

Integrating Computational Properties at the Term Level

Martin Pollet^{1*} and Volker Sorge^{2**}

¹ Fachbereich Informatik, Universität des Saarlandes, Germany,
pollet@ags.uni-sb.de, <http://www.ags.uni-sb.de/~pollet>

² School of Computer Science, University of Birmingham, UK,
V.Sorge@cs.bham.ac.uk, <http://www.cs.bham.ac.uk/~vxs>

1 Introduction

Human mathematicians often use representations for particular mathematical concepts that allow them to remember properties of the concepts. After its initial introduction or construction the further use of a concept abstracts from its construction. Moreover, special representations are used that are suitable for particular reasoning strategies, for example multiplication tables, matrices and diagrams [5].

On the contrary 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, in lambda calculus sets are usually represented as lambda terms containing a disjunction of equalities. For example, a set of the form $\{a, b, c\}$ is represented as $\lambda x. (x=a \vee x=b \vee x=c)$. While these representations are theoretically suitable for reasoning about properties of the abstract mathematical concept they are often a hindrance when dealing with concrete objects in practice. The representations are often very different from the informal mathematical vernacular. Furthermore they are typically also less suited for direct computations and cannot be directly passed to a computer algebra system. This has the disadvantage that identification and interpretation of terms has to be implemented in the interface between deduction and computation.

We present the notion of annotated constants as an abstraction over the construction of concepts. It enables abstract, concise representation of a mathematical object together with implicit handling of its computational properties inside a theorem proving system. An annotated constant replaces the functional expression of an object such as a set or a list. It is treated as a constant of the formal language by the prover, but it is associated with a datastructure that contains a representation of the object, which is suitable input for special tactics or computer algebra systems. Moreover, annotated constants distinguish particular objects from regular constants; a fact that can be exploited for input and display purposes as well.

With annotated constants, trivial properties of concrete objects are already implemented on the term level. Moreover, they ease the detection of equality

* The author's work is supported by EU IHP grant Calculemus HPRN-CT-2000-00102.

** The author's work was supported by a Marie-Curie Grant from the European Union.

between objects and abstract from certain proof obligations needed to establish necessary properties of the objects in question. Nevertheless annotated constants do not extend the formal language of the theorem prover as they can be expanded to their formal definitions on a more primitive term level and their required properties are rigorously checked.

We emphasise that annotated constants are a pragmatic approach to the representation of some concrete mathematical objects inside a theorem prover and are not a theoretical framework to encode semantic or heuristic informations such as existing formalisms which include annotations [4] or labels [2]. Moreover, their introduction does not extend the underlying formalism of the theorem prover as for instance the extension of type theory to inductive types does (cf. [7]). Finally, our approach aims at facilitating the application of external computer algebra systems inside the prover. Therefore, the objects are usually represented in a form that can be directly used as input for a computer algebra system, which, as a side-effect, also facilitates a human-oriented presentation of the objects. In particular, we do not intend to implement optimised datastructures for efficient computations (cf. [8]) from computer algebra itself.

2 Annotated Constants

Suppose we have a formal language \mathcal{L} , then an annotated constant is a triple (k, \mathbf{a}, t) , where k is a constant of \mathcal{L} , \mathbf{a} is the annotation, and t is a term in \mathcal{L} that is the formal definition of k .

The *annotation* \mathbf{a} can be an arbitrary datastructure that may contain other terms (without free variables) of the language \mathcal{L} and must fulfil the property that the *constant* k can be identified and the *defining term* t can be generated from the annotation. The datastructure for annotations is designed in such a way that particular relevant information about the objects becomes directly accessible, either to allow tactics to access their information, or to ease the communication with external systems, such as computer algebra systems. Annotations allow to identify different classes of mathematical objects not only for tactics but also to have a special display presentation.

The constant k is the formal representation and part of the language \mathcal{L} . With this representation we can map the properties of the annotations into the formal system, namely, that two annotations \mathbf{a} and \mathbf{a}' are equal if their associated constants are identical.

Annotated constants are introduced for numbers, lists, tuples, sets, and cycles. We describe the latter two in more detail.

