Structuring and Using a Knowledge Base of Mathematical Concepts: A Type-Theoretic Approach

Martin Strecker, Marko Luther, Matthias Wagner
{strecker/luther/wagner}@ki.informatik.uni-ulm.de
Universität Ulm,
Abteilung Künstliche Intelligenz
D-89069 Ulm,
Germany

Abstract. This paper describes an approach to representing mathematical concepts in a knowledge base which is structured by a subsumption relation between concepts. Two kinds of concepts are examined: Propositional concepts, with the subsumption relation given by a generalized implication, and parameterized theories, with the subsumption relation given by theory morphisms. It is shown which kinds of reasoning activities can be supported by such a knowledge base. A type theory in which the entities to be represented are first-class objects serves as formal framework.

Keywords:
Knowledge representation; concept languages; automated and interactive theorem proving; type theory

ECAI 96. *12th European Conference on Artificial Intelligence*
Workshop on Representation of Mathematical Knowledge
Edited by H. Stoyan, K. Homann, S. Jacob, M. Kerber

1 Introduction

Research in the automation of mathematical reasoning has to a large extent been dominated by approaches which emphasize powerful, low-level inference procedures, but usually ignore conceptual knowledge about the entities involved in the reasoning activity.

The work described here explores an approach to computer-assisted formal reasoning in which more emphasis is laid on the entities occurring in a mathematical statement or a proof, and in which knowledge about the relations between these entities plays a crucial role. This approach is inspired by knowledge representation formalisms developed in AI, in particular concept languages (terminological logics). *Concepts* are (usually intensionally defined) classes of entities; *instances* are entities belonging to the extension of one or several concepts. Concepts can be arranged along a *subsumption relation* in taxonomies which, by means of inheritance, permit an efficient transfer of properties known to hold in super-concepts to sub-concepts. Section 3 will briefly present *propositional concepts*, which closely resemble the concepts used in terminological logics. Section 4 will show how the notion of "concept" can be extended to *parameterized mathematical theories*, with the subsumption relation given by theory morphisms. In this latter case, the subsumption relation not only permits to interpret theorems of one theory in another, more specific theory. It also provides valuable information on applicability conditions of a theorem and on preconditions required to establish a theorem, thus leveling the ground for proof abstraction.

Even if the formalism presented here does not pretend to capture all interesting mathematical phenomena, it might be one among several frameworks in which fragments of mathematical knowledge can be represented in a cognitively adequate manner and used to support mathematical reasoning. The work reported in this paper is still in progress. A first experimental implementation in the specification and verification environment TYPELAB [vHLSW96] is currently under way. For reasons made more precise in section 2, type theories seem to be particularly suited to express the dichotomy of concepts and instances of concepts. However, using a type theory is not essential; the principles of this approach carry over to other logics and frameworks, for example HOL [GM93], Isabelle [Pau94], and PVS [SOR93].

2 Type-Theoretic Preliminaries

Type theories are extensions of the lambda calculus, in which types are associated with terms, given certain hypotheses. The *typing judgement* $\Gamma \vdash t : T$ expresses that term t has type T under the assumptions Γ, where Γ is a list of declarations of the form $x : A$ for variable x and type A.

The particular type theory this work is based on is the *Extended Calculus of Constructions (ECC)* [Luo94], which has been chosen for its great expressiveness. It has as type constructors dependent function types, useful for expressing polymorphism and parameterization of specifications, and dependent sum types, useful for representing specifications and mathematical theories. By the propositions-as-types paradigm, propositions can be understood as types, and the usual logical connectives and quantifiers can be coded in ECC. The elements of propositional types can be interpreted as proofs of the corresponding proposition.

Most of the type-theoretic apparatus can be hidden under a syntactic cover and could even be replaced by another formalism. For our purposes, the following aspects of the type theory ECC are important:

- Many entities occurring in mathematical practice can be represented as internal objects of the formalism, in particular (higher-order) functions, propositions, mathematical theories and mappings between theories (theory morphisms).
- In type theory, there is a natural distinction between classes of entities represented by types and incarnations of classes represented by elements of types. "Classes" in this sense are entities as diverse as logical propositions and mathematical theories, with "incarnations" given by evidence that a proposition holds and by models of a theory, respectively.

These general observations will be extended and illustrated in the following sections.

3 Propositional Concepts

Most knowledge representation formalisms developed in AI can be interpreted logically. In particular, research in terminological knowledge representation has focussed on identifying fragments of first-order logic for which tractable inference algorithms exist [Neb90]. "Concepts" are defined to be unary predicates, "roles" binary relations. New concepts resp. roles can be defined from primitive ones using a limited set of concept and role forming operators.

