

Competitive Proving for Fun

Maximilian P. L. Haslbeck and Simon Wimmer*

Technische Universität München
{haslbema,wimmers}@in.tum.de
<http://www.in.tum.de/~{haslbema,wimmers}>

Abstract

We propose a system for large-scale theorem proving contests. We hope that such contests could spark interest in the research field, attract a new generation of theorem proving savants, and foster competition among proof assistants. For the proof assistant Isabelle, we construct and evaluate two iterations of a prototype implementation of our proposed system architecture.

Programming contests such as the Google Code Jam, the ACM ICPC or the International Olympiad in Informatics challenge large numbers¹ of participants to design and implement correct programs to solve algorithmic problems within a short time. A typical problem statement consists of a problem description, and valid input and output pairs. The participant has to come up with an algorithmic idea to solve the given problem and to finally implement the idea in his favourite programming language. The grading is automated: a given solution is compiled and run on a set of unknown test cases. If the participant’s program produces the right answers within a given time the solution is considered correct.

In the formal methods community similar kind of contests exist, for instance the VerifyThis [5] challenge. However, this contest relies on physical presence of participants and manual grading of solutions.

In contrast, we propose a system for *proving contests* that runs online and can grade solutions automatically. Thus the approach is open to participants all over the world and is applicable on a large scale. Judging a formal proof is simpler than testing the correctness of an algorithm: it simply means proof checking, a task that lies at the very heart of proof assistants such as Isabelle [9], PVS [3], or Coq [11].

A typical problem in such a contest would look like this: a problem statement together with definitions for concepts that are used in the problem and propositions to be shown are given to the participant. The task the participants are intended to accomplish is to find a proof for the given proposition but actually they have to convince the proof assistant to accept their proof text.

While most users of a proof assistant would argue that these two tasks are the same in their favourite proof assistant, there are potential pitfalls. We will address some of the obvious ones when describing our first prototype implementation (Section 1) and more subtle ones that arose when evaluating our system (Section 2) with capable users of the chosen proof assistant.

Immediate motivations for carrying out such contests are the following:

- Seeing a team of three, as in the ACM ICPC, find and implement correct solutions for intricate algorithms in a matter of hours is inspiring to many. Similarly, seeing what other people can formally prove within a short amount of time might also spark motivation and interest among students and novices. In that light, one could also hope that companies would start to see the contests as a tool for recruiting, and even preparatory online tutorials (c.f. [7]) or books (c.f. [10]) could come into existence.

*Authors listed in alphabetical order. Haslbeck supported by Grant NI 491/16-1

¹In 2017 Google Code Jam had 25.289 participants in the qualification round [2].

<pre>theory Defs imports Main Primes begin definition pi ("π") where "π n = card {i. prime i ∧ i ≤ n}" save_test.theory end</pre>	<pre>theory Submission imports Defs begin lemma pi: "π (n + 1) - π n = (if prime (n + 1) then 1 else 0)" <proof> end</pre>	<pre>theory Check imports Submission begin test "π (n + 1) - π n = (if prime (n + 1) then 1 else 0)" by (rule Submission.pi) end</pre>
---	--	--

Figure 1: The *definition*, *submission* and *check* theories for a minimal example.

- Competitors can use different proof assistants to participate as long as the problem statements (definition and lemmas) are given in the corresponding logic. This could foster comparability and competition among proof assistants.
- The system can also be used in order to grade classwork by considering each week’s homework a problem. A similar system has proved to be successful for teaching functional programming [4].
- Different proofs for the same theorem (even in different logics) could be compared and grouped considering e.g. length, technical complexity or beauty of the proof idea. The data could be used for machine learning to learn how to guide successful proof, or to extract proof search strategies, for instance.

1 First Prototype Implementation

In this section we describe our first prototype implementation of the theorem proving contest system for Isabelle/HOL. User management, administration of contests, score reporting, and workload balancing among grading servers is realized by the programming contest system DOMJudge². Our implementation mainly needs to provide a grading server for Isabelle/HOL solutions.

Task Layout A task is split into three Isabelle theory files: two provided by the masters of competition (MCs) and one submitted by the competitor. A *definitions* file contains definitions and theorems necessary for the description of the problem. To assess the competitor’s solution, a *check* file contains statements of theorems that should be proved by the competitor and additionally specifies the names of the theorems in the competitor’s solution. It is then checked by a trivial proof method that the correct fact was indeed proved. In case the fact was not proved, Isabelle terminates with an error. The competitors hand in a single *submission* theory file with their solution. See Figure 1 for an example.

This approach works well for assigning a binary grade to a submission. If a submission can contain solutions to multiple tasks that should be graded separately, one can conceive a solution where the MCs provide additional annotations in their check file or where special grading commands are used to compute a list of proved propositions on the Isabelle/ML level. The second iteration, described in section 3, features such a mechanism.

Grading The grading process is started by first loading the definitions file in Isabelle, followed by the submission, and the check file. The final assessment is simple: if Isabelle does not report an error the solution is deemed correct.

