System Feature Description: Importing Refutations into the GAPT Framework

Cvetan Dunchev[1], Alexander Leitsch[1], Tomer Libal[1], Martin Riener[1],
Mikheil Rukhaia[1], Daniel Weller[2] and Bruno Woltzenlogel-Paleo[1]

{cdunchev,leitsch,shaolin,riener,mrukhaia,weller,bruno}@logic.at
[1] Institute of Computer Languages (E185)
[2] Institute of Discrete Mathematics and Geometry (E104)
Vienna University of Technology

Abstract

This paper describes a new feature of the GAPT framework, namely the ability to import refutations obtained from external automated theorem provers. To cope with coarse-grained, under-specified and non-standard inference rules used by various theorem provers, the technique of proof replaying is employed. The refutations provided by external theorem provers are replayed using GAPT's built-in resolution prover (TAP), which generates refutations that use only three basic fine-grained inference rules (resolution, factoring and paramodulation) and are therefore more suitable for manipulation by the proof-theoretic algorithms implemented in GAPT.

1 Introduction

GAPT[1] (General Architecture for Proof Theory) is a framework that aims at providing data structures, algorithms and user interfaces for analyzing and transforming formal proofs. GAPT was conceived to replace and expand the scope of the CERES system[2] beyond the original focus on cut-elimination by resolution for first-order logic [BL00]. Through a more flexible implementation based on basic data structures for simply-typed lambda calculus and for sequent and resolution proofs, in the hybrid functional object-oriented language Scala [OSV10], GAPT has already allowed the generalization of the cut-elimination by resolution method to proofs in higher-order logic [HLW11] and to schematic proofs [DLRW12]. Furthermore, methods for structuring and compressing proofs, such as cut-introduction [HLW12] and Herbrand Sequent Extraction [HLWWP08] have recently been implemented.

However, due to GAPT's focus on flexibility and generality, efficiency is only a secondary concern. Therefore, it is advantageous for GAPT to delegate proof search to specialized external automated theorem provers (ATPs), such as Prover9 [McC10a], Vampire [RV02] or Otter[3]. This poses the technical problem of importing proofs from ATPs into GAPT, which is less trivial than it might seem, because different ATPs use different inference rules and some inference rules are too coarse-grained, under-specified, not precisely documented [BW11] and possibly not so standard from a proof-theoretical point of view. As an example, take the explanation of the `rewrite` rule from Prover9's manual [McC10b]:

> `rewrite([38(5,R),47(5),59(6,R)])` – rewriting (demodulation) with equations 38, 47, then 59; the arguments (5), (5), and (6) identify the positions of the rewritten subterms (in an obscure way), and the argument R indicates that the demodulator is used backward (right-to-left).

[1] GAPT: http://code.google.com/p/gapt/

[2] CERES: http://www.logic.at/ceres/

[3] Other well-known and very efficient provers such as E and SPASS have not received much of our attention yet, because their output seemed not as easy to understand and parse.

The use of coarse-grained, under-specified and non-standard inference rules is especially problematic for GAPT, because its proof transformation algorithms require that the proofs adhere to strict and minimalistic calculi, as is usually the case in proof theory. To solve this problem in a robust manner, the technique of proof replaying was implemented in GAPT.

In GAPT's historical predecessor CERES, a more direct translation of each of Prover9's inference rules into pre-defined corresponding sequences of resolution and paramodulation steps had been implemented. Recomputing missing unifiers and figuring out the obscure undocumented ways in which Prover9 assigns numbers to positions made this direct translation particularly hard. Thanks to the technique of proof replaying, these problems were avoided in GAPT.

The main purpose of this paper is to document how GAPT's internal resolution prover (TAP) was extended to support proof replaying and to report our overall experience with this technique. TAP outputs resolution proofs containing only fine-grained resolution, factoring and paramodulation steps, as desired.

Proof replaying is a widely used technique, and the literature on the topic is vast (see e.g. [Fuc97, Amj08, PB10, ZMSZ04, Mei00]). One thing that distinguishes the work presented here is that proofs are replayed into a logical system whose main purpose differs substantially from the typical purposes (e.g. proving theorems, checking proofs) of most logical systems. GAPT's ongoing goal of automating the analysis and transformation of proofs can be seen as complementary and posterior to the goals of most other systems.

