Teaching Automated Theorem Proving by Example: PyRes 1.2 (system description)

Stephan Schulz[1] and Adam Pease[2]

[1] DHBW Stuttgart, Germany, `schulz@eprover.org`
[2] Articulate Software, USA, `apease@articulatesoftware.com`

Abstract. PyRes is a complete theorem prover for classical first-order logic. It is not designed for high performance, but to clearly demonstrate the core concepts of a saturating theorem prover. The system is written in extensively commented Python, explaining data structures, algorithms, and many of the underlying theoretical concepts. The prover implements binary resolution with factoring and optional negative literal selection. Equality is handled by adding the basic axioms of equality. PyRes uses the given-clause algorithm, optionally controlled by weight- and age evaluations for clause selection. The prover can read TPTP CNF/FOF input files and produces TPTP/TSTP proof objects.
Evaluation shows, as expected, mediocre performance compared to modern high-performance systems, with relatively better performance for problems without equality. However, the implementation seems to be sound and complete.

1 Introduction

Modern automated theorem provers for first order logic such as E [8,7], Vampire [3], SPASS [12] or iProver [2] are powerful systems. They use optimised data structures, often very tight coding, and complex work flows and intricate algorithms in order to maximise performance. Moreover, most of these programs have evolved over years or even decades. As a result, they are quite daunting for even talented new developers to grasp, and present a very high barrier to entry. On the other hand, minimalist systems like leanCoP [5] do not represent typical current ATP systems, in calculus, structure, or implementation language.

Textbooks and scientific papers, on the other hand, often leave students without a clear understanding of how to translate theory into actual working code.

With PyRes, we try to fill the gap, by presenting a sound and complete theorem prover for first order logic based on widely used calculus and architecture, that is written in an accessible language with a particular focus on readability, and that explains the important concepts of each module with extensive high-level comments. We follow an object oriented design and explain data structures and algorithms as they are used.

PyRes consists of a series of provers, from a very basic system without any optimisations and with naive proof search to a prover for full first-order logic with some calculus refinements and simplification techniques. Each variant gracefully extends the previous one with new concepts. The final system is a saturation-style theorem prover based on Resolution and the given-clause algorithm, optionally with CNF transformation and subsumption.

The system is written in Python, a language widely used in education, scientific computing, data science and machine learning. While Python is quite slow, it supports coding in a very readable, explicit style, and its object-oriented features make it easy to go from more basic to more advanced implementations.

Students have found PyRes very useful in getting a basic understanding of the architecture and algorithms of an actual theorem prover, and have claimed that it enabled them to come to grips with the internals of E much faster than they could have done otherwise.

PyRes is available as open source/free software, and can be downloaded from `https://github.com/eprover/PyRes`.

2 Preliminaries

We assume the standard setting for first-order predicate logic. A signature consists of finite sets P (of predicate symbols) and F (of function symbols) with associated arities. We write e.g. $f/n \in F$ to indicate that f is a function symbol of arity n. We also assume an enumerable set $V = \{X, Y, Z, \ldots\}$ of variables. Each variable is a *term*. Also, if $f/n \in F$ and $t_1, \ldots, t_n$ are terms, then so is $f(t_1, \ldots, t_n)$. This includes the special case of constants (function symbols with arity 0), for which we omit the parentheses. An *atom* is composed similarly from $p/n \in P$ and n terms. A *literal* is either an atom, or a negated atom. A *clause* is a (multi-)set of literals, interpreted as the universal closure of the disjunction of its literals and written as such. As an example, $p(X, g(a))$ is an atom (and a literal), $\neg q(g(X), a)$ is a literal, and $p(X, g(a)) \vee \neg q(g(X), a) \vee p(X, Y)$ is a three-literal clause. A first-order formula is either an atom, or is composed of existing formulas F, G by negation $\neg F$, quantification ($\forall X : F$ and $\exists X : F$), or any of the usual binary Boolean operators ($F \vee G$, $F \wedge G$, $F \rightarrow G$, $F \leftrightarrow G, \ldots$). We assume a reasonable precedence of operators and allow the use of parentheses where necessary or helpful. A substitution is a mapping from variables to terms, and is continued to terms, atoms, literals and clauses in the obvious way. A *match* from s onto t is a substitution σ such that $\sigma(s) = t$ (where s and t can be terms, atoms, or literals). A *unifier* is similarly a substitution σ such that $\sigma(s) = \sigma(t)$. Of particular importance are *most general unifiers.* If two terms are unifiable, a most general unifier is easy to compute, and, up to the renaming of variables, unique. We use $mgu(s, t)$ to denote the most general unifier of s and t