Sets. Finite sets have a special notation in the mathematical vernacular, for example $\{a, b, c\}$. The information connected with this representations is, that it is a set, it contains finitely many elements, and the elements are explicitly given. Usually finite sets with different ordering of the elements, for example $\{a, b, c\}$ and $\{b, a, c\}$, are trivially identified. We tried to capture these properties with annotated constants for finite sets. The annotated constant allows to access the elements of a set without further analysis on lambda terms (the defining term

for the constant) and already implements the equality for sets which differ only in the ordering of their elements.

Annotation for finite sets: The datastructure of sets (unordered lists), the elements of the sets are terms of the formal language, e.g., $\{b, a, c\}$ with $a, b, c \in \mathcal{L}$.

Constant: The identifier for the formal constant is generated from a duplicate free ordering of the elements of the set, for the example $k_{\{a,b,c\}} \in \mathcal{L}$.

Definition: The ordering of the elements of the set that is the annotation is also used to construct a lambda term as definition, e.g., $\lambda x.(x=a \vee x=b \vee x=c)$.

Cycles. A permutation is a bijective mapping of a finite set onto itself and is often given in cycle notation, for example, the permutation defined by the cycle $(1\ 2\ 3)$ acting on the set $\{1, 2, 3\}$ maps 1 to 2, 2 to 3, and 3 to 1. The elements of a cycle have to be duplicate free, this property is verified during parsing and allows to detect mistakes already in the input specification (see Section 2.1). Annotated constants for cycles implement a basic equality on cycles, that is cycles are identified if their elements are only shifted, for examples, $(1\ 2\ 3)$ and $(2\ 3\ 1)$ are equal. For the definition of annotated constants for cycles we use a representation that is similar to the (input) format of the computer algebra system GAP.

Annotation for cycles: The datastructure of lists, when the elements are in \mathcal{L} and of type integers. Additionally the list must be duplicate free, e.g., $(k_3\ k_1\ k_2)$ with constants $k_3, k_1, k_2 \in \mathcal{L}$ which are the constants of the annotations 3, 1, 2 representing integers.

Constant: A constant representing the cycle in a normal form, that is, the minimal element of the cycle is shifted to the first position, e.g., $k_{(k_1\ k_2\ k_3)} \in \mathcal{L}$.

Definition: The term representing the shifted cycle constructed with *nil*, *cons* $\in \mathcal{L}$, e.g., $t_{(k_1\ k_2\ k_3)} = \text{cons}(k_1, \text{cons}(k_2, \text{cons}(k_3, \text{nil})))$.

The definition of annotated constants includes the identification of the corresponding constant from its annotation. This identification lies outside of the formal system and can therefore be a possible source for errors. The verification of tactic applications which use the information provided by annotations, will be explained in Section 2.3.

2.1 Implementation of Annotated Constants

The basic functionality for handling annotated constants is implemented on the term level of the Omega system [6]. An annotated constant is essentially similar to regular constants: it has the datastructure it denotes as name and the same type as the expression (i.e. the defining term t) it represents. It can also be replaced by its defining term during the proof or when expanding the proof to check formal correctness. (The latter step is explained in more detail in Sec. 2.3.) Annotated constants do not have to be explicitly specified in the signature of the proof and their defining term is only computed when necessary.

We extended Omega's input language to provide markup for annotated constants to indicate the type of the object it represents. For each kind of annotated

constant the term parser has to be extended by an additional function. This allows to specially parse annotations and to immediately transform them into a normal form representation. During parsing additional properties can immediately be checked and errors in the specification can be detected, for example that the cycle is duplicate free. In other words the check for pure syntactic correctness via type checking can be enriched by additional functionality to verify certain semantic properties of the input. An additional output function for each kind of annotated constant allows to have different display forms for presenting formulas to the user.

2.2 Manipulating Annotated Constants

Annotated constants are usually manipulated using specialised tactics, which can directly operate on the datastructures comprising the annotations. These datastructures are deliberately chosen to reflect the intuitive representation and to ease the communication with external systems by using their input representation as annotation.

