Mathematical Vernacular and Conceptual Well-formedness in Mathematical Language*

Zhaohui Luo and Paul Callaghan

Department of Computer Science, University of Durham,
South Road, Durham DH1 3LE, U.K.
{Zhaohui.Luo, P.C.Callaghan}@durham.ac.uk

Abstract. This paper investigates the semantics of mathematical concepts in a type theoretic framework with coercive subtyping. The type-theoretic analysis provides a formal semantic basis in the design and implementation of Mathematical Vernacular (MV), a natural language suitable for interactive development of mathematics with the support of the current theorem proving technology.
The idea of semantic well-formedness in mathematical language is motivated with examples. A formal system based on a notion of conceptual category is then presented, showing how type checking supports our notion of well-formedness. The power of this system is then extended by incorporating a notion of subcategory, using ideas from a more general theory of coercive subtyping, which provides the mechanisms for modelling conventional abbreviations in mathematics. Finally, we outline how this formal work can be used in an implementation of MV.

1 Introduction

By *mathematical vernacular* (MV), we mean a mathematical and natural language which is suitable for developing mathematics, has a formal semantics, and is implementable for interactive mathematical development based on the technology of computer-assisted formal reasoning and Natural Language Processing. In a research project on MV at the University of Durham, a type-theoretic approach is being considered: we are developing an MV language together with its type-theoretic semantics and the associated techniques to implement MV based on existing theorem proving technology. Such technology is represented by type theory based proof development systems such as ALF [MN94], Coq [Coq96], and Lego [LP92].

One of the motivations of our work reported here is to study the *linguistic structure* of mathematical language based on which MV is to be designed and implemented. In particular, we are interested in how various linguistic aspects

* This work is supported partly by the Durham Mathematical Vernacular project funded by the Leverhulme Trust (see http://www.dur.ac.uk/~dcs7ttg/mv.html) and partly by the project on Subtyping, Inheritance, and Reuse funded by UK EPSRC (GR/K79130, see http://www.dur.ac.uk/~dcs7ttg/sir.html).

of mathematical language, such as its conceptual structure, may be analysed in a framework of constructive type theory. Note that, though being a semantic language in our discussion, a type theory is a syntactic language manipulated by a proof development system. Therefore, a type-theoretic analysis of the linguistic structure is an important step in the development and implementation of MV.

In this paper, we study the conceptual structure of mathematical language by studying a notion of conceptual category [NPS90, Luo94]. An important issue in this analysis is that of well-formedness and meaningfulness of expressions in MV. Mathematicians attach importance to the criterion of semantic well-formedness, as well as to grammatical well-formedness. Conceptual categories play an important role not only in correctness checking (ie, deciding whether an expression or a sentence is well-formed and meaningful) but also in capturing the generative nature of concept composition in MV.

Focussing on the conceptual structure reflected in the use of substantives (or common noun phrases) and adjectives – the key linguistic entities used to represent mathematical concepts (cf., [dB94]), we consider fine-structured categories that allow advanced treatment of deciding whether an expression is well-formed. With our type-theoretic semantics, checking semantic well-formedness is supported by type-checking (MV will be implemented around a type-checker) on the one hand, and helps to produce proof obligations in the interactive process of using MV, on the other.

Another contribution of this paper in studying the conceptual structure of MV is the treatment of the inheritance relation between conceptual categories based on *coercive subtyping* – a general theory of subtyping and inheritance in type theory [Luo97, Luo98, SL98]. It is interesting to see that coercive subtyping, a theory developed in a rather different context, can be successfully applied to understanding the linguistic structure of MV.

In Section 2, we shall briefly elaborate the background of this work, giving a general discussion of informal mathematical language and MV, and outlining the approach to developing MV based on type theory. Then, in Section 3, conceptual well-formedness is informally discussed. In Section 4, a formal notion of conceptual category is discussed and used to give a type-theoretic analysis of the issues of syntactic and semantic correctness in MV. Section 5 introduces coercive subtyping and discusses how its abbreviational mechanisms can be used in studying the conceptual structure of MV and providing mechanisms such as overloading for more flexible and productive analysis of meaningfulness of expressions. Section 6 considers some implementation issues of MV, and discusses related work.

2 Developing Mathematical Vernacular: a Type-theoretic Approach

In the following, we consider the idea of informal mathematical language, and then our notion of a mathematical vernacular, ending with background information on type theory and its automation.

2.1 Informal Mathematical Language

Mathematicians communicate with a mixture of natural language and symbols. However, the NL used is restricted, and certain idioms appear frequently. We call it "Informal Mathematical Language" (IML). IML has important differences from unrestricted natural language: many complications from the unrestricted natural language, such as tense and metaphor, do not occur. Furthermore, there are problems which are specific to, or assume a greater importance in the context of, IML. Note that IML has never been defined: it is not a formal language in any sense. It arose in an ad hoc way as a means to describe mathematical development in a convenient fashion. Consequently, there is no "universal" IML. However, one can identify many important characteristics; these include

- **Correctness**: Applications must not make mistakes. Users must not be allowed to believe they have proved a false theorem, and likewise must not be prevented from proving a true one.
- **Introduction of New Concepts**: Much mathematics consists of defining new concepts and investigating the consequences of the definitions. Concepts are usually referenced by NL phrases which are chosen to convey some of the meaning of the concept. Such phrases are not always used as fixed entities, eg one may understand "finite Abelian group" having only seen definitions of the individual concepts. This introduction of new phrases and their link to underlying meaning is a key part of IML.
- **Informality**: Proofs in IML do not contain the low-level detail that a formal mathematician or proof checker would require. Such steps are considered obvious, and hence often omitted, to concentrate on the important steps. Any competent mathematician should be able to provide the missing details. Some of this inference would be needed in interpreting IML, in the form of automated reasoning. Note that informality includes use of abbreviations like "finite group".
- **Idioms**: Certain set phrases exist with definite meanings, for example those used in statements of quantification ("for all X such that Y, P holds"). In a sense, such phrases have acquired a meaning independent of the constituent words, and are most easily analysed as phrases rather than word by word.
- **Mix with Symbolic Expressions**: Expressions using mathematical symbols must be understood to some degree, in order to understand the surrounding IML. These expressions may themselves be informal, compared to the detail required in a completely formal language. For example, in "$f(x, y)$ is finite", we must know what $f(x, y)$ produces, so we can understand the whole sentence.

