in certifying solutions to permutation group problems [4]. As next step the extension to annotated terms has to be evaluated in an ex-

---

<sup>3</sup> A term algebra  $T_\Sigma$  is called consistent with respect to the signature  $A$ , iff for all  $f \in A$  any term  $f(t_1, \dots, t_n) \in T_\Sigma$  already lies in the term algebra  $T_A$  [3].

tensive case study as well. In particular, it has to be investigated whether the proportion of proof obligations, which are solvable by computation, justifies the overhead caused by the implementation of the evaluation mechanism for annotated terms. With the presence of theorems containing annotated terms the question arises, whether it is possible to maintain the abstract representation during proofs, or if it is necessary to expand to the formal representation. We expect that our representation is especially beneficial for automated proof search in the context of proof planning.

Furthermore, we want to integrate our representation for ellipses which we have developed for matrices also for other objects and especially for objects containing variables. With ellipses we could formulate the conclusion of the lemma on cycles from a above more generally, namely as

$$p \circ \{(i_1, \dots, i_n)\} \circ p^{-1} = \{(p@i_1, \dots, p@i_n)\}.$$

The object  $i_1, \dots, i_n$  does not only contain variables but can be interpreted as a sequence variable itself. The application of techniques developed for sequence variables [10] can therefore be beneficial and should be investigated in future work.

## References

- [1] P. B. Andrews. *An Introduction To Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer, 2nd edition, 2002.
- [2] C. Ballarin, K. Homann, and J. Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In A. H. M. Levelt, editor, *Proc. of the 1995 International Symposium on Symbolic and Algebraic Computation (ISSAC-95)*, pages 150–157, Montreal, Canada, July 10–12 1995. ACM Press, Berkeley, CA, USA.
- [3] J. Calmet and K. Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, pages 221–234. Kluwer Academic Publishers, 1996.
- [4] A. Cohen, S. H. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. In *Proc. of CADE-19*, volume 2741 of *LNAI*, pages 258–273. Springer, 2003.
- [5] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proc. of Colog'88*, volume 417 of *LNCS*. Springer, 1990.
- [6] A. Dovier, E. Pontelli, and G. Rossi. Set unification. *CoRR*, cs.LO/0110023, 2001. <http://arxiv.org/abs/cs.LO/0110023>.
- [7] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *J. of Automated Reasoning*, 21(3):279–294, 1998.

- [8] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant - Version 8.0*, Apr. 2004. <http://coq.inria.fr>.
- [9] C. Kreitz. *The Nurpl Proof Development System, Version 5*, Dec. 2002. <http://www.cs.cornell.edu/Info/Projects/NuPr1/>.
- [10] T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Proc. of AICS'2002 & Calculemus'2002*, volume 2385 of *LNAI*. Springer, 2002.
- [11] S. Lang. *Algebra*. Addison-Wesley, 2nd edition, 1984.
- [12] W. McCune. *Otter 3.3 Reference Manual*, Aug. 2003. <http://www-unix.mcs.anl.gov/AR/otter/>.
- [13] Omega Group. Proof development with Omega. In *Proc. of CADE-18*, volume 2392 of *LNAI*, pages 143–148. Springer, 2002.
- [14] M. Pollet and V. Sorge. Integrating computational properties at the term level. In *Proc. of Calculemus'2002*, pages 78–83, 2003.
- [15] M. Pollet, V. Sorge, and M. Kerber. Intuitive and formal representations: The case of matrices. In *Proc. of MKM 2004*, volume 3119 of *LNCS*. Springer, 2004.