Towards typed repositories of proofs

Matthias Puech
puech@cs.unibo.it
Dept. of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna

Yann Régis-Gianas
yrg@pps.jussieu.fr
Laboratoire PPS, équipe πr^2
University Paris 7, CNRS, and INRIA
23, avenue d'Italie, 75013 Paris

Motivations Practical efforts in the formalization of mathematical results have naturally led to the question of how to manage large repositories of proofs, i.e. what is the *daily workflow* of users of a proof assistant. How does one elaborate a formal proof? What kind of *a posteriori* modification is he prone to doing? What do these modifications imply on the validity of the whole edifice or, the other way around, how does one rely on existing work to build up new results? Many of these questions remain largely unanswered, but the tendency seems to be to adapt existing methods coming from software development, as illustrated for example by the introduction of modules, file-based scripts and separate compilation in proof assistants like Coq [Coq Development Team, 2008] or Matita [Asperti et al., 2007], the use of dependency management tools (Make) or version control system (GIT, Subversion) to build and manage versions of a project. Both for the development of proofs or programs, these tools attempt to cope with the fact that most of a mathematician or programmer's time is actually spent *editing*, not *writing*.

We believe that these tools are not adapted for the new, demanding requirements of proof developments. Indeed, whereas compilation of a program is usually fast enough for the programmer to rely on the usual interaction loop ((edit; compile)*; commit)*, the operation of proof checking is usually too expensive computationally to mimic this workflow. But even beyond the time factor, this "traditional" way of formalizing mathematics hinders the process of mathematical discovery process: once a concept contained in a file is compiled, it is considered frozen and any changes to it require the recompilation of the whole file; the linearity of the development also gives no room for alternate, inequivalent definitions. This fact has nonetheless been shown to be crucial to the mathematical discovery process [Lakatos, 1964], and we believe that they should be taken into account in the realization of mathematical assistants.

In fact, although dedicated tools exist to formalize the description of languages and their metatheory (e.g. Twelf, [Pfenning and Schürmann, 1999]), and substantial formalizations have been undertaken [Lee et al., 2007], we still use legacy tools based on text representation to manage our developments. The general goal exposed here is to replace this tool chain and make it language-aware, both on a syntactic side through the use of abstract syntax trees (AST) instead of concrete syntax, and on a semantic side by using typing to ensure repository consistency.

We propose to discuss a small part of these questions, namely the enhancement and adaptation of version control paradigms to the management of mathematical repositories, to witness with more precision the *impact of changes*. Following the Type Theory approach, our work is based on an algebra of expressive types that are meant to assign precise specifications to object-level term

constructors and, in the meantime, capture fine-grained dependencies between these objects.

We will describe some of the possible directions to develop a tool to analyze the impact of changes through types. It involves at its core a typed description language for repositories, and is strongly related to incremental type-checking: only differences between versions are type-checked and not the entire development. In the first iteration of this project, we focus on a static, or data-driven model for repositories inspired by the repository model of GIT.

A core language to describe typed repositories The kernel of our system is a type-checker algorithm for a typed meta-language. In this language, we will declare both the syntax of the object (proof-)language and its typing rules, and define pieces of syntax (our proofs, potentially with omitted informations) and their derivations (fully explicit application of typing rules). Describing transformations among syntax objects is done by sharing common subterms or subderivations.

Representing syntax and logics is nicely done in a *logical framework* like LF: both the syntactic elements and the typing derivations can sit in the same tree structure, and both can be rechecked at the same time, thanks to dependent types. For the purpose of incremental type checking though, our needs are a little bit different: first, we need to record, that is to name all intermediate values of our developments, so as to be able to address and reuse them multiple times. Secondly, we need to make sure that those intermediate values (sub-terms) are not recorded twice, so as to not type-check them twice: we are looking to represent syntactic and typing objects as a directed acyclic graph (DAG) rather than a tree. Moreover, we enforce by typing a property of *maximal sharing*: every different subterm can be constructed exactly once. Our system will have these properties w.r.t. LF:

- In this first iteration of the project, we do not need computations to take place within our DAGs. Our syntax will then be restricted to product types $(x : t) \cdot t$ and applications;
- Every term should be a flat application of variables $a ::= \vec{x}$, so that we don't introduce compound terms without naming them and recording their types;
- Finally we need a way to record intermediate definitions: we introduce a new kind of binder, the equality binder $(x = a) \cdot t$. We maintain the invariant that no two syntactically equal definition can sit in the context while typing, thus guaranteeing *maximal sharing* among terms and derivations.