PyRes implements standard resolution as described in [6], but like most implementations, it separates resolution and factoring. It also optionally adds negative literal selection. This refinement of resolution allows the *selection* of an arbitray negative literal in clauses that have at least one negative literal, and the restriction of resolution inferences involving this clause to those that resolve on the selected literal [1]. PyRes also supports subsumption, i.e. the discarding

$$\text{(BR)}\frac{C \vee A \qquad D \vee \neg B}{\sigma(C \vee D)} \text{ if } \sigma = mgu(A,B) \qquad \text{(BF)}\frac{C \vee L \vee M}{\sigma(C \vee L)} \text{ if } \sigma = mgu(L,M)$$

$$\text{(CS) } \frac{C \qquad \sigma(C) \vee D}{\frac{}{C}}$$

A, B stand for atoms, L, M stand for literals, and C, D are arbitrary clauses.
If negative literal selection is employed, additional constraints to (BR) are that $\neg B$ is *selected*, and that no literal in C is selected. (CS) is a contraction rule, i.e. it *replaces* the premises by the conclusion, in effect removing the larger clause.

Fig. 1. Binary resolution with subsumption

of clauses covered by a more general clause. The inference system, consisting of *binary resolution* (BR), *binary factoring* (BF) and *clause subsumption* (CS) is shown in Fig. 1.

3 System Design and Implementation

3.1 Architecture

The system is based on a layered software architecture. At the bottom is code for the lexical scanner. This is followed by the logical data types (terms, literals, clauses and formulas), with their associated input/output functions. Logical operations like unification and matching are implemented as separate modules, as are the generating inference rules and subsumption. On top of this, there are clause sets and formula sets, and the proof state of the given-clause algorithms, with two sets of clauses - one for those clauses that have been processed and one set that has not yet been processed.

From a logical perspective, the system is structured as a pipeline, starting with the parser, optionally followed by the clausifier and a module that adds equality axioms if equality is present, then followed by the core saturation algorithm, and finally, in the case of success, proof extraction and printing. To keep the learning curve simple, we have created 3 different provers: `pyres-simple` is a minimal system for clausal logic, `pyres-cnf` adds heuristics, indexing, and subsumption, and `pyres-fof` extends the pipeline to support full first-order logic with equality [11].

3.2 Implementation

Python is a high-level multi-paradigm programming language that combines both imperative and functional programming with an object-oriented inheritance system. It includes a variety of built-in data types, including lists, associative arrays/hashes and even sets. It shares dynamic typing/polymorphism, lambdas, and a built-in list datatype with LISP, one of the classical languages for symbolic AI and theorem proving. This enables us to implement both terms, the most frequent data type in a saturating prover, and atoms, as simple nested lists (s-expressions), using Python's built-in strings for function symbols, predicate symbols, and variables.

Literals are implemented as a class, with polarity, atom, and a flag to indicate literals selected for inference. Logical formulas are implemented as a class of recursive objects, with atoms as the base case and formulas being constructed with the usual operators and quantifiers. Top-level formulas are wrapped in a container object with meta-information. Both these formula containers and clauses are implemented as classes sharing a common super-class `Derivable` that provides for meta-information such as name and origin (read from input or derived via an inference record). The `Clause` class extends this with a list of literals, a TPTP style type, and an optional heuristic evaluation. The `WFormula` class extends it with a type and the recursive `Formula` object.

The `ClauseSet` class implements simple clause sets. In addition to methods for adding and removing clauses, it also has an interface to return potential inference partners for a literal, and to return a superset of possibly subsuming or subsumed clauses for a query clause. In the basic version, these simply return all clauses (clause/literal pairs for resolution) from the set. However, in the derived class `IndexedClauseSet`, simple indexing techniques (*top symbol hashing* for resolution and *predicate abstraction indexing*, a new technique for subsumption) return much smaller candidate sets. Resolution, factoring, and subsumption are implemented as plain functions.

