

Mathematical Vernacular in Type Theory-based Proof Assistants*

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

March 27, 1998

Abstract

In this paper we present the Durham Mathematical Vernacular (MV) project, discuss the general design of a prototype to support experimentation with issues of MV, explain current work on the prototype – specifically in the type theory basis of the work, and end with a brief discussion of methodology and future directions.

The current work concerns an implementation of Luo’s typed logical framework LF, and making it more flexible with respect to the demands of implementing MV – in particular, meta-variables, multiple contexts, subtyping, and automation. This part of the project may be of particular interest to the general theorem proving community.

We will demonstrate a prototype at the conference.

1 Introduction: Defining a Mathematical Vernacular

The long term aim of the project is to develop theory and techniques with which the complementary strengths of NLP (Natural Language Processing) and CAFR (Computer-Assisted Formal Reasoning) can be combined to support computer-assisted reasoning. Our chosen area is mathematical reasoning, with the goal of implementing proof systems that allow mathematicians to reason in a “mathematical vernacular” (MV). Thus, mathematicians may work in a style and a language which is closer to traditional practice, rather than in a formal setting.

The MV project is a natural consequence of existing projects at Durham on Type Theory and on Natural Language Processing. The work on type theory includes coercive subtyping [10, 8], implementation of automatic decision procedures [21], and applications of type theory, eg in concurrency. The NL work is represented by the LOLITA system [19, 4]: this large project aims to build a general purpose platform with which specific NL applications can be built, eg information extraction [14], or object-oriented requirements analysis [12].

In the remainder of this section, we briefly discuss the notion of mathematical vernacular, and our type-theoretic approach.

Mathematical Vernacular The term “Mathematical Vernacular” has been used before with varying meanings, eg in [13] and [5]. We define it to mean a language which is suitable for ordinary mathematical practice, and which can be implemented with current technology and under the guidance of a formal semantics. The basis for implementation will be constructive type theory and its associated technology.

This MV is not necessarily the same as the language mathematicians commonly use, which we shall call “Informal Mathematical Language” (IML). Thus, we do not intend to model completely what mathematicians do, for several reasons. Firstly, IML is an informal solution to the problem

*This work is supported partly by the University of Durham Mathematical Vernacular project funded by the Leverhulme Trust (see <http://www.dur.ac.uk/~dcs0z1/mv.html>) and partly by the project on Subtyping, Inheritance, and Reuse, funded by UK EPSRC (GR/K79130, see <http://www.dur.ac.uk/~dcs0z1/sir.html>).

of communicating mathematics, and we can't be sure that it is the most effective way. Secondly, groups of people have their own idea about IML, of what is acceptable and what isn't, and so on. It would be hard to study such variation in opinion. Lastly, IML does not take account of the new technology for proof checking. This is an important point: we are still learning how to use the formal side of the technology, so an examination of how natural language may be productively used with it is certainly worthwhile. We discuss further methodological issues in section 4.1.

A Type-theoretic Approach Type theory provides a flexible language for expressing and checking a lot of non-trivial mathematics (eg [3]). The process of checking proofs (via type checking) is decidable for the type theories we will use, so it may be mechanised (implemented) straightforwardly. However, type theory and the technology based on it is hard to use: humans require a lot of training to use it effectively, and experts are not fully satisfied with the current implementations. One aim of our project is to contribute to this technology, especially in terms of being able to support the demands of MV.

Constructive type theory with dependent types is also a useful framework for natural language, eg in semantics (see Ranta [15]). The use of coercive subtyping may add flexibility to formal accounts of NL [11].

2 A Prototype for Experimentation

It is difficult to design an interactive system based on MV in a top-down style. We have ideas on various pieces of the problem, but tying these together is difficult. Thus, we need a prototype which will allow experimentation with different ideas. This prototype will also be a useful intermediate tool. Bear in mind that our interest is not only practical – we would also like to understand mathematical language and how it differs from general language – so prototype development will not be a pure ‘engineering’ matter.