We describe the state of a repository at a given moment by a type t in the following syntax. Well-typed types in the repository meta-language guarantee that it contains only well-typed proofs in the object language.

$$\begin{aligned} a &::= x \mid a\ x \\ t &::= a \mid \mathsf{s} \mid (x : t) \cdot t \mid (x = a : t) \cdot t \end{aligned}$$

From an implementation point of view, following GIT, we store these terms in a database of objects indexed by *keys*, which are hash values of their contents, so that retreiving the type of a whole term boils down to compute its hash, and look for it in the database.

Note that in this system, computation has no existence: we only provide ways to verify that syntax and typing rules of an object language are correct. Therefore, patches – functions from repositories to repositories – have no existence *per se*. A necessary enhancement of our theory would be to consider computation as a way to apply patches as if they were constructive metatheorems. We conjecture that it would be done by re-extending the system towards LF (adding abstraction and reduction).

Related work The Twelf project [Pfenning and Schürmann, 1999] is an implementation of the Logical Framework [Harper et al., 1993]. It was used in [Anderson, 1993] to devise transformations of proofs in order to extract efficient programs. Our kernel language reformulates a fragment of LF to make dependencies syntactically explicit.

The problems of managing a formal mathematical library have been dealt with in various proof assistant and mathematical repositories. The HELM project [Asperti et al., 2006] was an attempt to create a large library of mathematics, importing Coq's developments into a searchable and browsable database. Most ideas from this project were imported into the Matita proof assistants [Asperti et al., 2007], especially a mechanism of *invalidation and regeneration* to ensure the global consistency of its library w.r.t changes, with granularity the whole definitions or proofs and their dependencies. The MBase project [Kohlhase and Franke, 2001] attempts at creating a web-based, distributed mathematical knowledge database putting forward the idea of *development graph* [Hutter, 2000, Autexier et al., 2000] to manage changes in the database, allowing semantic-based retrieval and object-level dependency management.

This idea, generalized over structured, semi-formal documents gave birth to *locutor* [Müller and Kohlhase, 2008], a fine-grained extension of the SVN version control system for XML documents, embedding ontology-driven, user-defined semantic knowledge which allows to go across the filesystem border. It embeds a *diff* algorithm, operating on the source text modulo some equality theory to quotient the syntax. On the same line of work, we should mention the *Coccinelle* tool [Padioleau et al., 2008]. It is an evolution over textual patches, specialized on the C language, allowing more flexibility in the matching process, and was developed to deal with the problem of *collateral evolutions* in the Linux kernel. It embeds a declarative language for matching and transforming C source code, operating on text modulo defined isomorphisms.

Our approach to the "impact of changes" problem seems novel on several aspects: first, it applies uniformly on proofs and programming languages by virtue of the Curry-Howard isomorphism, and because we operate at the AST level. Secondly, by taking *types* as witnesses for the evolution of a development, we refine the usual, dependency-based approach for a finer granularity.

References

P. Anderson. *Program Derivation by Proof Transformation.* PhD thesis, CMU, 1993.

A. Asperti, F. Guidi, C.S. Coen, E. Tassi, and S. Zacchiroli. A content based mathematical search engine: Whelp. *Lecture Notes in Computer Science*, 3839:17–32, 2006.

Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reasoning*, 39(2):109–139, 2007.

S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. *Lecture Notes In Computer Science*, pages 73–88, 2000.

The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.2.* INRIA, December 2008.
http://coq.inria.fr/doc-eng.html.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

D. Hutter. Management of change in structured verification. In *Proceedings 15th IEEE International Conference on Automated Software Engineering*, pages 23–34. Citeseer, 2000.

M. Kohlhase and A. Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation*, 32(4):365–402, 2001.

I. Lakatos. Proofs and refutations (IV). *The British Journal for the Philosophy of Science*, 14(56): 296–342, 1964.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 173–184, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: http://doi.acm.org/10.1145/1190216.1190245.

N. Müller and M. Kohlhase. Fine-Granular Version Control & Redundancy Resolution. In *LWA Conference Proceedings (FGWM)*, pages 1–8, 2008.

Y. Padioleau, J. Lawall, R.R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. *ACM SIGOPS Operating Systems Review*, 42(4):247–260, 2008.

F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, pages 202–206, 1999.