The notion of "concept" to be used in the following is not subject to a similar syntactic restriction. Rather, a *concept* is taken to be an arbitrary n-ary relation. *Subsumption* is defined as a generalized implication. As a first approximation, concept C_1 subsumes C_2 (i.e., C_2 is a subconcept of C_1) if

$$(*) \quad \forall\, x_1 : T_1, \ldots, x_n : T_n.\ C_2(x_1, \ldots, x_n) \rightarrow C_1(x_1, \ldots, x_n)$$

holds. In view of the subsumption relation required in section 4, we make one more generalization: We allow functions f_i to operate on the parameters of concepts, which gives the following definition for subsumption of C_2 by C_1:

$$\begin{aligned}(\dagger) \quad & \forall\, x_1 : T_1, \ldots, x_n : T_n. \\ & C_2(x_1, \ldots, x_n) \rightarrow C_1(f_1(x_1, \ldots, x_n), \ldots f_m(x_1, \ldots, x_n))\end{aligned}$$

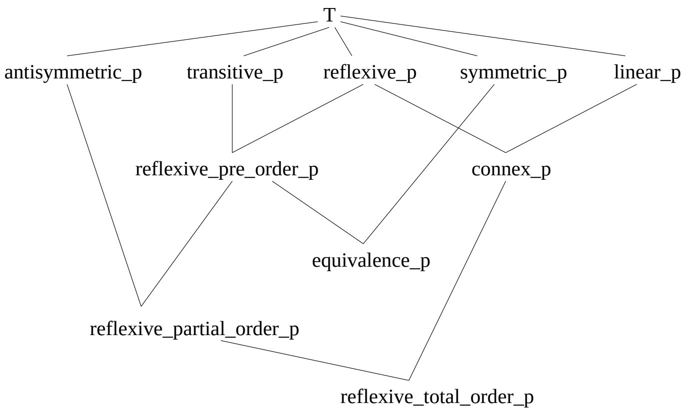

Figure 1. (Excerpt of a) taxonomy of predicates over homogeneous relations. A line connects subsumers and subsummees, the more general concepts are above the more specific ones.

Figure 1 gives an example of a taxonomy of mathematical concepts, in this case predicates describing properties of relations. The following are typical definitions of concepts in this taxonomy:

$$\begin{array}{lcl} \mathit{symmetric\_p} & := & \lambda T : \mathit{Type}, R : \mathit{Relation}(T). \\ & & \forall (x, y : T) R(x, y) \rightarrow R(y, x) \\ \mathit{equivalence\_p} & := & \lambda T : \mathit{Type}, R : \mathit{Relation}(T). \\ & & \mathit{reflexive\_p}(T, R) \wedge \mathit{transitive\_p}(T, R) \\ & & \wedge \mathit{symmetric\_p}(T, R) \end{array}$$

Note that these concepts are higher-order, since they take relations as arguments, and polymorphic, since they are quantified over an arbitrary type T. In general, any n-ary predicate can be regarded as a concept definition.

Classification is the process of constructing a taxonomy, given a set of concept definitions. Due to the generality of the concepts involved, the subsumption relation is not decidable, and a complete classification algorithm does not exist. It can however be observed that many subsumption relations between concepts are purely definitional, as for *symmetric_p* and *equivalence_p*. By a process of pre-classification, these easy-to-detect subsumption relations can be determined (see [Lut95] for a more detailed description). Further subsumption relations can be discovered by a general proof search procedure or by specialized tactics (see [Wag95] for a description of proof facilities of the TYPELAB system). Lastly, subsumption relations which are too difficult to find automatically can be inserted manually into the knowledge base, their correctness can later be established by an interactive proof.

Terminologies, which have been discussed so far, provide a conceptual framework for an application domain (cf. the notion of TBOX in terminological languages like KL-One). A terminology is constructed once, stored in a library and can subsequently be used to support reasoning activities in the application domain. In particular, "instances" can be attached to the concepts, and the structure of the terminology can be exploited to infer properties of these instances by a kind of forward-chaining. The set of instances forms the "assertional" part (or ABOX) of a knowledge base.

An *instance* of a concept C is here defined to be an n-tuple of values $(a_1, \ldots, a_n)$ such that $C(a_1, \ldots, a_n)$ holds, together with evidence for this fact. In a constructive type theory like ECC, this evidence is given by a proof term prf such that $\Gamma \vdash prf : C(a_1, \ldots, a_n)$ is a valid judgement. However, another notion of "evidence", such as a tactic, could also be appropriate. An instance of a concept can easily be interpreted as an instance of one of its super-concepts: In type theory, the proof of a subsumption relation such as $(*)$ yields a function h which transforms the proof term prf into a proof term of an instance of the superconcept. A tactic applying a generalization of Modus Ponens has a similar effect and could equally well serve as evidence.