This section outlines our initial design of such a prototype, discussing the basic requirements of functionality, and sketching the architecture which we will use to achieve this. More ambitious aims and future developments are discussed in section 4.2.

2.1 Establishing Requirements

We are taking a simple piece of mathematics as a guide in this early work. This step gives substance to our initial aim: it helps us gauge progress and to ‘characterise’ what the prototype is capable of. It is in general difficult to write exact requirement specifications for NL systems [4]. One possible reason for this is the sheer variability possible in NL and the complexity thus required of systems to handle this variability, and a general lack of techniques to talk precisely about NL.

Our informal aim is then that the prototype can accept and use a good selection of the possible ways of presenting the material in this simple development. We do not quantify this condition: it does not seem possible that we could do so in a meaningful way. Hence it is not a strict minimum requirement. Neither is it a maximum requirement. We shall generalise when reasonable and ‘over-implement’ to a certain extent based on our expectations for the next level of the prototype, eg with more complex grammar or more powerful logical facilities. Such considerations also affect our choice of architecture.

2.2 Required Functionality

For this prototype, we take as basis a simple mathematical development: of simple definitions on binary relations, and three proofs concerning those definitions. The development may be briefly summarised as:

- We introduce the notion of binary relations on a type or set.
- We define what it means for a relation to be reflexive, irreflexive, symmetric, and transitive. Two less common properties are also defined: ‘separating’ and ‘apartness’¹.
- The negation of a relation is defined.

¹The former defined as $\forall x, y \in U \cdot (Rxy) \Rightarrow \forall z \in U \cdot (Rxz) \vee (Ryz)$, and the latter that a relation is both separating and irreflexive.

- Theorem 1: the negation of a symmetric relation is symmetric.
- Theorem 2: if there is a ‘top’ element to which all others are related by some relation R , and R is symmetric and transitive, then it is also reflexive.
- Theorem 3: an apartness is symmetric.

We remark:

- It is mathematically simple, hence formalisation is straightforward and it can be expressed in simple natural language. Note that the above statements of the three theorems are the kind of language we would like to implement.
- Versions of this material appear in many text books in some form, eg introductions to discrete mathematics. It has also been formalised under several approaches, including Mizar [13] (comparing their MV to ours will be interesting).
- The mathematical basis is quite fundamental, so we do not need a large ‘library’ of basic concepts first. A key point of MV is that users define the terminology they are about to use. If we need a non-trivial library before studying a development, then the library itself should be used as the case study.
- Language-wise, there is potential for a good range of phenomena, such as mixing symbolic expressions and NL, different ways of referring to properties of relations (ie, some are simple adjectives which have noun forms – eg ‘transitive’ and ‘transitivity’, others as nouns without adjectival forms – eg ‘apartness’), and the proofs are non-trivial enough to admit variations in presentation.
- Technically, the development includes quantifier reasoning, some equality reasoning, an operation on relations (negation), properties defined both independently and in terms of others. The proofs are short and simple.

There is no fixed NL version of this development². In addition to textbook and Mizar presentations, we are conducting an informal empirical study of how people would express the proofs, including how type-theorists would explain the formalised version, how mathematicians would express that piece of mathematics, and how non-experts would do it. Thus, we have access to several independently-produced ways of expressing the ideas in IML, which we can use for development and testing.

2.3 Basic Architecture and Development Method

This discusses how we produce a prototype to satisfy the above aim, in terms of what components and techniques we will use. There are two aspects to this (type theory and natural language), which we discuss separately; then we discuss how we envisage the two interacting. As noted before, we will implement the prototype more generally than needed for short term goals, so in places we discuss techniques which are more complex than immediately required. The prototype will be constructed “bottom up”, ie starting with the type theory levels and then implementing the NL functionality. The main implementation language is the non-strict functional programming language Haskell [6].