2.2 Mathematical Vernacular

In order to mechanise any aspect of mathematics which involves IML, such as our long-term aim to implement an interactive system based on MV, we need a good *formal* understanding of the language. As mentioned above, there are quite a lot of interesting and novel problems attached to IML. Given the prime need for correctness in implementing IML, we believe it is necessary to identify successful parts of IML together with good practices suggested by our experience of formal mathematics, and fully formalise that, rather than attempt to formalise "all" of IML, a concept which we find hard to define precisely.

We therefore characterise this language as one which has a formal, type-theoretic semantics, and is implementable with current CAFR. Naturally, the language should be as close as possible to good IML, in the sense of allowing the richness of IML without having too many restrictions. For convenience, we will call it "mathematical vernacular" (MV). This term underlines the informality of the language relative to formal mathematics, but reminds that formality is still important. Similarly to IML, note that there is no "universal" MV: naturally there will be limitations of applicability for this MV language.

Our view is that MV will be *developed* by formalising the essential 'core' of mathematical language, without which no useful mathematics may be done – even if the means of expression are cumbersome, then by extending this core to make the language more flexible without losing the formal properties. The material in section 4 considers one part of this core: the "well-formedness" of mathematical concepts.

A further constraint on MV is the limits of type theory and its associated technology, which we introduce below. But MV can also make new demands on them, and show new ways of looking at them. One need we have identified is for better 'meta-variable' facilities, which will allow a user to omit parts of his proofs temporarily – such as details he considers trivial. This 'feedback' of ideas can also occur for IML: studying it in order to create MV will help to identify good parts and bad parts, showing a way to improvements in IML.

Thus, we can regard development of MV as a constraint satisfaction problem – MV sits between mathematics and its realisation in IML, and mathematics and its realisation in type theory and its implementations; MV will have implications for all of those areas.

Related work includes that of de Bruijn [dB94], the Mizar group [Miz], and Ranta [Ran94, Ran96]; see section 6.2.

2.3 Type-theoretic Approach

Our type-theoretic approach to the study of MV builds on existing work on computer-assisted formal reasoning, represented by the use of proof assistants based on type theory in the formal development of mathematical theories and proofs. One motivation in studying MV is to investigate how to implement a suitable natural language interface for interactive development of mathematics with the support of such mechanised theorem proving. We give a brief introduction to formalising and checking mathematics using type theory, and to its associated proof assistants.

Type theory was chosen because of its properties of decidable checking and flexible representation. (This decidability was also a consideration in our decision to produce a type-theoretic semantics for MV.)

Proof Checking with Type Theory A type theory is a language which allows representation of types and objects, with a judgement form of when an object belongs to a certain type, and inference rules for constructing judgements. The notion of *dependent* types, and the addition of types to represent logic and data structures allows complex reasoning. Propositions are represented by types, and a proof of a proposition as a term which inhabits the type representing the proposition. Theorems are typically expressed as propositions, and proved by a term of suitable type; this term can then be used as a function in proofs of other propositions. We can also have algebraic structures, eg groups – which can be represented as a (dependent) tuple of a set, a binary operator over that set, an identity element, and an inverse function over the set, plus a proof that the data items satisfy the axioms[2].

Type theory and its associated technology is a developing field. Novel ways of using and presenting type theory are being investigated, and aspects of representation are being studied: equality is a particular issue that needs careful consideration. A lot of serious mathematics has already been done based on type theory, eg Ruys' work on the Fundamental Theorem of Algebra [Ruy], or the formalism of type checking of certain type systems [vBJMP93], or Alex Jones' work on the Decidable Dependency Theorem in Linear Algebra [Jon95]. A good source of further information is Bailey's thesis [Bai98], which contains much discussion on how to formalise maths with type theory and how to make the formalised version more understandable, followed by the formalisation of a substantial example (parts of Galois Theory).

Mechanised Type Theory Here we outline the current "tool support" for type theory. This is in the form of *proof assistants*, such as Lego [LP92] or Coq [Coq96], which help with several aspects of proof production: managing the details of representing a piece of maths (definitions of concepts); storing proved theorems and assumptions (which comprise a 'context', together with the definitions); guiding the process of proof construction; and checking whether a given term proves the statement it is claimed to.

Construction of proofs is by 'refinement'. For example, a proof of $A \wedge B$ is achieved by finding proofs of A and of B separately, at which point the proof assistant will combine them appropriately to prove the main goal. There is currently little automation in these proof assistants: most of the work is done by the user, although there are some simple mechanisms to prove basic propositions, eg simple logic or to help with repetitive operations in induction proofs.

There are obviously quite a few differences between doing conventional maths and using a proof assistant. Some of these will need attention to support interactive work with MV, most importantly to allow *holes* or meta-variables in proof development that will allow representation of statements from the user that are incomplete in a formal sense. We discuss this further in section 6.

[2] Actually, dependent tuples are represented with a Σ type constructor. $\Sigma(A, B)$ is the type of dependent pairs whose first element is of type A, and whose second element is of type $B(x)$ where x is the first element. Other representations are possible.