The rest of the paper is organized as follows: Section 2 describes the general algorithm for proof replay in our system and explains in more details how we implemented this algorithm for Prover9 output. In Section 3 we give a concrete example. The final section concludes our work and discusses some future improvements.

2 Proof Replaying in GAPT

The aim of this section is to describe how a proof from an external theorem prover is replayed in GAPT. At our disposal is the interactive prover TAP, which implements Robinson's resolution calculus [Rob65] and paramodulation. It is not efficient enough to prove complex theorems, but provided with an external proof it is often able to derive, for each inference step, the conclusion clause from its premises. In principle, if the conclusion clause C is not a tautology, a clause will be derived subsuming C by the forward computation of resolution (see [Lee67]). It works for any calculus with tautology-deletion. For our purposes, the specialized coarse-grained inference steps used in proofs output by optimized theorem provers have to be translated into a series of simple resolution and paramodulation steps. If this series is not too long, TAP will find it and use it instead of the specialized inference rule used by the external prover.

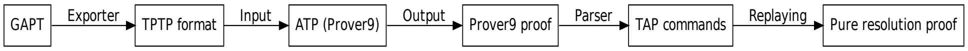

Figure 1: Flowgraph of the Transformation

The complete transformation process is visualized in Figure 1. The theorem to be proved is exported into TPTP format [Sut11, Sut09] and passed to the external theorem prover. For the sake of this paper, we chose to use Prover9 as the external theorem prover. In principle, any prover whose proof output can be parsed by GAPT can be used in place of Prover9, and we plan to add support for more external provers in the future. In the case of a successful result, the proof output is usually a text file, containing for each inference step its ID, a clause which was derived and a list of the rules which were applied[4]. The output file is then parsed into commands which steer TAP to replay the proof.

The API of TAP has two main modules: the search algorithm and the proof replay. The search space consists of elements called configurations. Each configuration consists of a state, a stream of scheduled commands, additional arbitrary data and a result (possibly empty) which is a resolution proof. The state consists of the persistent data of the configuration and might be shared between different configurations. A command transforms a configuration to a list of successor configurations.

The so called "engine function" takes a configuration, executes the first of its scheduled commands and inserts the newly generated configurations into the search space. By default, the prover explores the search space using breadth-first search, but this is configurable.

We now describe the commands used for replay in more detail. In principle, we could replay an input proof purely using the Replay command. However, the actual implementation treats Prover9's inference rules *assumption*, *copy* and *factor* specially because they do not create new proofs and are therefore translated directly into TAP commands. In the first case, a proof of the assumption without premises is inserted into the list of derived clauses. In the second case, the guidance map containing the association of proofs to Prover9's inference identifiers is updated. Factoring is treated as a copy rule, because it is integrated into the resolution rule like in Robinson's original resolution calculus[Rob65]. Therefore we postpone the factoring of a clause until its next use in a resolution step. All other Prover9 inferences are replayed. The commands are grouped into different tasks: initialization, data manipulation and configuration manipulation. Guided commands are a subset of data commands. Table 1 provides an overview over the commands necessary for replay (commands in italics were specifically added for replaying).

Initialization Commands	**Replay Commands**	**Data Commands**
Prover9Init	*Replay*	SetTargetClause
		SetClauseWithProof
Configuration Commands	**Guided Commands**	Variants
SetStream	*AddGuidedInitialClause*	Factor
PrependOnCond	*AddGuidedClauses*	DeterministicAnd
RefutationReached	*GetGuidedClauses*	Resolve
	IsGuidedNotFound	Paramodulation
		InsertResolvent

Table 1: Selection of TAP Commands

The initialization commands interface TAP with an external prover. At the moment, there is only one command handling Prover9. It exports the given clause set to TPTP format, hands it over to Prover9, processes its output with the Prooftrans utility to annotate the inference

[4] Some provers have scripts translating their proof format to XML or TSTP. Since TSTP does not fix the set of inference rules, some adjustments to the replaying have to be made for the different provers. In the actual system, a postprocessing to XML format is used which already separates the inference identifiers from the rule name and the list of clauses, but which does not apply any proof tranformations.

identifiers, clauses and rule names with XML tags and uses Scala's XML library to parse the resulting proof into the system. Each assumption is registered together with its inference ID and put into the set of derived clauses (using AddGuidedInitialClause and InsertResolvent). The copy and the factor rules are treated by adding the proof with the new ID to the guidance map (using AddGuidedClauses). For all other rules, the replay command is issued.

