System Description: GAPT 2.0*

Gabriel Ebner[1], Stefan Hetzl[1], Giselle Reis[2], Martin Riener[3],
Simon Wolfsteiner[1], and Sebastian Zivota[1]

[1] Vienna University of Technology
[2] Inria & LIX/École Polytechnique
[3] Inria & MSR-Inria Joint Centre, Saclay

Abstract. GAPT (General Architecture for Proof Theory) is a proof theory framework containing data structures, algorithms, parsers and other components common in proof theory and automated deduction. In contrast to automated and interactive theorem provers whose focus is the construction of proofs, GAPT concentrates on the transformation and further processing of proofs. In this paper, we describe the current 2.0 release of GAPT.

1 Introduction

This paper describes the system GAPT (General Architecture for Proof Theory). GAPT is a versatile proof theory framework containing data structures, algorithms, parsers and other components common in proof theory and automated deduction. In contrast to automated and interactive theorem provers whose focus is the construction of proofs, GAPT concentrates on the transformation and further processing of proofs.

We are convinced that such a system is of importance to computational proof theory and automated deduction because of the growing interest in the *output* of provers. It is no longer enough for a prover to answer with yes or no as often a proof object (or a countermodel) is sought for further processing. For example, the use of SAT-solvers for solving various problems in NP needs the solver to return a propositional interpretation or a refutation as certificate of unsatisfiability. The use of interpolation in software verification needs proofs (or interpolants) as output. The use of automated reasoning systems in proof assistants—e.g., Sledgehammer in Isabelle—needs to provide proofs to incorporate in the proof script. This change in role of automated theorem provers is also reflected in the growing interest in proof certificates, e.g., in research projects like ProofCert [16], Dedukti [4], in common formats for proofs shared between provers like TPTP derivations [21] and OpenTheory [13] and in conference series like Certified Programs and Proofs (CPP). It is also reflected in CASC (the CADE ATP System Competition) evaluating theorem provers considering the number of problems solved presenting a solution, i.e. a proof [20].

* Supported by the Vienna Science Fund (WWTF) project VRG12-04, the Austrian Science Fund (FWF) projects P25160 and W1255-N23, and the ERC Advanced Grant *ProofCert*

GAPT provides a rich reservoir of functionality for the transformation and further processing of formal proofs in a uniform framework. GAPT contains interfaces to a variety of automated reasoning systems including first-order provers, SAT-solvers and SMT-solvers. Thus it provides a platform which is well-suited not only for computational proof theory but also for the cooperation of automated provers.

GAPT has been used as an environment to experiment with the implementation of several specific algorithms and tools: cut elimination by resolution [3,10,2], post-processing of resolution proofs [12,14], cut introduction [7,11,8,9], and inductive theorem proving based on tree grammars [6]. In addition to these applications, the graphical user interface has been described in detail in [5,14] and GAPT's use of expansion trees for proof import in [17]. Using a single system for these applications had a synergistic effect since all these algorithms share a common basis. This basis has been developed and extended into the GAPT system which has now reached a level of maturity to be of interest for its own sake. We mark this occasion by the release of version 2.0[4] and the first system description of GAPT as a whole. GAPT is implemented in Scala and licensed under the GNU General Public License. It is available at

`https://logic.at/gapt`

2 Features

Formulas. Terms and formulas are uniformly represented as expressions in a simply typed lambda calculus with multiple base sorts. This representation allows considerable code reuse: for example, substitutions are only defined once for terms, atoms, formulas, etc. While these are all represented as lambda expressions, they are each instance of a more specific Scala type as well: `FOLAtom` is a subtype of `HOLFormula`, which is in turn a subtype of `LambdaExpression`. These Scala types are determined at run-time using smart constructors. In this way, we support type-safe programming with defined subsets of `LambdaExpression`. GAPT allows arbitrary Unicode strings as names for constants, variables, predicate symbols, etc.