We rely on libisabelle [6] to realize the Isabelle grading functionality. It allows to automatically install a specific version of Isabelle (as specified by the MCs) from the internet and to send

²<https://www.domjudge.org/>

theory “snippets” to an Isabelle process via a Scala interface. The concrete implementation poses a number of technical challenges.

Firstly, Isabelle “snippets” need to be sent without the header part of a theory file, thus we need a parser to strip the three files from this information.

Secondly, the MCs should be able to choose which part of the standard library (or the AFP [1]) should be available to competitors and whether the competitors should be allowed (typically not) to import additional theories from the library on their own. The latter option is configured by the MCs, while we need to extract the first information from the theory header in the definitions file. The aforementioned parsing and preparation functionalities are implemented in Python. The prepared input is then passed on to the Scala process.

Moreover, one wants to pre-build Isabelle session images that bundle part of the library to make the grading process efficient. Session images are then stored on the server between grading runs and only the three input files need to be checked when grading a solution. Thus the MCs specify which image their competition relies on and we provide a script that can pre-build a list of standard session images when a grading server is initialized.

Finally, the grading pipeline of DOMJudge does not quite fit our purposes. It expects the MCs to provide a *compilation* script that compiles the submission, a *test runner* script that given the compiled submission can run a test case on it, and a number of test cases. We need to bundle up the definition and check files as single test cases. The compilation script then simply leaves the submission unchanged. The test runner script executes a Python script that first splits the test case into the definitions and the check file and then runs the remainder of our grading pipeline.

Cheating An important consideration is to which amount cheating can and should be prevented. A simple variant of cheating is the use of skipped proofs, as expressed by the *Admitted* keyword in Coq or the *sorry* keyword in Isabelle. In Isabelle this can be detected by a simple textual analysis or in a more reliable way by using its internal tracking of the oracles that a theorem depends on, with *sorry* just being an instance of an oracle that can prove anything.

One can also make the argument that the use of skipped proofs should be allowed in proving competitions. Just as a programming solution can only pass some of the test cases, a formalization with some skipped proofs could receive partial credit. However, automated assessment is harder in this case: the proof of a minor arithmetic manipulation but also the proof of the main proposition could be skipped. Thus a grading scheme that would deduct a number of points for each skipped proof is meaningless in the general case. Despite, we can carve out a use case where one certainly wants to allow skipped proofs: automated grading of homework submissions. A human assessor only needs to inspect the part of the assignment where skipped proofs were used. A technical complication in Isabelle is that a regular build fails whenever *sorry* is used. Thus we introduce a new variant of the keyword that does not show this behaviour but can be identified as an oracle in theorems.

2 First Evaluation

A first internal competition was conducted with our colleagues at the Isabelle group in Munich, i.e. the competitors can all be considered expert users of the system. The results were as expected: great fun with theorem proving was enjoyed, the expert users cracked our simple problems within a number of minutes, and managed to cheat the system in the most creative ways. Among others, the following tricks were used to make our system accept a proof of *False*:

1. The simplest one of them all: one could simply define a new constant *False* as *True*. The system internally distinguishes the two constants with the name *False* but our proposition in the check script would unintentionally refer to the new constant and thus the proof of *False* would be accepted. Other variants of the trick make the system misinterpret the MCs’ intended propositions by introducing coercions or redefining syntax.
2. A sophisticated trick would use the known soundness problem from [8] and turn off the cyclicity checker that usually rectifies the problem.
3. An untrusted code generator translation can be introduced to map *False* to *True*, which allows one to prove *False* by code reflection.

In light of these exploits, we made improvements to our grading system to prevent cheating. Tricks of types two and three can be prevented by using a blacklist of keywords that would allow to introduce new axioms, or to tinker with the Isabelle/ML environment or the code generator setup. The other exploits are remedied by a more sophisticated approach, which relies on the logical context of Isabelle that records existing constants and types (among others). Using a special command (*save_test_theory*) at the end of the definitions file, the current logical context is saved on the Isabelle/ML level. Then, in the check file, we use a modified proposition command (*test*) that parses a term using the saved logical context. This means that for instance *False* would parse to the standard *False* constant of Isabelle/HOL instead of the newly defined one.

Further on the technical side, we realized that DOMJudge is too tightly bound to the “compile-run-check” scheme of grading programming contest submissions and can only be shaped to our needs with quite some effort. Also the sandboxing capabilities of DOMJudge were not suitable for running Isabelle.

3 Second Prototype - Proving for Fun

For a second prototype we implement our own frontend replacing DOMJudge and come up with a more tailor-made backend API.

This API is provided by the contest server to the prover backends and is designed to be generic for any proof assistant: a prover backend polls judgment jobs—which essentially consist of the three theories described above—and replies to the server whether the task was solved correctly. While the API is generic, the prover backends are obviously prover-dependent but follow a common structure. We provide an implementation for Isabelle/HOL which serves as a blueprint for other backends. Experienced users of other proof assistants have used this to come up with backends for Coq, Lean, and HOL4.