As an example of the kind of reasoning supported by such a knowledge base, assume that the transitivity of the relation $\leq$ of a lattice has to be proved. By definition, this relation is a reflexive partial order. Thus, an instance $\langle d, (L, \leq)\rangle$ of the concept *reflexive_partial_order_p* is created and entered into the knowledge base (here, d represents the statement made by the definition, L is the carrier of the lattice). By propagating this instance upwards in the taxonomy, an instance $\langle p, (L, \leq)\rangle$ of the concept *transitive_p* is created, from which the proof p of the desired property can directly be read off. Note that it is not necessary to unfold the definitions and redo the proof in detail. Rather, previously established knowledge is directly applied in a form which is quite similar to the way humans make inferences on the basis of their mathematical background knowledge.

A first experimental implementation of the knowledge-based reasoning component described above has been integrated in the TYPELAB system and can be used in interaction with the prover of TYPELAB [Wag95]. For a more detailed discussion of technical aspects, the reader is referred to [Lut95].

4 Theories as Concepts

The idea to make mathematical theories (or "specifications" in programming) and mappings between them ("theory morphisms") a topic of study is present in theoretical work [BG77, Gog86] and is manifesting itself in systems like IMPS [FGT93], OBJ [GW88], PVS [SOR93] and Specware [SJ94]. The TYPELAB system takes these approaches a step further by representing theories and theory morphisms as internal objects of the logic. We expect that this leads to a more uniform treatment of certain phenomena. In particular, the notions of "concept", "subsumption" and "instance" that have been developed in section 3 can be extended to mathematical theories, so that some of the mechanisms presented there can be reused.

Consider, for example, the following parameterized theory `LIST_REC_MONOID` which is an extension of a theory `LIST` of lists in which the usual functions (`nil`, `cons`, `append` etc.) are defined. The theory has two parameters which are themselves theories: A theory `MONOID` of monoids, and a theory `ELEM` of arbitrary nonempty domains. `LIST_REC_MONOID` introduces a recursor `rec` over lists which maps the empty list to the unit of the monoid and combines the elements of a non-empty list with the monoid operation `op`. The theorem `rec_distr_append` can be proved in this theory. It states that the recursor "distributes" over the `append` function.

```
LIST_REC_MONOID := fun(M:MONOID) fun(E:ELEM)
   EXTEND (LIST E) BY
   rec: List -> M.T,
   f: E.T -> M.T,

   AXIOM rec_nil: (rec nil) = M.unit,
   AXIOM rec_cons:
      all(x:E.T,l:List)
      (rec (cons x l)) = (M.op (f x) (rec l)),

   THEOREM rec_distr_append:
      all(l1,l2:List)
      (rec (append l1 l2)) = (M.op (rec l1) (rec l2))
   END;
```

From a technical viewpoint, parameterized theories are functions with n parameters mapping into a dependent sum type. Thus, the notion of concept developed in section 3 can be adopted with minor modifications, the definition (†) of subsumption then directly carries over to parameterized theories. Now we can identify, among others, the following concepts and subsumption relations:

- `MONOID` subsumes the theory `NAT` of natural numbers, by interpreting 0 as `unit` and addition as `op` of `MONOID`.
- `MONOID` subsumes the theory `LIST`, which is parameterized over theory `ELEM`, by interpreting `nil` as `unit` and `append` as `op` of `MONOID`.
- The theory `MONOID` subsumes the theory `C_MONOID` of commutative monoids.
- Suppose the theory `LIST_REC_C_MONOID` is defined like `LIST_REC_MONOID`, but by abstracting over `C_MONOID` instead of `MONOID`. In this theory, we can prove, for example, that the recursor `rec` applied to a list yields the same result as `rec` applied to a permutation of the list. Since we have a covariant subsumption relation, `LIST_REC_MONOID` subsumes `LIST_REC_C_MONOID`.

Figure 2 shows a taxonomy built upon these and other subsumption relations.