3 Mathematical concepts and meaningfulness of expressions

3.1 Meaningfulness of expressions

Meaningfulness, or *semantic acceptability*, as opposed to syntactic acceptability, can be illustrated by Chomsky's example, "Colourless green ideas sleep furiously". His observation is that this sentence is perfect as far as (pure) syntax is concerned, and that its unacceptability lies in the domain of conceptualisation (although it can be meaningful in some imaginative context).

In mathematics, semantic acceptability presumes a distinguishably important role. Semantically ill-formed expressions and meaningless sentences are strictly unacceptable. This is partly because that rigour and correctness are the basic requirements for mathematical development. A formal analysis of mathematical concepts is important in understanding how to exclude ill-formed constructions such as "Abelian set" on the one hand, and why we should accept the more flexible uses such as "finite group" and the generative uses such as "finite Abelian group" even if no explicit definitions of such phrases have been given, on the other.

In the development of mathematical vernacular, such an analysis (a type-theoretic analysis in our case) gives us guidance to the design and implementation of MV. For example, checking semantic acceptability involves type-checking, verification of presuppositions, generation of proof obligations; these can trigger proof search and system-user interactions. (See Section 6 for a further discussion.)

3.2 Mathematical Concepts: an Informal Discussion

Before considering a formal treatment in the next two sections, we first give an informal discussion of mathematical concepts, their formation, and relationships between them[3].

Objects, Basic Classes, and Properties A starting point is to observe that in IML, as in NL, the terms of description are oriented towards human thinking. Mathematical objects, including primitive ones, are naturally organised into different basic classes, objects in each of which may share common structures. These basic classes correspond to primitive types in type theory (eg, the inductive type of natural numbers). Objects have properties and a property is defined over a class of objects. In logical terms, properties are expressed by means of predicates over objects of a type (eg, 'even' and 'odd' over natural numbers).

[3] Our discussion on mathematical concepts is largely independent of surface linguistic details. For example, the phrases "group which is Abelian" and "Abelian group" are just regarded as different ways of describing the same conceptual entity. Note that we only use natural language phrases to denote concepts as a matter of convenience.

Mathematical Concepts Denoting Classes of Objects Mathematical concepts, typically represented by means of substantives (common noun phrases) such as 'group' and "finite set", denote classes of objects. Introduction of a new mathematical concept is done by giving an explicit definition. For example, "A semigroup is a set together with an associative binary operation over the set." Forming a class of objects introduces abstraction in the sense that one can assume a hypothetical member of the class, and define and study properties of the objects in the class (eg, by means of quantification and generalisation).

Existing concepts may be combined to form new concepts. Typical ways to do this include structural composition (eg, set together with a binary operation) and specification of logical properties or constraints (eg, associativity) shared by the objects in the class denoted by the concept. The distinction between structure and logical constraints is usually conceptually clear, though it may sometimes be blurred in practice.[4] Logical constraints alone result in concepts denoting more restricted classes, often labelled by a qualified name (eg "finite set"). In some cases, especially when new *structure* is postulated, the new class is treated as a distinguished concept with a new name (eg, 'semigroup' obtained from 'set', 'field' defined by means of 'ring'), rather than being qualified by some property.

Generative Formation of Concepts Because of the generative nature of IML, not every concept used in mathematics has to be introduced by explicit definition, and our study of MV should reflect this.[5] Defining a concept also introduces many other composite concepts by linguistic convention. Linguistic mechanisms for this include, at the surface level of IML, the use of adjectives and their combinations in syntactically valid constructs to form substantives representing concepts. For instance, having defined 'cyclic' and 'Abelian' over groups, we can also use "cyclic Abelian group" without having to define it explicitly; having to do so would be unproductive at least.

Inheritance between mathematical concepts In general, it is semantically invalid to use a property to qualify a concept over whose objects the property is undefined. For example, just as "green ideas" is unacceptable, so is "odd group", if 'odd' is a property defined over natural numbers rather than groups.

However, though naturally rejecting obviously meaningless constructions, mathematicians do allow a considerable flexibility into their practice in communicating mathematical ideas and proofs. Many of these are based on special

[4] Sometimes, especially in classical mathematics, a property can suggest additional structure of objects through existential statements, especially those with uniqueness property. A good example is of an element in a group which is an identity of the associated operator; this can easily be proven unique, so one usually refers to 'the' identity element.

[5] This is a different view from that taken by Prof de Bruijn in his study of MV, where he correctly identifies the importance of definitions in mathematical development, but considers that everything should be introduced by explicit definitions [dB94].

relationships between different mathematical concepts. Examples of such relationship include concept implication (or class inclusion), structural inheritance, and their combinations. For instance, although 'finite' is a property defined over sets, "finite group" is usually regarded as valid in expressing "group whose carrier set is finite". As another example, one often says "for all elements in [the carrier set of] group G, ...", with the phrase in the bracket omitted. Such abbreviations are regarded as tacit conventions and are not expected to cause problems.

Other issues concerning mathematical concepts

- *Transformations between Concepts.* Expressions that denote transformations or operations between concepts (eg, union of two sets) constitute an important class of entities to be studied. An interesting issue is to study what properties they preserve and how we represent such a notion of preservation of properties, eg as in the union of two finite sets being automatically finite.
- *Non-restrictive adjective-noun modification.* Not every use of an adjective to modify a concept results in a more restricted concept. A good example is "left monoid". (A left monoid is a similar structure to a monoid which has only a left identity.) A left monoid is not a monoid (cf, a fake gun is not a gun.) General analysis and treatment of such phenomena are out of the scope of this paper.
- *Lexical ambiguity.* In mathematics, truly ambiguous expressions and sentences are regarded as unacceptable, or a bad practice at the best. However, overloading of terminology and notation does occur quite often for the sake of abbreviation or simplicity. When this happens, a term is used to have multiple meanings with contrasted ambiguity (homonymy). For example, 'finite' in "finite set" and "finite sequence" may have formally different (though informally related) meanings.
- *Redundant information.* Whether expressions such as "finite finite set" are semantically acceptable is debatable. It is certainly desirable to avoid but it seems difficult to find logical disciplines to deal with it. A more interesting example is "Abelian left monoid", which is logically a monoid because commutativity implies that the left and right identities are the same.