The computations themselves are either implemented in the programming language underlying Omega or are performed using external computer algebra systems. For instance, functions on integers such as addition, multiplication, etc., are simply mapped to their counterpart in the Lisp programming language in which Omega is implemented. We have one tactic that simplifies expressions containing integers by executing the appropriate Lisp functions.

On the other hand operations on cycles such as their application or the composition of two cycles are executed using tactics that call the computer algebra system GAP [3]. The results are then directly incorporated into the proof. Since our notation of cycles is similar to the one used in GAP the translation is straightforward.

We use annotated constants in a case study in which they substantially contribute to the abstraction and simplification of proofs. The case study itself is concerned with the verification of computations on permutation groups performed by the computer algebra system GAP with the help of the proof planner of Omega. Permutations are formalised as sets of cycles and concrete permutations are represented by annotated constants. Due to lack of space we refer to [1] for details.

2.3 Guaranteeing Correctness

To guarantee correctness for a proof in Omega the tactics justifying single proof lines have to be expanded to a machine-checkable calculus level. Thereby it is often necessary that annotated constants are substituted by their defining terms, for instance, to verify single computational steps. Moreover, additional properties on the annotated constants need to be checked, if there are any. In particular, those properties that have been checked informally during parsing or generation of an annotated constant have to be painstakingly verified at the logic level.

For example, for cycles it is always crucial to verify that they are duplicate free lists of integers. This is achieved by recursively checking that a predicate *cycle* holds. For the concrete cycle (1 2 3) the first two steps of this verification are: $L_1 : cycle(k_{(1\ 2\ 3)})$, $L_2 : cycle(cons(1, cons(2, cons(3, nil))))$, $L_3 : 1 \notin \{2, 3\} \wedge cycle(cons(2, cons(3, nil)))$. In the first step the constant $k_{(1\ 2\ 3)}$ in line L_1 is replaced with its defining term $cons(1, cons(2, cons(3, nil)))$. The second step is the expansion of the cycle predicate in L_2 which yields two new proof obligations in L_3 : to show that 1 is not an element of $\{2, 3\}$ and that (2 3) is again a valid cycle.

3 Conclusion and Future Work

With annotated constants we have introduced a technique to attach information to objects implemented as constants in the formal language. This extension does not change the formal system and has therefore no influence on the correctness. We have currently implemented annotations for some special types of mathematical objects (numbers, lists, sets, tuples, and cycles), however the extension for other special representations is straightforward. An interesting case are functions and operations which can be evaluated when they are applied to annotated constants. The operations of intersection and union could have computational information annotated to be used when they are applied to concrete sets. With functional annotations and arguments, one could think about annotations for terms, which are the result of the evaluation.

We tried to capture some aspects of mathematical representations with annotated constants. There are further issues to be considered, for example, how can different representations of the same concept be expressed. This is especially important when the key step of a proof is to use a suitable representation of the problem or if different representations are necessary for efficient computations. Naturally the use of various, possibly interchangeable, representations has substantial implications for the underlying prover and its formal system. Robustness of the proving techniques will need to be ensured by designing tactics, planning methods, appropriate matching algorithms, etc. that can handle different representations of the same mathematical object. Moreover, guaranteeing soundness, in particular for equality, will be less trivial than for annotated constants.

References

1. A. Cohen, S. H. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. *Proc. of CADE-19*, LNAI, 2003. Springer Verlag. to appear.
2. D. Gabbay. *Labelled Deductive Systems – Vol. 1*. Number 33 in Oxford Logic Guides. Oxford University Press, 1996.
3. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.3*, 2002. <http://www.gap-system.org>.
4. D. Hutter. Annotated reasoning. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):183–222, 2000.
5. M. Kerber and M. Pollet. On the design of mathematical concepts. Technical Report CSRP-02-06, The University of Birmingham, School of Computer Science, 2002.

6. The Omega Group. Proof development with Omega. *Proc. of CADE-18*, vol. 2392 of *LNAI*, pages 143–148, 2002. Springer Verlag.
7. Coq Development Team. The coq proof assistant reference manual, 2002.
8. R. Zippel. *Effective Polynomial Computation*. Kluwer Academic Press, 1993.