We are interested in the following kind of reasoning activities to be supported by a knowledge base:

- Inheritance of theorems by projecting a theorem that is known to hold in a theory (e.g. `MONOID`) down to more specific theories (e.g. `NAT` or `LIST`). This approach has already been explored in IMPS [FGT93], although only for non-parameterized theories.
- Selection of theorems which can be used for discharging a proof obligation. This eventually requires finding appropriate instances of parameter theories and completing partial models of theories. This is best illustrated by the following example: Suppose we want to prove for lists `l1, l2` with elements of an arbitrary non-empty type `T` that

$$(\text{length } (\text{append } l1\ l2)) = (+\ (\text{length } l1)\ (\text{length } l2))$$

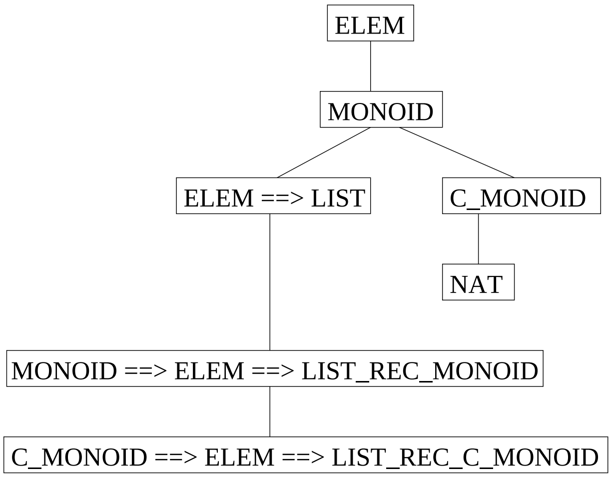

Figure 2. Taxonomy of mathematical theories. Parameterized theories are displayed together with their parameter theories (those before the `==>`)

holds. Note that this is just an instance of the scheme described by the theorem `rec_distr_append`: In this case, the length function "distributes over" the append function. Thus, we can expect to use the theorem `rec_distr_append`, after carrying out the following steps:

- We have to find appropriate instances of the parameter theories of `LIST_REC_MONOID`: The instance of theory `NAT` that supplies the operation `+` is interpreted as an instance of `MONOID`. The non-empty type `T` can be interpreted as an instance of `ELEM`.
- We have to complete the partial model of `LIST_REC_MONOID` by constructing realizations of `rec` and the function `f` such that the axioms `rec_nil` and `rec_cons` are satisfied. By choosing the function `fun(x:E.T) 1` for `f` and `length` for `rec`, we obtain the desired model (note that, by definition of the `length` function, `(length nil) = 0` and `(length (cons x l)) = (+ 1 (length l))`).

At first glance, this might appear as a roundabout way of proving a theorem. Note, however, that this proof does not appeal to an inductive argument which would be required if the theorem were proved from scratch. Once the realizations of `rec` and `f` have been constructed, the remaining proof obligations are trivial.

- Abstraction of theorems and proofs: Suppose that the theorem `rec_distr_append` had been proved in the theory `LIST_REC_C_MONOID`. An analysis of the proof would reveal that some of the preconditions of the theorem, provided by the parameter theory `C_MONOID`, are not necessary for the proof to go through. The preconditions could then be weakened to parameter theory `MONOID`, the theorem could be abstracted and transferred to theory `LIST_REC_MONOID`.

In this section, some forms of reasoning with and about theories have been informally discussed. Even though the general principles seem to be clear, much more research is required to establish a formal basis and arrive at practically feasible solutions. Currently, work is in progress to integrate these ideas into the TYPELAB system.

REFERENCES

BG77 R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. *Proceedings IJCAI*, pages 1045–1058, 1977. Description of the specification language CLEAR.

FGT93 William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *J. of Automated Reasoning*, 11:213–248, 1993.

GM93 M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

Gog86 J.A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, February 1986.

GW88 Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.

Luo94 Zhaohui Luo. *Computation and Reasoning*. Oxford University Press, 1994.

Lut95 Marko Luther. Wissensbasierte Methoden zur Beweisunterstützung in Typentheorie. Master's thesis, Universität Ulm, 1995. Electronically available at URL http://www.informatik.uni-ulm.de/ki/Forschung/Deduktion/ml-diplomarbeit.html.

Neb90 Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *LNAI*. Springer, 1990.

Pau94 Larry Paulson. *Isabelle - a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.

SJ94 Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, 1994. Updated version appeared in Proc. of the Conference on Mathematics of Program Construction, Kloster Irsee, Germany, July 1995.

SOR93 N. Shankar, S. Owre, and J. M. Rushby. A tutorial on specification and verification using PVS. Technical report, SRI International, Menlo Park CA 94025 USA, 1993.

vHLSW96 F.W. von Henke, Marko Luther, Martin Strecker, and Matthias Wagner. The TYPELAB specification and verification environment. In *Proceedings of AMAST'96*, 1996.

Wag95 Matthias Wagner. Entwicklung und Implementierung eines Beweisers für konstruktive Logik. Master's thesis, Universität Ulm, 1995. Electronically available at URL http://www.informatik.uni-ulm.de/ki/Forschung/Deduktion/mw-diplomarbeit.html.