4 Conceptual categories: a type-theoretic analysis

In this section, we introduce and study a notion of category. This is used to investigate the semantics of expressions denoting mathematical concepts. We do not consider syntactic issues at this stage; the semantic details are studied independently of such issues[6]. We also ignore (for the moment) issues concerning instances of mathematical concepts, such as the meaning of the claim that a particular object x is an instance of concept C.

[6] Moreover, the work here does not force a particular approach to syntax, so any comments we make would be provisional, and not add to the theoretical content of this paper. The same applies to other related issues.

The basic formal theory we use has been studied in several contexts, including the subset theory developed by Nordström, Petersson and Smith in Martin-Löf's type theory [NPS90], the specification calculus by Luo in the Extended Calculus of Constructions and UTT [Luo93], and the related (but different) framework on deliverables [BM92, McK92] and mathematical theories [Luo91a]. Here, we apply this theory to mathematical concepts and the related well-formedness issues. In the next section, we shall extend this to introduce a notion of subcategory based on the theory of coercive subtyping.

Our presentation below will be precise but informal in that type theory is used informally as the semantic language to define the meanings of categories and category constructors. The underlying type theory is UTT, which consists of an impredicative type universe of logical propositions ($Prop$), predicative type universes, inductive types including type constructors for functional types $A \to B$, dependent functional types $\Pi(A, B)$, types of dependent pairs $\Sigma(A, B)$, and types of natural numbers, lists, trees, etc. (See [Luo94] for details.) UTT is implemented in the proof system Lego [LP92].

4.1 Conceptual categories

A *(conceptual) category* represents a mathematical concept. We shall use the judgement form $C : \textsc{Cat}$ to denote that C is a category.

In general, a category C consists of two components: the syntactic category of C, $\textsc{Syn}(C)$, and the logical constraint of C, $\textsc{Log}(C)$. The syntactic category $\textsc{Syn}(C)$ is a type, representing the structure of the objects of the represented concept and the logical constraint $\textsc{Log}(C)$ is a predicate over the syntactic structures (ie, $\textsc{Log}(C)$ is of type $\textsc{Syn}(C) \to Prop$).

Another form of judgement is $e : C$, asserting that the expression e is of category C. The meaning of this judgement is defined as:

- $e : C$ if, and only if,
  - e is of type $\textsc{Syn}(C)$, and
  - $\textsc{Log}(C)(e)$ is true (provable in the type theory).

We introduce the following notions of well-formedness and logical correctness.

Definition (well-formedness and logical correctness)

- M is *well-formed* (more precisely, an expression that denotes M is well-formed) if either $M : \textsc{Cat}$ or $M : \textsc{Syn}(C)$ for some $C : \textsc{Cat}$.
- e is *logically correct wrt* C if $\textsc{Log}(C)(e)$ is true.

Note that logical correctness of an object e wrt C presupposes the well-formedness of category C and the well-formedness of e itself. Since type checking is decidable, we can automatically check whether an expression, either denoting a category or an object, is well-formed. Of course, this is not the case for logical correctness of objects.

We note that many of the cases traditionally discussed for semantic (un)acceptability are incorporated into the notion of well-formedness and some require checking logical correctness. For example, "odd group" is not well-formed, while "finite set" and "finite group" are (see below). Therefore, checking semantic acceptability (meaningfulness in the traditional sense, as discussed in section 3) can be helped by (decidable) type-checking. In addition, checking logical correctness incorporates verification obligations (eg, verification of presuppositions).

Remark. We have taken the view that proof terms (proofs of logical propositions) should not be regarded as explicit objects. Effectively, an object of the category denoted by "finite set" (we name it $FSet$) is just a set that is finite, as opposed to a set together with a proof of its finiteness. This allows a direct analysis of sentences such as "If A is a finite set, then ...", where A denotes an object of category set. Note that the hypothesis of the sentence corresponds to the judgement $A : FSet$. If an object of $FSet$ were required to be a structure which contains the set and a proof term rather than just a set, the if-clause would have been trivially false (ie, ill-typed and not derivable)[7]. We omit further specific discussion of how objects and proof terms will be treated under this strategy (see [CL98] for more information).

4.2 Category Constructors

Categories can either be base categories or the result of applying a category constructor. A category or a category constructor is defined by giving only its formation rule and definitions of its syntactic category and logical constraints.

Category of Logical Sentences An example of base category is S, the category of logical sentences. The syntactic category of S is the type of logical propositions and the logical constraint of S is the true predicate:

- $\text{S}\textsc{yn}(S) = Prop$
- $\text{L}\textsc{og}(S)(e) = true$, for any $e : \text{S}\textsc{yn}(S)$.

A base category is isomorphic to its syntactic category, since its logical part is always true (ie, there are no logical constraints). In the following, for notational convenience, we shall just write S for the syntactic category of S.

Syntactic categories Besides the other category constructors, one can consider $\text{S}\textsc{yn}$ as a category constructor as well. We have:

- $\text{S}\textsc{yn}(\text{S}\textsc{yn}(C)) = \text{S}\textsc{yn}(C)$
- $\text{L}\textsc{og}(\text{S}\textsc{yn}(C))(e) = true$, for any $e : \text{S}\textsc{yn}(C)$

[7] To manipulate structures that contain proof terms requires some non-trivial operations to ensure that expressions remain well-typed. For example, if a binary relation R is said to be symmetric, reflexive, and transitive, then to show that it is transitive and symmetric requires some unpacking and rebuilding of R.