Proofs. GAPT contains an implementation of a standard sequent calculus LK for classical higher-order logic as well as a version of the sequent calculus using Skolem terms instead of eigenvariables (LKsk, see [10] for details). In addition, it contains resolution calculi: $\mathcal{R}_{al}$ (see [10]) which is a labelled variant of Andrew's $\mathcal{R}$ [1] and a standard first-order resolution calculus. GAPT also contains expansion proofs [15], a generalisation of the notion of Herbrand-disjunction to arbitrary formulas in higher-order logic. The proof objects in these calculi are automatically validated during the construction of each inference, preventing ill-formed proofs. This eager validation has been highly valuable for the early

[4] For a list of changes and new features in the 2.0 release specifically, please refer to the release notes: `https://github.com/gapt/gapt/blob/master/RELEASE-NOTES.md`

detection of bugs. Our main focus is on tree-like proofs in first and higher-order logic, these usually have less than 1000 inferences. In resolution, GAPT can work with dag-like proofs of about 10000 inferences.

Algorithms. GAPT contains a number of basic algorithms like transformations between the above-mentioned proof calculi, Skolemisation and regularisation of sequent calculus proofs, naive and structural first-order clause normal form transformations, proof pruning, etc. More advanced algorithms include: Gentzen-style cut elimination in the sequent calculus, interpolation in first-order proofs and a built-in tableaux prover for (classical) propositional logic as a quick way to generate propositional sequent calculus proofs.

First-order theorem proving. GAPT interfaces with several first-order theorem provers: it can invoke and import proofs from Vampire, the E prover, and Prover9. There is specific proof import code for Prover9, which successfully imports more than 99% of the Prover9 solutions in the TSTP [19] as GAPT resolution proofs. In addition, there is a general purpose import for TPTP-proofs based on proof replay, which currently imports 34% of the FOF and CNF solutions in the TSTP from a total of 12 different provers. We are currently also developing leanCoP-specific import code [17] to have reliable import for non-resolution first-order provers.

SAT- and SMT-solving. GAPT is able to export formulas as SMT-LIB benchmarks and can check their satisfiability modulo `QF_UF` with an arbitrary SMT-LIB compliant SMT-solver. This interface natively supports many-sorted logic, and works with at least Z3, CVC4, and veriT. Proof import is implemented for the `QF_UF` logic for veriT, see [17]. For propositional formulas, GAPT writes DIMACS files and can use any DIMACS-compliant SAT-solver to check their satisfiability and import satisfying assignments. We support Glucose, Sat4j, and miniSAT out of the box—adding support for other solvers usually only requires specifying the executable path. In addition, GAPT also provides an interface to solvers for the MaxSAT optimization problem, such as OpenWBO and the MaxSAT solver in Sat4j.

User interfaces. GAPT comes with two user interfaces: the system's full functionality is available via a customised Scala shell, thus providing a flexible and scriptable command-line interface. In addition, GAPT provides a graphical user interface, prooftool, to conveniently display large proofs and other objects. For example, prooftool also includes a viewer for expansion trees with a point-and-click interface to selectively expand quantifiers, see [12]. Large proofs in LK can be visualized using a so-called Sunburst viewer [14]. Sunburst visualisations are radial, space-filling representations of hierarchical information [18]: instead of a tree, the inferences in a proof are displayed as concentric rings.

These logics, proof systems and interfaces with other provers are not intended to be a final fixed set of features in GAPT. The system's architecture allows the implementation of extensions, so new logics and proof systems can be added as they become necessary while having a versatile collection of tools readily available for tests and analysis.

