Software Engineering for Mathematics

Georges Gonthier

Microsoft Research Cambridge
gonthier@microsoft.com

Despite its mathematical origins, progress in computer assisted reasoning has mostly been driven by applications in computer science, like hardware or protocol security verification. Paradoxically, it has yet to gain widespread acceptance in its original domain of application, mathematics; this is commonly attributed to a "lack of libraries": attempts to formalize advanced mathematics get bogged down into the formalization of an unwieldly large set of basic resuts.

This problem is actually a symptom of a deeper issue: the main function of computer proof systems, checking proofs down to their finest details, is at odds with mathematical practice, which ignores or defers details in order to apply and combine abstractions in creative and elegant ways. Mathematical texts commonly leave logically important parts of proofs as "exercises to the reader", and are rife with "abuses of notation that make mathematics tractable" (according to Bourbaki). This (essential) flexibility cannot be readily accomodated by the narrow concept of "proof library" used by most proof assistants and based on 19th century first-order logic: a collection of constants, definitions, and lemmas.

This mismatch is familiar to software engineers, who have been struggling for the past 50 years to reconcile the flexibility needed to produce sensible user requirements with the precision needed to implement them correctly with computer code. Over the last 20 years *object* and *components* have replaced traditional data and procedure libraries, partly bridging this gap and making it possible to build significantly larger computer systems.

These techniques can be implemented in compuer proof systems by exploiting advances in mathematical logic. *Higher-order logics* allow the direct manipulation of functions; this can be used to assign *behaviour*, such as simplification rules, to symbols, similarly to objects. Advanced *type systems* can assign a secondary, contextual meaning to expressions, using mechanisms such as type classes, similarly to the *metadata* in software components. The two can be combined to perform *reflection*, where an entire statement gets quoted as metadata and then proved algorithmically by some decision procedure.

We propose to use a more modest, *small-scale* form of reflection, to implement *mathematical components*. We use the type-derived metadata to indicate *how* symbols, definitions and lemmas should be used in other theories, and functions to implement this usage — roughly, formalizing some of the *exercize* section of a textbook. We have applied successfully this more engineered approch to computer proofs in our past work on the Four Color Theorem, the Cayley-Hamilton Theorem, and our ongoing long-term effort on the Odd Order Theorem, which is the starting point of the proof of the Classification of Finite Simple Groups (the famous "monster theorem" whose proof spans 10,000 pages in 400 articles).

J. Carette et al. (Eds.): Calculemus/MKM 2009, LNAI 5625, p. 27, 2009.