That is, the syntactic category of $\textsc{Syn}(C)$ is itself and the logical constraint is always true, ie, there are no logical conditions on a plain syntactic object.

Δ-categories The Δ-operator creates a new category by attaching to a category C a logical predicate on the syntactic structure of C. Its formation rule is:

$$\frac{C : \textsc{Cat} \quad p : \textsc{Syn}(C) \to S}{\Delta(C,p) : \textsc{Cat}}$$

The syntactic category of $\Delta(C,p)$ is the same as that of C and its logical constraint is the logical conjunction of that of C and the extra constraint p.

- $\textsc{Syn}(\Delta(C,p)) = \textsc{Syn}(C)$
- $\textsc{Log}(\Delta(C,p))(e) = \textsc{Log}(C)(e) \wedge p(e)$

For example, if *set* is the category of sets and $finite$ is the finiteness predicate defined over sets, the category of finite sets can be represented by $\Delta(set, finite)$. Similarly, with *Abelian* and *cyclic* defined over group structures, "cyclic Abelian group" can be represented as $\Delta(\Delta(group, Abelian), cyclic)$. Note that, conversely, the calculus does not allow the ill-formed concepts such as "Abelian set" or "transitive group" since their representations fail to type-check.

Σ-categories The formation rule for Σ-categories is

$$\frac{C : \textsc{Cat} \quad f(x) : \textsc{Cat}\ [x{:}\textsc{Syn}(C)]}{\Sigma(C,f) : \textsc{Cat}}$$

which says that, if C is a category and f is a family of categories indexed by C-structures, then $\Sigma(C,f)$ is a category. The definitions of the corresponding syntactic category and logical constraint are:

- $\textsc{Syn}(\Sigma(C,f)) = \Sigma(\textsc{Syn}(C), \textsc{Syn} \circ f)$
- $\textsc{Log}(\Sigma(C,f))(e) = \textsc{Log}(C)(\pi_1 e) \wedge \textsc{Log}(f(\pi_1 e))(\pi_2 e)$

where Σ on the RHS of the first equation is the Σ-type constructor (types of dependent pairs) with the projection operators π_1 and π_2, and $\textsc{Syn} \circ f$ is the composition of $\textsc{Syn}$ and f. The above definitions indicate that the syntactic component of an object of a Σ-category is a pair of the base structure and its extension, and that the logical component is a combination of those from the base with those from the extension. Note that we have overloaded Σ for both the type constructor and the category constructor (this is for notational convenience).

As Σ-types can be used to represent types of structures (tuples), Σ-categories can be used to represent mathematical structures of algebraic theories etc. Note that properties of such structures can only be added by using the Δ constructor (otherwise, explicit proof terms will appear in the system, which we wish to avoid). A simple example of structure is the formation of the concept of monoid from that of set by adding a binary operator and an identity with their properties. A more sophisticated example would be to form the concept of ring by combining the concepts of group and monoid by adding extra logical constraints; this involves sharing of the common carrier set, and we omit the details here (see [Luo93, Luo91a]).

Functional categories We can form the functional category of two categories:

$$\frac{C : \textsc{Cat} \quad D : \textsc{Cat}}{C \Rightarrow D : \textsc{Cat}}$$

The corresponding syntactic category and logical constraint are defined as:

– $\textsc{Syn}(C \Rightarrow D) = \textsc{Syn}(C) \to \textsc{Syn}(D)$
– $\textsc{Log}(C \Rightarrow D)(f) = \forall c{:}\textsc{Syn}(C).\ \textsc{Log}(C)(c) \supset \textsc{Log}(D)(f(c))$

That is, the objects of a functional category are the functions that preserve the logical constraints.

Objects of a functional category represent operations or transformations between concepts. Here is an example. Let B be a set, the concept "subset of B" can be represented as $Sub(B) = \Delta(set, sub_B)$, where $sub_B(A) = A \subseteq B$. Then, "complement of ... wrt B", the operation that takes a subset of B and returns its complement, is defined to be of category $Sub(B) \Rightarrow set$. Note that the logical correctness of "complement of A wrt B" requires that A be a subset of B – a presupposition to be verified. If A is not a subset of B, the phrase is logically incorrect.

There are other category constructors that can be introduced and used to analyse well-formedness and logical correctness of expressions and sentences, but we omit their discussion here.

4.3 Equivalence between categories

Another important notion is the equivalence between categories. A category C is *equivalent* to category D if $\textsc{Syn}(C) = \textsc{Syn}(D)$ (computational equality) and $\textsc{Log}(C)$ and $\textsc{Log}(D)$ are equivalent, ie, $\forall x{:}\textsc{Syn}(C).\ \textsc{Log}(C)(x) \Leftrightarrow \textsc{Log}(D)(x)$ is true. For example, the concept of symmetric, transitive relation is equivalent to that of transitive, symmetric relation.

Note that this notion of equivalence between categories is extensional. This reflects the fact that a mathematical concept can be defined in intensionally different ways. For example, the concept of "prime number smaller than 3" is equivalent to that of "number that is equal to 2".

5 Coercive subtyping, subcategories, and sense selection

As discussed in Section 3, structural inheritance and conceptual implication are important relationships between mathematical concepts. Many linguistic conventions, in particular, abbreviational conventions, are based on them. In this section, we extend our study of conceptual categories to investigate such relationships between categories.

In particular, we shall study how coercive subtyping [Luo97, Luo98], a new theory of subtyping and inheritance in type theory, can be applied in our type-theoretic analysis of mathematical language. The benefits are to provide adequate abbreviational mechanisms, to understand the inheritance relationships between mathematical concepts, and to deal with contrastive lexical ambiguity by means of the overloading mechanism.

5.1 Coercive subtyping

We first give an informal introduction to the underlying theory of coercive subtyping and then show how it allows abbreviations such as "finite group".