Implementing Luo’s LF The type theory aspects will be based on Luo’s typed logical framework LF (see [9], chapter 9). LF has not been implemented in any form before. The plan is to first implement a very basic proof assistant for LF, with a single context, definition mechanisms, basic refinement proof etc. We will then be able to experiment with ways of using type theory, as explained in section 3. A minor goal is to implement an efficient type checker in Haskell which will

²And there is no unique formalisation of the mathematics. This raises the interesting question of how we actually demarcate the development: in a sense, it is the mathematics involved, but in what form is this to be expressed? The various NL and formal versions are just expressions of some underlying idea. What is this underlying form? We leave this question open, since our concern is with how the manifestation in NL corresponds to a manifestation in a formal language (which then may be checked etc), and not with what may or may not be the ‘real’ process in-between.

be suitable for large-scale work, eg by implementing inductive types as a primitive notion, rather than as an extension to an existing system.

Using LF avoids the particularities of specific proof assistants (which are oriented towards helping experts prove things quickly) and of specific type theories (we can use LF to implement just the type theory we require). Thus we have more control over what happens – for example, we don't have to attempt to explain to the user when the proof assistant makes some complicated inference which would not be understandable to a novice type theorist. The LF implementation will also be useful in other projects, eg the coercive subtyping research at Durham.

Processing MV We will use a conventional NL analysis architecture³: essentially this is lexical analysis (normalising and categorising word forms), parsing (building the structure of a sentence), semantics (converting the tree to an expression in the semantic language), pragmatics (checking restrictions are observed and propagating other restrictions), and discourse (connecting the information from several clauses or sentences). Generating replies to the user will be implemented by using simple sentence patterns.

Parsing is (algorithmically) the most complex component: for the moment we shall use an implementation of Tomita's algorithm [20] due to Hopkins [7], as is used in LOLITA. This will allow us to write and parse heavily ambiguous grammars. It is almost certain that syntactic ambiguity will be present. For example, even the simple grammar in Ranta's prototype [16] contains ambiguity. However, a powerful parser introduces problems of (syntactic) disambiguation, of choosing which interpretation to use in further analysis. This will be considered when we get evidence about the behaviour of a realistic grammar.

Semantic and pragmatic interpretation will be implemented as operations on an abstract machine. This will hide detail of the type theory underneath, such as details of how concepts are actually represented. Part of this design is considered in [11]. Discourse will be implemented as a simple post-processor of semantic information.

As to the strategy for populating this infrastructure with rules, we shall start with the basic language sufficient to communicate the development, even if it is cumbersome to use, and then look at adding "surface language" conveniences such as forms of deixis⁴ and more varied syntax.

Particular *new* problems to be solved include: mechanisms for introducing new terminology and for getting that recognised as such even with standard transformations of language⁵; interaction with symbolic expressions – particularly the understanding of these in order to perform semantic checking and interpretation, plus the translation of these to formal notation.

One distinction that may be useful is the language used for definitions vs. that used for proofs. Definitions and statements of theorems must be expressed with precision – they are statements of ideas or concepts in a mathematician's mind, and it is generally impossible to infer missing details. Contrast this with proof, where competent mathematicians can fill in the missing details by inference, thus the communication does not need to be as precise.

How the Prototype will Work The type checker will be used during NL analysis, thus interspersing analysis and interpretation to a degree (the alternative is to perform all type checking once analysis has finished). One consequence is that type information is potentially available for disambiguation – although we need to experiment to see how effective or useful this is. Another consequence is that errors can be detected quickly and the user given useful error messages, rather than just saying 'yes' or 'no' to a sentence.

We would also like to use the type theory level to handle parts of NL analysis where appropriate. For example, anaphora could be attempted purely at the NL level, but some anaphoric references can be interpreted as meta-variables and attempted with the automatic reasoning procedures. (NB especially above sentence level, the factors governing anaphora are heavily dependent on domain

³See [1, 19] for more information.