3 Example

Figure 1 shows a first-order prover utilising the GAPT API. This example is not meant to implement a practically relevant, efficient, or short prover, but to illustrate the features provided by GAPT. The prover continuously generates new instances of clauses in the input clause set by unifying literals of opposite polarity. The `done` set contains the clauses where the pairwise unifiers have already been computed, in each iteration we pick a clause from the `todo` queue and unify it with all clauses in `done`. When the set of instances becomes propositionally unsatisfiable (which we check using the Sat4j SAT solver[5]), we minimize the number of instances using `minimalExpansionSequent` and convert the instances to a proof in LK using `ExpansionProofToLK`. The resulting proof is then displayed in a GUI window using `prooftool`.

Utilising the functionality already provided by GAPT, we can concentrate on the actual algorithm, while the interface and "glue code" is already implemented for us, such as:

- formula parsing
- robust structural clausification (including Skolemisation)
- unification, matching and substitution
- SAT solver interface
- proof construction (and validation)
- proof simplification
- graphical visualisation of the resulting proof

This example prover can be immediately executed from the binary distribution of GAPT[6], without installing any other extra dependencies. It will read the problem from standard input, refute it, and then open the resulting proof in the graphical user interface:

```
./gapt.sh instprover.scala <example.in
```

This usage of GAPT scripts is convenient for early prototyping. But should our prototype develop into a larger project, we are not stuck with developing it as a single file. Since GAPT is available as a Scala library from the JCenter repository, it can be added as a dependency for another project by adding a single line to its `sbt` build script. This way, we can seamlessly move from a small prototype to a full-fledged separate project.

[5] We use Sat4j as it is bundled with GAPT. To use another solver, it is enough to replace `Sat4j` with `Glucose` or `MiniSAT` in the source code.

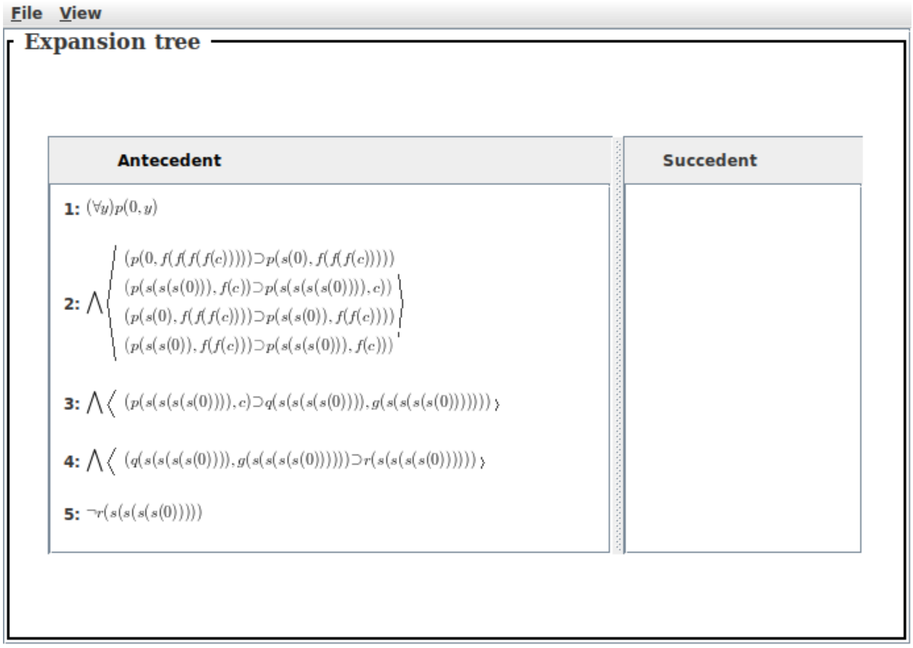
[6] This example is included in the `examples/scriptability` directory in the binary distribution of GAPT.

```
import scala.collection.mutable

// Parse input
val sequent = Stream.continually(Console.in.readLine()).
  takeWhile(_ != null).map(_.trim).filter(_.nonEmpty).
  map(parseFormula).map(univclosure(_)) ++: Sequent()

// Transform into clause normal form
val (cnf, justifications, definitions) = structuralCNF(sequent,
    generateJustifications = true, propositional = false)

// Main loop
val done = mutable.Set[FOLClause]()
val todo = mutable.Queue[FOLClause](cnf.toSeq: _*)
while (Sat4j solve (done ++ todo) isDefined) {
  val next = todo.dequeue()
  if (!done.contains(next)) for {
    clause2 <- done
    clause1 = FOLSubstitution(
      rename(freeVariables(next), freeVariables(clause2)))(next)
    (atom1,index1) <- clause1.zipWithIndex.elements
    (atom2,index2) <- clause2.zipWithIndex.elements
    if !index2.sameSideAs(index1)
    mgu <- syntacticMGU(atom1, atom2)
  } todo ++= Seq(mgu(clause1), mgu(clause2))
  done += next
}
// Postprocessing
val instances = for (clause <- cnf) yield clause ->
  (for { inst <- done ++ todo
         subst <- syntacticMatching(
           clause.toFormula, inst.toFormula)
       } yield subst).toSet
val expansion = expansionProofFromInstances(
  instances.toMap, sequent, justifications, definitions)
val Some(minimized) = minimalExpansionSequent(expansion, Sat4j)
val lkProof = ExpansionProofToLK(minimized)

// Visualisation
prooftool(lkProof)
```