The basic idea is to consider subtyping as an abbreviational mechanism. A is a subtype of B, notation $A \leq B$, if either $A = B$ (computational equality) or A is a proper subtype of B (notation $A < B$) such that there is a unique implicit coercion from A to B. Any function from A to B can be specified as a coercion, as long as the coherence property holds for the overall system (in particular, there cannot be two different coercions from any A to any B).

This idea generalises both the traditional notion of inclusion-based subtyping (eg, between types of natural numbers and integers) and that of inheritance-based subtyping (eg, between record types). Anthony Bailey has implemented coercion mechanisms in Lego (and Saibi in Coq [Sai97]), and considered its applications to formal development of mathematics (Galois theory) based on type theory [Bai98]. For some meta-theoretic results of coercive subtyping, see [JLS97, SL98].

The mechanism with which coercive subtyping works can be explained informally as follows. If A is a proper subtype of B with coercion c, $a : A$, and $\mathcal{C}[\_]$ is a context where an object of type B is required, then a can be used in that context — $\mathcal{C}[a]$ stands for (more precisely, is computationally equal to) $\mathcal{C}[c(a)]$.

For example, based on the usual formalisation of algebraic structures in type theory, the types of structures of groups and sets (ie, $\textsc{Syn}(group)$ and $\textsc{Syn}(set)$ in our notation) are Σ-types, with the former having the latter as a substructure. One can define a coercion $carrier$ from groups to sets which extracts the type corresponding to the carrier set of a group. Then, if $G : \textsc{Syn}(group)$, the logical proposition $\forall(G, P)$ (ie, $\forall x{:}G.\ P(x)$ in a more usual notation), which is not well-typed without subtyping, is well-formed and actually stands for $\forall(carrier(G), P)$, because $\forall$ requires a type as its first argument.

Similarly, if we interpret the conceptual categories of the previous section in UTT, and assume that $\textsc{Syn}(group)$ is a proper subtype of $\textsc{Syn}(set)$ with the forgetful map κ as coercion, then the category $Delta(group, finite)$ (for "finite group") is well-formed because

- $finite : \textsc{Syn}(set) \to S$
- $(\textsc{Syn}(set) \to S) \leq (\textsc{Syn}(group) \to S)$[8]

[8] Note that we have lifted the coercion between group and set to the function level; hence the use of function composition $\circ$ as part of the coercion applied to *finite*.

In particular, we have the following computational equality: $\Delta(group, finite) = \Delta(group, finite \circ \kappa)$. With the coercion made explicit, "finite group" means literally "group whose carrier set is finite".

Coercions may be propagated over data structures, extending the usefulness of simple coercions. More technically, subtyping relations can generalise to various type constructors by default rules, unless the defaults are overridden. For example, if $A' \leq A$ and $B \leq B'$, then $(A \to B) \leq (A' \to B')$ (ie, subtyping is contravariant wrt functional types). The subtyping relation generalises to Σ-types (types of dependent pairs): if $A \leq A'$ and $B(x) \leq B'(x)$ for $x : A$, then $\Sigma(A, B) \leq \Sigma(A', B')$. The expected composite coercions with respect to these generalisations can be easily constructed from the component coercions, but we omit the details here (see [Luo98]).

5.2 The subcategory relation

We can use the notion of coercion *directly* in the calculus of section 4 by developing a notion of *subcategory relation* as a lifted version of the subtyping relation. We say C is a subcategory of D, written $C \preceq D$, if and only if:

- $\textsc{Syn}(C) \leq \textsc{Syn}(D)$, and
- $\forall x{:}\textsc{Syn}(C).\ \textsc{Log}(C)(x) \supset \textsc{Log}(D)(x)$ is true (provable in type theory).

In other words, C is a subcategory of D if $\textsc{Syn}(C)$ is a subtype of $\textsc{Syn}(D)$ and the corresponding coercion preserves the logical constraints. Structural inheritance is reflected in the subtyping relationship between the syntactic categories, and conceptual implication in the relationship between logical constraints.

Also, we introduce the following terminology and notations.

- If $C \preceq D$ and $\textsc{Syn}(C) < \textsc{Syn}(D)$ with coercion c, we say that C is a *proper subcategory* of D (with coercion c), notation $C \prec D$.
- If $C \preceq D$ and the syntactic categories of C and D are computationally equal, we say that C *implies* D, notation $C \supset D$.

Note that, because we require the subtyping relation to be coherent, if $C \preceq D$ and $D \preceq C$, then it must be the case that C and D are equivalent.

It is easy to check that the subcategory relation is reflexive and transitive, and furthermore it has the expected properties concerning various category constructors. For instance, assuming the default coercive subtyping rules concerning Σ-types and functional types and the well-formedness of the categories concerned, the following are derivable judgements or rules that represent the typical subcategory relationships concerning the category constructors discussed in Section 4.2.

- $C \supset \textsc{Syn}(C)$.

Furthermore, subtyping is contrapositive over functions, ie $A \leq A', B \leq B'$, then $B' \to A' \leq A \to B$.

- $\Delta(C, p) \supset C$.
- $\Delta(C, p') \preceq \Delta(C', p')$, if $C \preceq C'$.
- $\Sigma(C, f) \preceq \Sigma(C', f')$ if $C \preceq C'$ and $f(x) \preceq f'(x)$ for all $x : \mathrm{S}_{\mathrm{YN}}(C)$.
- $(C \Rightarrow D) \preceq (C' \Rightarrow D')$ if $C' \preceq C$ and $D \preceq D'$.

Therefore, the mechanisms for abbreviation and inheritance provided by coercive subtyping are lifted to categories in a uniform way. For instance, if f is of category $D \Rightarrow D'$, c is of category C and $C \preceq D$, then $f(c)$ is of category D'.