⁴Indirect references to objects – eg use of pronouns and partial expressions like 'it', "the group", etc. Anaphora is the common case where the referent occurs in previous context.

⁵A simple example: transitivity being introduced as an adjective 'transitive', and later being referred to with the noun 'transitivity'. Where we allow phrases containing more than one word, more complexity is possible, eg "monotonically increasing with respect to x ", "increases monotonically wrt. x ", "monotonically increases" (with the variable left implicit), etc.

factors, such as types of objects – hence this is reasonable.) Coercive subtyping is another good example [11]: we can handle aspects of NL analysis at the type theory level (see section 3.4).

The style of operation is expected as follows. Users will make statements, which are possibly incomplete – eg requiring pre-conditions or additional formal detail. The NL analysis translates such sentences into a representation of this imperfect information, using meta-variables to represent the incompleteness; eg steps in proofs will give rise (internally) to proof terms. Cues from the sentence will be used to classify the meta-variables, and depending on the classification, different kinds of automatic reasoning will be employed to resolve the omissions. For example, statements marked with phrases like “obviously” or “qed” can be viewed as assertions that a proof exists and is easy to obtain in the current mathematical context – automatic reasoning will try to find one by combining the proof steps (ie the internal proof terms) the user has previously described.

3 Technical Issues for Supporting Implementation of MV

This section outlines work in progress on the type theory side of the system. The basis of this is an implementation of Luo’s LF in Haskell, as explained above. We discuss issues of how this type theory is used, ie what support we give the user. These are points which we have also identified as being necessary or extremely useful in an implementation of MV; they are also open questions in contemporary proof assistants.

3.1 Meta-variables

Meta-variables are ‘holes’ or ‘place-holders’ in a proof. They must be replaced with a concrete term before a judgement can be decided. They have known types, but of course the types may contain other meta-variables. There are several uses of meta-variables:

- Productivity: less important details of a proof can be left whilst the key details are explored. For example, proofs which are simple but tedious to do formally can be left, and supplied when the main proof is complete.
- Flexibility in creation of a proof. Lemmas which are found to be useful can be ‘assumed’ for the purposes of a proof, then proved separately when convenient.
- Interface with automation. Meta-variables can be used to represent information which the user believes should be inferable. Hence automation can be used to try to remove (or satisfy) meta-variables.
- For MV, to allow direct expression of NL statements in the formal system. The alternative is to implement some mediator which collects NL information and when possible outputs completed formal expressions. Clearly, use of meta-variables is more flexible.

However, meta-variables cannot be used like normal variables because of logical difficulties. In particular, they should not be abstracted over – they cannot appear in the body of a newly created lambda abstraction, for example. The term used to satisfy a meta-variable must be constructed from the entities available when the meta-variable was introduced. Abstraction (and then application of this abstraction) alters this context – the information that the metavariable could have used some x is lost when x is abstracted and then replaced by some arbitrary term E .

One solution is the restriction that meta-variables are eliminated (ie satisfied) before abstraction occurs. Another solution is to annotate meta-variables with the terms that can be used in their satisfaction – but this is much more complex than the first solution.

Meta-variables will be implemented as variables with restrictions on how they are used, and their elimination as a Cut operation. Cut is, however, an expensive operation – it essentially involves global substitution of the meta-variable, hence a rewrite of the relevant context. This could be made less expensive through lazy evaluation (ie the non-strictness of Haskell), and by the structure of multiple contexts restricting the amount of rewriting needed.