Fig. 1. instprover.scala: Instantiation-based first-order prover with graphical proof output

```
p(0,y)  &  (p(x,f(y)) -> p(s(x),y))
(p(x,c) -> q(x,g(x)))  &  (q(x,y) -> r(x))  &  -r(s(s(s(s(0)))))
```

Fig. 2. example.in: Example input for the prover from Figure 1



1: $(\forall y)p(0,y)$

2: $\bigwedge\left\langle \begin{array}{l} (p(0,f(f(f(f(c)))))\supset p(s(0),f(f(f(c))))) \\ (p(s(s(s(0))),f(c))\supset p(s(s(s(s(0)))),c)) \\ (p(s(0),f(f(f(c))))\supset p(s(s(0)),f(f(c)))) \\ (p(s(s(0)),f(f(c)))\supset p(s(s(s(0))),f(c))) \end{array}\right\rangle$

3: $\bigwedge\langle (p(s(s(s(s(0)))),c)\supset q(s(s(s(s(0)))),g(s(s(s(s(0))))))) \rangle$

4: $\bigwedge\langle (q(s(s(s(s(0)))),g(s(s(s(s(0))))))\supset r(s(s(s(s(0)))))) \rangle$

5: $\neg r(s(s(s(s(0)))))$

Fig. 3. Graphical visualisation of the resulting proof using the expansion tree viewer.

4 Applications

We have used GAPT primarily as a basis for prototype implementations of newly developed algorithms. We briefly review these applications here, highlighting the aspects of the GAPT-system which are of particular relevance.

Cut elimination by resolution (CERES). This is a method for cut elimination which is based on using a resolution theorem prover to generate a skeleton structure for a cut-free proof. This method has been applied to show that Fürstenberg's topological proof of the infinity of primes can be transformed into Euclid's original proof by cut elimination [2]. The CERES method depends heavily on several non-trivial proof transformation like the Skolemisation of proofs with cut or the combination of a resolution refutation of a clause set $\mathcal{C}$ with cut-free sequent calculus proofs ψ_C of $\Gamma \vdash \Delta \circ C$ for $C \in \mathcal{C}$ to a sequent calculus proof with only atomic cuts. To analyse the results produced by CERES, expansion tree extraction and visualisation is used.

Cut introduction. GAPT has been used as basis for the implementation of a method for cut introduction (i.e., lemma generation) [11,8,9]. This method is based on a structural analysis of expansion trees using tree grammars. It relies heavily on the flexible use of expansion trees and on the interface to MaxSAT-solvers which are used to compute minimal tree grammars. Hence GAPT also contains an implementation of some tree grammars. As database for the testing and evaluation this algorithm, the TSTP and the reliable Prover9 import have proved indispensable.