5.3 Lexical ambiguity, overloading and sense selection

As discussed in Section 3, ambiguity is not desirable in mathematical texts. However, forms of local ambiguity which are resolvable in a wider (linguistic) context do occur frequently and naturally, thus should be allowed in MV. An example is of contrasted ambiguity, eg, "finite" in "finite$_1$ set" and "finite$_2$ sequence". Coercive subtyping allows a satisfactory treatment of this phenomenon through overloading of unit types (types containing only one object). This idea first appeared in [Luo97], where overloading pairs of Σ-types and product types is considered, and is further developed in Bailey's thesis [Bai98], where he makes extensive use of coercions and overloading.

The technique is to use several coercions from a unit type Unit (inductive type with only one object) to encode the multiple senses of an expression. The expression (eg, "finite") is represented by the object in the unit type, while the images of the coercions are its different senses (eg, "finite$_1$" and "finite$_2$"). When the expression is used in a context, its appropriate sense is selected, according to the coercive subtyping mechanism. For example, we shall have "finite sequence" = "finite$_2$ sequence" and, "finite set" = "finite$_1$ set".

Note that in using this mechanism, coherence must be maintained. So one must ensure that coherence is maintained (by having acceptable coercions) when the types of two different senses are related by the subtyping relation[9]. In our example concerning "finite$_1$ set" and "finite$_2$ sequence", we have that $finite_1 : set \Rightarrow S$ and $finite_2 : sequence \Rightarrow S$, and overloading coercions $\kappa_1 : Unit \rightarrow \mathrm{S}_{\mathrm{YN}}(set \Rightarrow S)$ and $\kappa_2 : Unit \rightarrow \mathrm{S}_{\mathrm{YN}}(sequence \Rightarrow s)$. If, for example, $sequence \prec set$ with some coercion, then we have $(set \Rightarrow S) \prec (sequence \Rightarrow S)$ by some coercion κ; in this case, we have to ensure $\kappa \circ \kappa_1 = \kappa_2$ to preserve coherence.

5.4 Related work on lexical semantics and remarks

It is not surprising that some of the phenomena discussed above have their counterpart in ordinary natural language. For example, one can consider the similarity between the following two examples:

---

[9] One possibility in an implementation is to suspend the subtyping relations that are oriented towards checking well-formedness of categories when performing the type checking necessary for sense disambiguation on the input text.

– a finite group = a group whose carrier set is finite
  $\neq$ an algebraic structure that is a group and that is finite
– a fast typist = someone who types fast
  $\neq$ someone who is a typist and who is fast

However, the NL example is defeasible – further information may affect which interpretation is chosen (eg a race between typists and accountants, [CB96]). This situation does not occur in IML: mathematical terms must have been given precise definitions, and the same meaning must result in all contexts.

Very recently, the work by Pustejovsky [Pus95] on generative lexicon and the work by Jackendoff on enriched composition [Jac97] have come to our attention. They have studied the conceptual structure and lexical semantics of natural language based on an idea of coercion. It would be interesting to study the connections of their work and the work reported here; in particular, we believe that the theory of coercive subtyping may have its application in the wider context as well.

A remark on methodology of our research may be worth making. We have taken an approach of componential analysis in our study of the conceptual structure (and lexical semantics). We argue that this is a suitable approach to the study of MV (and IML in general). This is partly because terms in mathematics are (or, can be) precisely defined and partly because mathematical terms can be considered to be defined from some basic concepts such as those in a foundational language. For these reasons, the traditional arguments against componential analysis in lexical semantics (cf, [Lyo95]) do not apply to mathematical language.

6 Implementing MV: Discussion and Related Work

As stated in Section 2, one of our research objectives is to develop the implementation technology of MV based on type theory. The long-term research aims include development of interactive systems with MV as the user language, that support productive communications between the user and the system. We are currently studying basic but crucial techniques such as more advanced treatment of meta-variables and multi-thread proof development techniques. A prototype is being developed for exploring these ideas; it is based on the typed logical framework LF introduced in Chapter 9 of [Luo94], which allows specification of different type theories suitable for different applications (such as the formal system of section 4).

In the following, we first consider how the type-theoretic analysis of mathematical concepts considered above can be used in implementing MV, and discuss related work.

6.1 Applications of Conceptual Categories in Implementation

In implementing MV based on type theory, the formal analysis reported in this paper supplies not only important theoretical ideas, but practical guidance on how to represent and reason about mathematical concepts, and how to check well-formedness and logical correctness of expressions and sentences in MV. In fact, we regard this analysis as one of the key steps in developing and implementing MV as it deals with objects, classes, and properties in mathematics.

First, the formal framework of conceptual categories is a basis for a type-theoretic semantics of MV (ie, how to translate MV expressions and sentences into type theory). Although working in a type-theoretic framework, we believe that explicit use of proof terms (or expressions of proofs) in mathematical language is unnatural to most of the mathematicians (even for constructive mathematicians doing informal mathematics). The system of conceptual categories provides a suitable framework based on type theory that allows direct and natural interpretations of MV expressions and sentences.

Secondly, the framework not only relates checking of well-formedness and correctness to type-checking, but provides a basis for more intelligent system-user interaction. For example, the categories of expressions such as "complement of ... wrt B" (of category $Sub(B) \Rightarrow set$) capture the presuppositions of sentences involving such expressions (eg, the logical correctness of "complement of A wrt B" presupposes that "A is a subset of B"). Among other things, this would enable a system that implements MV to make use of the subtle difference between presuppositions and unproven claims (eg, 'facts' explicitly claimed by the user without proof, and which can't always be proven automatically by the system) to produce more reasonable response in system-user interaction. Claims will be left as gaps in the development, while phrases or sentences with unverified presuppositions would require justification, by a mixture of automatic reasoning and further interaction with the user.