3.2 Multiple Contexts

Most proof assistants implement a *single* context, that is, development proceeds in a single line, and a new proof cannot be started until the previous one is completed or abandoned. Clearly, this is inflexible. However, how to design an alternative is not clear, and like meta-variables, the concept needs careful design. Multiple contexts may have a deep effect on treatment of meta-variables, and vice-versa. Allowing multiple contexts could have several benefits, depending on the scheme chosen:

- It will allow flexible development of proofs – in particular, allowing the user to introduce lemmas freely when needed, or to develop several proofs in parallel. This is very powerful.
- For NL, different contexts of development can be used to represent scoping of names and concepts, eg separating re-use of standard names like x , R .
- Libraries: can be represented as parts of the context, and ‘imported’ when names from them are used. One could also extend or create new libraries more easily. The problem of organising a large development, which includes formation of libraries, is frequently mentioned (eg [3]).
- The structure of the multiple contexts is likely to be a graph which represents dependency between objects of the development (see below), thus it can represent association between objects. This can replace fixed linear structure in type theory to provide a more natural view. For example, a prototype group g can be introduced, and then the components of g introduced and noted as being related to, or dependent on, g . Then, when a theorem based on g is used in another context for another group g' , the user would have to supply objects which are related to g' in analogous fashion.

We are considering the following scheme. A ‘context’ is the set of *objects* which are in scope at a particular time. Objects can be in several scopes simultaneously, eg a lemma which is used in several theorems. Objects in a given context have unique *names*; obviously, identical names in different contexts can refer to different objects. Contexts can be joined by importing objects from other contexts, eg by applying a theorem in the current proof. (Obviously the context used in developing the applied proof should remain hidden.)

Many issues still need to be considered and experimented with. One is how theorems developed inside a multiple contexts framework are used. We could do a conventional ‘abstraction’ or discharge operation, and then reapply to local terms. Note that this operation is not a procedure a mathematician would recognise: they seem to use a Cut-like operation, essentially substituting local terms for the variables in a proof statement. This cut operation seems more natural than abstraction for multiple contexts.

3.3 Questions of Automation

A useful implementation of MV should contain automatic reasoning, in order to support the user. For MV, techniques which help a formal mathematician may not be useful. What we need is techniques to fill in the small details a mathematician would typically omit, and to tie steps of proofs together to prove the whole.

Meta-variables provide ‘hooks’ for automatic reasoning, by representing the places which need attention. We can also differentiate between kinds of meta-variable to provide appropriate treatment for certain cases of reasoning. As mentioned above, the user’s statements will be translated in to terms containing meta-variables, and automation will attempt to build complete proof terms from these, eg when the user claims a proof is complete.

Several kinds of automation will be required, such as type-theoretic model checking [21], or the various procedures implemented in the verification tool PVS [18]. Interfacing to computer algebra systems will also be useful for heavy calculations.

3.4 Use of Coercive Subtyping

Coercive subtyping aids use of a type theory by providing a type-secure abbreviational mechanism [10]. As noted in [11], it can be used to simplify implementation of MV, in particular expressions denoting classes of mathematical object. For example, if ‘finite’ is defined for sets, and ‘group’ is

declared as a subtype of ‘set’, then the construction representing “finite group” is well-typed and immediately understandable as meaning “group whose set is finite”. No further action is necessary, eg NL analysis does not need to make this inference.

Forms of coercive subtyping have been implemented in Lego [2] and Coq [17], but have certain restrictions, such as working on syntactically equal terms rather than computationally equal terms (the latter being more general). Implementing subtyping at a more fundamental level, ie in LF, avoids such restrictions [10]. The implementation will also complement theoretical work on subtyping, eg [8], by helping to explore ideas.

4 Discussion

We have presented the initial design of a prototype for interactive development of mathematics, in terms of a first goal of functionality and the architecture we will use to achieve it. We discuss a few methodological points, and then outline future directions for the project.

4.1 A Bottom-Up Strategy

There are many methodological issues to be considered in an interdisciplinary project like this. For example, how do we define what our requirements are, and how do we evaluate the results, or quantify the worth of the work? De Bruijn noted in his design of an MV that there were many arbitrary decisions to make [5]; what guidance do we have in such cases? These questions are hard to answer.