The configuration commands allow control over the proof search process. It is possible to schedule insertion of additional commands into certain configurations and to stop the prover when a (desired) resolution deduction is found.

All the data commands transform a configuration to a (finite) list of successor configurations. A simple example is SetTargetClause which configures with which derived clause to stop the prover. Also the commands for the usual operations of variant generation, factoring, paramodulation and resolution are in this group. It also contains commands to insert a found proof into the set of already found derivations and a command for executing two commands after each other on the same state.

The purpose of the guided commands is the bookkeeping of derived proofs. It allows storage of the proof of a clause in a guidance map which is part of the state. When a guided inference is retrieved, the proof is put into the list of derived clauses within that state. There is also a special command looking for the result of a guided inference and inserting it into the set of derived clauses.

Replaying a rule first needs to retrieve the proofs of the parent clauses from the guidance map. Then it creates a new TAP instance, which it initializes with these proofs as already derived clauses and the reflexivity axiom for equality. The conclusion clause of the inference step to be replayed is set as target clause and the prover is configured to use a strategy which tries alternating applications of the resolution and paramodulation rule on variants of the input clauses. Also forward and backward subsumption are applied after each inference step. In this local instance neither applications of the replay rule nor access to the guidance map is necessary. If the local TAP instance terminates with a resolution derivation, it is put into the global guidance map and returned as part of the configuration. In case TAP can not prove the inference, the list of successor states is empty. Since the scheduled replay commands are consecutive transformations on the same configuration, this also means the global TAP instance will stop without result.

3 An Example

In this section we explain with a simple example how our algorithm works for a concrete proof. Consider the clause set from Figure 2, which was obtained from an analysis of a mathematical proof [BHL$^+$06].

```
cnf( sequent0,axiom,'f'('+'(X1, X0)) = '0' | 'f'('+'(X0, X1)) = '1').
cnf( sequent1,axiom,~'f'('+'(X2, X1)) = '0' | ~'f'('+'('+'('+'(X2, X1), '1'), X0)) = '0').
cnf( sequent2,axiom,~'f'('+'(X2, X1)) = '1' | ~'f'('+'('+'('+'(X2, X1), '1'), X0)) = '1').
```

Figure 2: Example of a clause set in TPTP format

We give this clause set to Prover9, which outputs the refutation given on Figure 3. We see that the information contained in a rule description is incomplete – the unifier is normally left out and the variable names in the resulting clause are normalized. In many cases (such as in the last step) more than one step is applied at once. Clause 22 is rewritten twice into clause 3 (`back_rewrite(3),rewrite([22(2),22(8)])`), yielding two equational tautologies (`xx(a),xx(b)`) which are deleted, resulting in the empty clause.

```
1 f(plus(A,B)) = zero | f(plus(B,A)) = one # label(sequent0) # label(axiom).
[assumption].
2 f(plus(A,B)) != zero | f(plus(plus(plus(A,B),one),C)) != zero # label(sequent1) #
label(axiom).  [assumption].
3 f(plus(A,B)) != one | f(plus(plus(plus(A,B),one),C)) != one # label(sequent2) #
label(axiom).  [assumption].
5 f(plus(A,B)) != zero | f(plus(C,plus(plus(A,B),one))) = one.  [resolve(2,b,1,a)].
11 f(plus(A,plus(plus(B,C),one))) = one | f(plus(C,B)) = one.  [resolve(5,a,1,a)].
16 f(plus(A,B)) = one | f(plus(C,D)) != one.  [resolve(11,a,3,b)].
20 f(plus(A,B)) = one | f(plus(C,D)) = one.  [resolve(16,b,11,a)].
22 f(plus(A,B)) = one.  [factor(20,a,b)].
24 $F. [back_rewrite(3),rewrite([22(2),22(8)]),xx(a),xx(b)].
```

Figure 3: Example of a Prover9 refutation of the clause set

In our approach each (nontrivial) step is translated into a series of commands to the internal prover. The series of commands starts by initializing the prover with only those clauses contributing to the current inference and then schedules the resolution derivation. The last command of the series inserts the proof of the resolution step into the already replayed derivation tree.