Inductive theorem proving based on tree grammars. Currently, GAPT is used for a prototype implementation of an inductive theorem prover based on the method described in [6]. This being a generalisation of the method for cut introduction to induction, it also benefits from the availability of proof transformations and the flexible handling of expansion trees and tree grammars as described above. In addition, for this application, the use of resolution provers and SMT-solvers for generating instance proofs is necessary.

Teaching. GAPT is used, along with several automated theorem provers, in a graduate course on automated deduction taught at the Vienna University of Technology. The students are asked to perform various computational experiments relating run-time, size of output, and other parameters of various algorithms. For example: naive clause form transformation by distributivity vs. Tseitin transformation, a SAT-solver on sequences of propositional tautologies of varying proof complexity, a first-order resolution prover vs. a SAT-solver on a propositional clause set and on ground instances of a first-order clause set. Such comparisons crucially rely on having a uniform framework with interfaces to different automated reasoning systems.

5 Future Work and Conclusion

We are currently implementing a tactics language for the more convenient input of formal proofs which will make it into the next release. A built-in superposition-based theorem prover will be included in the next release as well, enabling more efficient proof replay without external dependencies. As further future work, we plan to implement support for a wider variety of different proof calculi (e.g., natural deduction) and logics (e.g., intuitionistic logic). In addition, we are looking to extend the existing support for multiple uninterpreted base sorts to interpreted sorts such as integers and arrays, and interface them with the built-in theories of SMT solvers. We will continue to use the system for the applications described in Section 4.

The power of GAPT comes from the integration of a wide variety of different systems (e.g. SAT-solvers, SMT-solvers, resolution and connection provers) and the flexibility of combining them using a large number of standard algorithms and transformations, all within one uniform framework. GAPT is developed in Scala which, on the one hand, permits elegant functional code close to mathematical definitions, but on the other hand also provides access to the whole Java library, including, e.g., Swing, on which prooftool is based. GAPT has already proved very useful for the development of and experiments with new algorithms in computational proof theory and automated deduction and we are convinced that it will continue to do so.

While GAPT already interfaces with a large number of external provers, we always try to expand our support to other provers. As next steps we plan to add support for the DRUP format used by SAT solvers, and to add proof import for first-order provers that employ inferences rules that go beyond the standard resolution calculus, such as the splitting rule in SPASS. Adding support for a new prover takes a considerable amount of work, ranging from minute details such as recognizing different headers in the output files to supporting new proof systems. We hope that GAPT will benefit from further efforts in the standardisation of proof output.

Acknowledgements. The authors would like to thank the following students, researchers, and software developers for their contributions to the development of GAPT (in alphabetic order): Alexander Birch, Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Bernhard Mallinger, Olivier Roland, Mikheil Rukhaia, Christoph Spörk, Janos Tapolczai, Daniel Weller, and Bruno Woltzenlogel Paleo.