One way in which we tackle them is by adopting a “bottom-up” strategy of working. That is, we look at pieces of the problem that we can understand, and fit them together to establish a coherent, if limited, whole. The prototype is one example of this. We shall use it to explore further issues. Also bear in mind that our intention is to develop the *technology* for a successful future system, and not to create a system in the short term.

4.2 Future Developments

The chosen development is a modest start, so clearly there are many aspects of mathematics which it does not contain. The following two aspects are the obvious next steps, and will allow us to do more substantial examples.

Algebraic Structures Definition of structures that satisfy certain axioms, and examination of how such structures relate to other structures is a central part of mathematics. An obvious example is of ‘group’. MV must allow us to define the notion of group, and then to use that definition in the usual ways, including access to the component parts and use of the axioms of groups. A particular problem is correct handling of proof terms, especially since these will not be visible to users. We may also allow different axiomatisations of structures; this is useful if mathematicians wish to explore the relationships between axiomatisations. Multiple contexts may support such activity.

Induction We shall concentrate on induction with natural numbers, this being the form most used by mathematicians. We will need to implement the basic language, and the necessary automatic reasoning to support it. Induction is a standard form of argument, so in an interactive system we could prompt the user for the necessary cases, and calculate what those cases are. Note that a proof by induction often begins by the user stating this is the method he will use.

References

- [1] James Allen. *Natural Language Understanding*. Addison Wesley, 1995. (2nd Ed.).
- [2] A. Bailey. Lego with coercion synthesis. <ftp://ftp.cs.man.ac.uk/pub/baileya/-Coercions/LEGO-with-coercions.dvi>, 1996.

- [3] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998. (submitted).
- [4] Paul Callaghan. *An Evaluation of LOLITA and Related Natural Language Processing Systems*. PhD thesis, Department of Computer Science, University of Durham, 1998.
- [5] N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In Nederpelt, Geuvers, and de Vrijer, editors, *Selected Papers on Automath*. 1994.
- [6] Haskell. Haskell report 1.4. <http://www.haskell.org>, 1998.
- [7] Mark Hopkins. Demonstration of the Tomita parsing algorithm. <ftp://iecc.com/pub/file/tomita.tar.gz>, 1993.
- [8] 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.
- [9] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
- [10] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 1998. To appear.
- [11] Z. Luo and P. Callaghan. Mathematical vernacular and conceptual well-formedness in mathematical language. In *Proceedings: Logical Aspects of Computational Linguistics*, 1997. (forthcoming in LNAI series).
- [12] L. Mich. NL-OOPS: From Natural Natural Language to Object Oriented Requirements using the Natural Language Processing System LOLITA. *J. Natural Language Engineering*, 2(2):161–187, 1996.
- [13] Mizar. Mizar WWW Page. <http://mizar.uw.bialystok.pl/>.
- [14] R. Morgan, R. Garigliano, P. Callaghan, S. Poria, M. Smith, A. Urbanowicz, R. Collingham, M. Costantino, C. Cooper, and the LOLITA Group. Description of the LOLITA System as Used in MUC-6. In *Proceedings: The Sixth Message Understanding Conference*, pages 71–87. Morgan Kaufman, Nov 1995.
- [15] A. Ranta. *Type-theoretical Grammar*. Oxford University Press, 1994.
- [16] A. Ranta. A grammatical framework (some notes on the source files), 1997.
- [17] A. Saibi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.
- [18] SRI. The Prototype Verification System. <http://www.csl.sri.com/pvs.html>, 1998.
- [19] The LOLITA Group. *The LOLITA Project*. Springer Verlag. (forthcoming in 3 vols: “The System Core”, “Applications”, “Philosophy and Methodology”).
- [20] M. Tomita. *Efficient Parsing of Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Boston, Ma, 1986.
- [21] S. Yu and Z. Luo. Implementing a model checker for Lego. *Proc. of the 4th Inter Symp. of Formal Methods Europe, FME'97: Industrial Applications and Strengthened Foundations of Formal Methods, Graz, Austria. LNCS 1313*, 1997.