In the example above, all steps except the last one are trivial steps and TAP returns exactly the same inferences. For the last step the following command is created:

```
List(ReplayCommand(List(0, 3, 22, 22), 24, ([], [])), InsertResolvent())
```

which says that from the reflexivity predicate 0, clauses 3 and variants of 22 it should derive clause 24, i.e. the empty clause.

A full output of TAP for this example is too big to fit nicely on a page.[5] Since the only interesting case is the last step, Figure 4 displays the corresponding generated resolution derivation – in this case of the empty clause.

4 Conclusion

In this paper we described GAPT's new feature of replaying refutations output by ATPs. Our approach is based on interpreting coarse-grained, under-specified and non-standard inference rules as streams of commands for GAPT's built-in prover TAP. By executing these commands, TAP generates resolution refutations containing only inference rules that are fine-grained and standard enough for GAPT's purposes. This approach is simpler to implement and more robust. The drawback is that its reliance on proof search by a non-optimized prover (TAP) makes replaying less efficient than a direct translation.

In the future, we plan to add support for the TSTP proof format, in order to benefit not only from Prover9 but from any prover using this format. As GAPT's algorithms for proof analysis and proof compression mature, we expect them to be of value in post-processing the proofs obtained by ATPs.

[5] The full output can be found here: `http://code.google.com/p/gapt/wiki/PxTP2012`

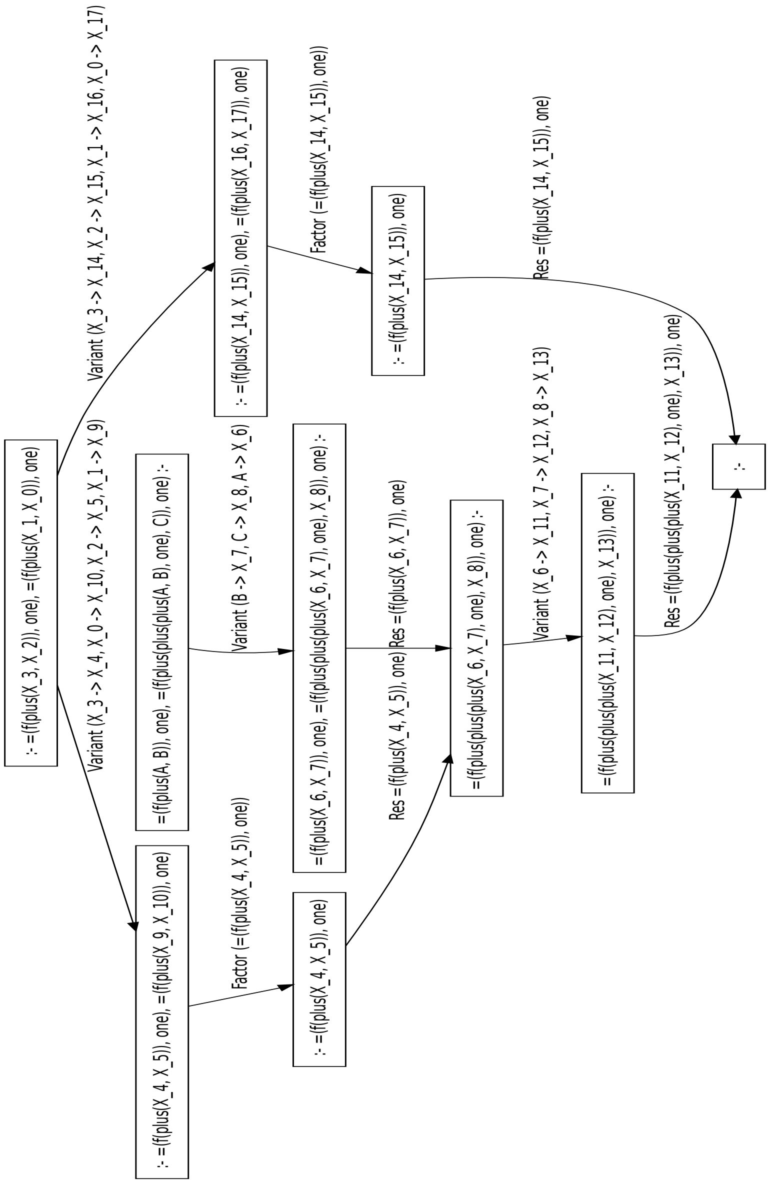
:- =(f(plus(X_3, X_2)), one), =(f(plus(X_1, X_0)), one)
Variant (X_3 -> X_14, X_2 -> X_15, X_1 -> X_16, X_0 -> X_17)
:- =(f(plus(X_14, X_15)), one), =(f(plus(X_16, X_17)), one)
Factor (=(f(plus(X_14, X_15)), one))
:- =(f(plus(X_14, X_15)), one)
Res =(f(plus(X_14, X_15)), one)
Variant (X_3 -> X_4, X_0 -> X_10, X_2 -> X_5, X_1 -> X_9)
=(f(plus(A, B)), one), =(f(plus(plus(plus(A, B), one), C)), one) :-
Variant (B -> X_7, C -> X_8, A -> X_6)
=(f(plus(X_6, X_7)), one), =(f(plus(plus(plus(X_6, X_7), one), X_8)), one) :-
Res =(f(plus(X_6, X_7)), one)
=(f(plus(plus(plus(X_6, X_7), one), X_8)), one) :-
Variant (X_6 -> X_11, X_7 -> X_12, X_8 -> X_13)
=(f(plus(plus(plus(X_11, X_12), one), X_13)), one) :-
Res =(f(plus(plus(plus(X_11, X_12), one), X_13)), one)
:-
:- =(f(plus(X_4, X_5)), one), =(f(plus(X_9, X_10)), one)
Factor (=(f(plus(X_4, X_5)), one))
:- =(f(plus(X_4, X_5)), one)
Res =(f(plus(X_4, X_5)), one)