Since Isabelle 2018, the *Isabelle Server* [12] provides infrastructure for prover sessions via a client-server connection. This enables a more efficient and convenient way of checking Isabelle/HOL theories, which we now employ instead of libisabelle. We run the Isabelle server within a sandboxed environment using firejail³.

To enable grading with partial credit, we implement the idea sketched above: the users are allowed to use the *sorry* keyword in their submissions, but while checking the theories, points are only awarded for each theorem that was proved without using *sorry* or any other oracle.

The prototype system is available online⁴. It is both used for monthly Isabelle/HOL contests and for homework submission for the Semantics lecture at TUM in the winter term 18/19. See

³<https://firejail.wordpress.com/>

⁴<https://competition.isabelle.systems/>

Position	User	Score	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
1	xxxxx	10.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	xxxxx	9.51	1.0	1.0	0.8	1.0	1.0	1.0	0.71	1.0	1.0	1.0
3	xxxxx	9.51	1.0	1.0	0.8	1.0	1.0	1.0	0.71	1.0	1.0	1.0
4	yyyyy	9.2	1.0	1.0	0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5	xxxxx	9.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0
6	xxxxx	8.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0	0
7	xxxxx	7.76	1.0	1.0	0.6	0.8	1.0	1.0	0.86	1.0	0.5	0
8	xxxxx	7.66	1.0	1.0	0.8	1.0	1.0	0.0	0.86	1.0	1.0	0
9	xxxxx	7.39	1.0	0.83	0.4	0.8	1.0	0.5	0.86	1.0	0.5	0.5
10	xxxxx	6.6	1.0	1.0	0.4	0.2	1.0	0.0	1.0	1.0	1.0	0
11	xxxxx	5.27	1.0	0.67	0.4	0.2	1.0	0.0	0	1.0	0.5	0.5
12	xxxxx	5.23	1.0	0.67	0.2	1.0	0.0	0.0	0.86	1.0	0.5	0
13	xxxxx	5.11	1.0	1.0	0.4	0.0	0.0	0	0.71	1.0	0.5	0.5
14	xxxxx	4.76	1.0	1.0	0.6	0.2	0.0	0	0.71	0.75	0.5	0
15	xxxxx	4.54	1.0	0.83	0	0	0.0	0	0.71	1.0	1.0	0
16	xxxxx	4.19	1.0	0.83	0.4	0.0	0.0	0.0	0.71	0.75	0.5	0

Figure 2: Screenshot of an anonymized leaderboard of a contest at the *Proving for Fun* website.

figure 2 for an example scoreboard: it gives a sorted summary of the participants' aggregated points gained by solving the problems of a contest; partial credits yield fractions of points, and ties are broken by favoring earlier submissions.

Conclusion

Inspired by programming contests, we proposed a system for conducting proving competitions. Refining the idea through two iterations of prototypes, we implemented an online system that is currently in use for teaching and contests in Isabelle/HOL. By designing it in a generic way, we support other proof assistants. We plan to conduct contests involving several proof assistants in order to foster competition and learn from the results.

Acknowledgments

We thank Ondřej Kunčar, Stefan Toman, and Lars Hupel for technical contributions. Also we thank Manuel Eberl, Fabian Immler, and Peter Lammich for participation and their creative ways of abusing our system. We thank Armaël Guéneau, Johannes Hölzl and Mohammad Abdulaziz for providing prover backends for Coq, Lean, and HOL4, respectively.

References

- [1] Archive of Formal Proofs. <https://www.isa-afp.org/>, last viewed January 2019.
- [2] Code jam statistics. <https://www.go-hero.net/jam/17/round/0>, last viewed January 2019.
- [3] PVS homepage. <http://pvs.csl.sri.com/>, last viewed January 2019.
- [4] Jasmin Christian Blanchette, Lars Hupel, Tobias Nipkow, Lars Noschinski, and Dmitriy Traytel. Experience report: The next 1100 haskell programmers. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 25–30, New York, NY, USA, 2014. ACM.
- [5] Marieke Huisman, Vladimir Klebanov, Rosemary Monahan, and Michael Tautschnig. Verifythis 2015. *International Journal on Software Tools for Technology Transfer*, 19(6):763–771, Nov 2017.
- [6] Lars Hupel. larsrh/libisabelle. <https://lars.hupel.info/libisabelle/>, last viewed January 2019.
- [7] Rob Kolstad. USACO training gateway. <http://train.usaco.org/usacogate>, last viewed January 2019.
- [8] Ondřej Kunčar and Andrei Popescu. A consistent foundation for Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 234–252. Springer, 2015.
- [9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [10] Steven S. Skiena and Miguel A. Revilla. *Programming challenges: The programming contest training manual*. Springer Science & Business Media, 2006.
- [11] The Coq development team. The Coq proof assistant. <http://coq.inria.fr/>, last viewed January 2019.
- [12] Makarius Wenzel. The Isabelle System Manual. <https://isabelle.in.tum.de/dist/Isabelle2018/doc/system.pdf>, last viewed January 2019.