References

1. Andrews, P.B.: Resolution in Type Theory. Journal of Symbolic Logic 36(3), 414–432 (1971), `http://dx.doi.org/10.2307/2269949`
2. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: CERES: An Analysis of Fürstenberg's Proof of the Infinity of Primes. Theoretical Computer Science 403(2–3), 160–175 (2008)
3. Baaz, M., Leitsch, A.: Cut-elimination and Redundancy-elimination by Resolution. Journal of Symbolic Computation 29(2), 149–176 (2000)
4. Boespflug, M., Carbonneaux, Q., Hermant, O.: The $\lambda\Pi$-calculus modulo as a universal proof language. In: Pichardie, D., Weber, T. (eds.) Proceedings of PxTP2012: Proof Exchange for Theorem Proving. pp. 28–43 (2012)
5. Dunchev, C., Leitsch, A., Libal, T., Riener, M., Rukhaia, M., Weller, D., Paleo, B.W.: PROOFTOOL: a GUI for the GAPT framework. In: Kaliszyk, C., Lüth, C. (eds.) Proceedings 10th International Workshop On User Interfaces for Theorem Provers (UITP) 2012. EPTCS, vol. 118, pp. 1–14 (2012)
6. Eberhard, S., Hetzl, S.: Inductive theorem proving based on tree grammars. Annals of Pure and Applied Logic 166(6), 665–700 (2015)
7. Hetzl, S.: Project Presentation: Algorithmic Structuring and Compression of Proofs (ASCOP). In: et al., J.J. (ed.) Conference on Intelligent Computer Mathematics (CICM) 2012. Lecture Notes in Artificial Intelligence, vol. 7362, pp. 437–441. Springer (2012)
8. Hetzl, S., Leitsch, A., Reis, G., Tapolczai, J., Weller, D.: Introducing Quantified Cuts in Logic with Equality. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Automated Reasoning - 7th International Joint Conference, IJCAR. Lecture Notes in Computer Science, vol. 8562, pp. 240–254. Springer (2014)
9. Hetzl, S., Leitsch, A., Reis, G., Weller, D.: Algorithmic introduction of quantified cuts. Theoretical Computer Science 549, 1–16 (2014)
10. Hetzl, S., Leitsch, A., Weller, D.: CERES in Higher-Order Logic. Annals of Pure and Applied Logic 162(12), 1001–1034 (2011)

11. Hetzl, S., Leitsch, A., Weller, D.: Towards Algorithmic Cut-Introduction. In: Logic for Programming, Artificial Intelligence and Reasoning (LPAR-18). Lecture Notes in Computer Science, vol. 7180, pp. 228–242. Springer (2012)
12. Hetzl, S., Libal, T., Riener, M., Rukhaia, M.: Understanding Resolution Proofs through Herbrand's Theorem. In: Galmiche, D., Larchey-Wendling, D. (eds.) Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX) 2013, Proceedings. Lecture Notes in Computer Science, vol. 8123, pp. 157–171. Springer (2013)
13. Hurd, J.: The OpenTheory standard theory library. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) Third International Symposium on NASA Formal Methods (NFM 2011). Lecture Notes in Computer Science, vol. 6617, pp. 177–191. Springer (2011)
14. Libal, T., Riener, M., Rukhaia, M.: Advanced Proof Viewing in ProofTool. In: Benzmüller, C., Paleo, B.W. (eds.) Proceedings of the 11th Workshop on User Interfaces for Theorem Provers (UITP) 2014. EPTCS, vol. 167, pp. 35–47 (2014)
15. Miller, D.: A Compact Representation of Proofs. Studia Logica 46(4), 347–370 (1987)
16. Miller, D.: Proofcert: Broad spectrum proof certificates (Feb 2011), `http://www.lix.polytechnique.fr/Labo/Dale.Miller/ProofCert.pdf`, an ERC Advanced Grant funded for the five years 2012-2016
17. Reis, G.: Importing SMT and Connection proofs as expansion trees. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving (PxTP). EPTCS, vol. 186, pp. 3–10 (2015)
18. Stasko, J., Zhang, E.: Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In: Information Visualization, 2000. InfoVis 2000. IEEE Symposium on. pp. 57–65 (2000)
19. Sutcliffe, G.: The TPTP World – Infrastructure for Automated Reasoning. In: Clarke, E., Voronkov, A. (eds.) Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 1–12. No. 6355 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2010)
20. Sutcliffe, G., Suttner, C.: The State of CASC. AI Communications 19(1), 35–48 (2006)
21. Sutcliffe, G., Schulz, S., Claessen, K., Gelder, A.V.: Using the TPTP Language for Writing Derivations and Finite Interpretations. In: Furbach, U., Shankar, N. (eds.) Proceedings of the 3rd International Joint Conference on Automated Reasoning. Lecture Notes in Artificial Intelligence, vol. 4130, pp. 67–81. Springer (2006)