Furthermore, combined with coercive subtyping, the formal framework gives flexible mechanisms that allow a systematic treatment of abbreviations, direct inheritance, and multiple sense selection. For example, dealing with words such as "finite" that can be used to qualify many different but related concepts (set, group, ...), a system based on coercive subtyping does not have to consider all of these possible meanings when processing a text, but just consider very few essentially different meanings of the word, with the rest subsumed by the subtyping relations. Another practical benefit is that the abbreviational mechanisms allow compression of formal expressions (semantic denotations in type theory of MV expressions) and hence promote clearer communication with the user.

The notion of conceptual category is defined by interpretation into the underlying type theory (UTT in our case), which is a relatively sophisticated system. It would be very interesting to see whether it is possible to design an independent but simpler system of categories that can be used to capture the need in correctness checking of MV. The benefit of such a system is twofold. In theory, it would make it clear what basic mechanisms are needed for checking well-formedness and correctness for MV. In practice, it may provide simpler ways to perform independent well-formedness checking without having to consider unnecessary details of formalisation. We can both study and implement this using the implementation of LF mentioned above.

6.2 Related Work on Mathematical Vernacular

There have been several research efforts to study, to design, and to implement mathematical vernaculars in the general sense of the term. Some of them are more closely related to our research in their objectives and methods than others; among the more related are de Bruijn's work on mathematical vernacular [dB94], Ranta's work on type-theoretic grammar [Ran94] and his research on informal mathematical language based on type theory [Ran95] and its implementation [Ran97][10], Coscoy's work on proof explanation in Coq [CKT95], and the Mizar project [Miz], where a mathematical vernacular has been defined and implemented in a batch system whose logic is based on set theory, and subsequently used to formalise an impressive amount of mathematics.

Our work on mathematical vernacular has been substantially influenced and improved by general discussions in the works above, and by communications with Aarne Ranta. In particular, Prof de Bruijn's pioneering work on MV (and Automath [dB80]) has significantly influenced the research field and our work. His work on MV offers many insights in designing mathematical vernaculars, which we very much believe are deserving of further investigation and development.

Acknowledgements

We would like to thank the following for many useful discussions: Peter Aczel, James McKinna, Aarne Ranta, and members of the Computer Assisted Reasoning Group in Durham.

References

[Bai98] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.

[BM92] R. Burstall and J. McKinna. Deliverables: a categorical approach to program development in type theory. LFCS report ECS-LFCS-92-242, Dept of Computer Science, University of Edinburgh, 1992.

[CB96] A. Copestake and T. Briscoe. Semi-productive polysemy and sense extension. In J. Pustejovsky and B. Boguraev, editors, *Lexical Semantics: The Problem of Polysemy.* Clarendon, 1996.

[CKT95] Y. Coscoy, G. Kahn, and L. Théry. Extracting texts from proofs. Technical Report 2459, INRIA, Sophia-Antipolis, 1995.

[CL98] P. Callaghan and Z. Luo. Mathematical vernacular in type theory-based proof assistants. In R. Backhouse, editor, *User Interfaces for Theorem Proving, UITP '98*, July 1998.

[Coq96] Coq. *The Coq Proof Assistant Reference Manual (version 6.1).* INRIA-Rocquencourt and CNRS-ENS Lyon, 1996.

[10] One function of this system is to translate simple expressions in mathematical language to expressions in a type theory. We have adapted this system to produce expressions in the Lego type theory (UTT): this involved working around a few important technical differences in the type theories concerned.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[dB94] N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors, *Selected Papers on Automath.* North Holland, 1994.

[Jac97] R. Jackendoff. *The Architecture of the Language Faculty.* MIT, 1997.

[JLS97] A. Jones, Z. Luo, and S. Soloviev. Some proof-theoretic and algorithmic aspects of coercive subtyping. *Proc. of the Annual Conf on Types and Proofs (TYPES'96)*, 1997. To appear.

[Jon95] A. Jones. The formalization of linear algebra in LEGO: The decidable dependency theorem. Master's thesis, University of Manchester, 1995.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[Luo91a] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1), 1991.

[Luo91b] Z. Luo. Program specification and data refinement in type theory. *Proc. of the Fourth Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT), LNCS 493*, 1991. Also as LFCS report ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University.

[Luo93] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3(3), 1993. An earlier version appears as [Luo91b].

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[Luo97] Z. Luo. Coercive subtyping in type theory. *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258*, 1997.

[Luo98] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 1998. To appear.

[Lyo95] J. Lyons. *Linguistic Semantics.* Cambridge University Press, 1995.

[McK92] J. McKinna. *Deliverables: a categorical approach to program development in type theory.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[Miz] Mizar. Mizar home page. `http://mizar.uw.bialystok.pl/`.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proof and Programs, LNCS*, 1994.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[Pus95] J. Pustejovsky. *The Generative Lexicon.* MIT, 1995.

[Ran94] A. Ranta. *Type-theoretical Grammar.* Oxford University Press, 1994.

[Ran95] A. Ranta. Type-theoretical interpretation and generalization of phrase structure grammar. *Bulletin of the IGPL*, 1995.

[Ran96] Aarne Ranta. Context-relative syntactic categories and the formalization of mathematical text. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs.* Springer-Verlag, Heidelberg, 1996.

[Ran97] A. Ranta. A grammatical framework (some notes on the source files), 1997.

[Ruy] Mark Ruys. *Formalizing Mathematics in Type Theory.* PhD thesis, Computing Science Institute, University of Nijmegen. (to be submitted).

[Sai97] A. Saibi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.

[SL98] S. Soloviev and Z. Luo. Coercive subtyping: coherence and conservativity, 1998. In preparation.

[vBJMP93] L. van Benthem Jutting, James McKinna, and Robert Pollack. Type-checking in pure type systems. submitted for publication, 1993.