Figure 4: Replayed resolution tree of the last step of the example

References

[Amj08] Hasan Amjad. Data compression for proof replay. *Journal of Automated Reasoning*, 41:193–218, 2008.

[BHL+06] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Proof transformation by CERES. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management (MKM) 2006*, volume 4108 of *Lecture Notes in Artificial Intelligence*, pages 82–93. Springer, 2006.

[BL00] Matthias Baaz and Alexander Leitsch. Cut-elimination and redundancy-elimination by resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.

[BW11] Sascha Böhme and Tjark Weber. Designing proof formats: A user's perspective. In *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving*, 2011.

[DLRW12] C. Dunchev, A. Leitsch, M. Rukhaia, and D. Weller. About Schemata And Proofs web page, 2010–2012. `http://www.logic.at/asap`.

[Fuc97] Marc Fuchs. Flexible proof-replay with heuristics. In Ernesto Coasta and Amilcar Cardoso, editors, *Progress in Artificial Intelligence*, volume 1323 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1997.

[HLW11] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. CERES in higher-order logic. *Annals of Pure and Applied Logic*, 162(12):1001–1034, 2011.

[HLW12] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. Towards algorithmic cut-introduction. In *LPAR*, pages 228–242, 2012.

[HLWWP08] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Herbrand sequent extraction. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 462–477. Springer Berlin, 2008.

[Lee67] Char-Tung Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms*. PhD thesis, 1967. AAI6810359.

[McC10a] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[McC10b] W. McCune. Prover9 and mace4 manual - output files, 2005–2010. `https://www.cs.unm.edu/~mccune/mace4/manual/2009-11A/output.html`.

[Mei00] Andreas Meier. System description: TRAMP - Transformation of machine-found proofs into natural deduction proofs at the assertion level. In *Proceedings of the 17th International Conference on Automated Deduction, number 1831 in Lecture Notes in Artificial Intelligence*, pages 460–464. Springer-Verlag, 2000.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima, Inc., 2nd edition, 2010.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Eugenia Ternovska, and Stephan Schulz, editors, *Proceedings of the 8th International Workshop on the Implementation of Logics*, 2010.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of Vampire. *AI Commun.*, 15(2,3):91–110, 2002.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Sut11] G. Sutcliffe. TPTP Syntax EBNF, 2011. `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`.

[ZMSZ04] J. Zimmer, A. Meier, G. Sutcliffe, and Y. Zhang. Integrated proof transformation services. In *Workshop on Computer-Supported Mathematical Theory Development*, 2004.