The core of the provers is a given-clause saturation algorithm, based on two clause sets, the processed clauses and the unprocessed clauses. In the most basic case, clauses are processed first-in-first out. At each operation of the main loop, the oldest unprocessed clause is extracted from the unprocessed clauses. All its factors, and all resolvents between this *given clause* and all processed clauses, are computed and added to the unprocessed set. The clause itself is added to the processed set. The algorithm stops if the given clause is empty (i.e. an explicit contradiction), or if it runs out of unprocessed clauses. Fig. 2 shows the substantial methods of `SimpleProofState`. In contrast to most pseudo-code versions, this actually working code shows e.g. that clauses have to be made variable-disjoint (here by creating a copy with fresh variables).

The more powerful variant `pyres-cnf` adds literal selection, heuristic clause selection with multiple evaluations in the style of E [9], and subsumption to this loop. For clause selection, each clause is assigned a list of heuristic evaluations (e.g. symbol counting and abstract creation time), and the prover selects the next clause in a fixed scheme according to this evaluation (e.g. 5 out of 6 times, it picks the smallest clause, once it picks the oldest). Subsumption checks are performed between the given clause and the processed clauses. Forward subsumption checks if the given clause is subsumed by any processed clause. If so, it is discarded. Backward subsumption removes processed clauses that are subsumed by the given clause.

For the full first-order `pyres-fof`, we first parse the input into a formula set, and use a naive clausifier to convert it to clause normal form.

```
def processClause(self):
    """
    Pick a clause from unprocessed and process it. If the empty
    clause is found, return it. Otherwise return None.
    """
    given_clause = self.unprocessed.extractFirst()
    given_clause = given_clause.freshVarCopy()
    print("#", given_clause)
    if given_clause.isEmpty():
        # We have found an explicit contradiction
        return given_clause
    new = []
    factors     = computeAllFactors(given_clause)
    new.extend(factors)
    resolvents = computeAllResolvents(given_clause, self.processed)
    new.extend(resolvents)
    self.processed.addClause(given_clause)
    for c in new:
        self.unprocessed.addClause(c)
    return None

def saturate(self):
    """
    Main proof procedure. If the clause set is found
    unsatisfiable, return the empty clause as a witness. Otherwise
    return None.
    """
    while self.unprocessed:
        res = self.processClause()
        if res != None:
            return res
    else:
        return None
```

While most of the code should be self-explanatory, [] stands for the empty list, and extend() is a list method that adds the elements of another list at the end of a given list.

Fig. 2. Simple saturation

The code base has a total of 8553 lines (including comments, docstrings, and unit tests), or 3681 lines of effective code. For comparison, our prover E has about 377000 lines of code (about 53000 actual C statements), or 170000 when excluding the automatically generated strategy code.

3.3 Experiences

We would like to share some experiences about coding a theorem prover in Python. First, the high level of abstraction makes many tasks very straightforward to code. Python's high-level data types are a good match to the theory of automated theorem proving, and the combination of object-orientation with inheritance and polymorphism is particularly powerful.

Python also has good development tools. In particularly, the built-in unit-test framework (and the coverage tool) are very helpful in testing partial products and gaining confidence in the quality of the code. The Python profiler (cProfile) is easy to use and produces useful results. On the negative side, the lack of a strict type system and the ad-hoc creation of variables has sometimes caused confusion. In particular, when processing command line options, the relevant function sometimes has to set global variables. If these are not explicitly declared as `global`, a new local variable will be created, shadowing the global variable. Also, a misspelled name of a class- or structure member will silently create that member, not throw an error. As an example, we only found out after extensive testing that the prover never applied backward subsumption, not because of some logic error or algorithmic problem, but because we set the value of `backward_subsuption` (notice the missing letter "m") in the parameter set to `True` trying to enable it.

Overall, however, programming a prover in Python proved to be a lot easier and faster than in e.g. programming in C, and resulted in more compact and easier to read code. This does come at the price of performance, of course. It might be an interesting project to develop datatype and algorithm libraries akin to NumPy, TensorFlow, or scikit-learn for ATP application, to bring together the best of both worlds.

4 Experimental Evaluation

We have evaluated PyRes (in the `pyres-fof` incarnation) with different parameter settings on all clausal (CNF) and unsorted first-order (FOF) problems from TPTP 7.2.0. Table 1 summarizes the results. We have also included some data from E 2.4, a state-of-the-art high-performance prover, Prover9 [4] (release 1109a), and leanCoP 2.2. Prover9 has been used as a standard reference in the CASC competition for several years. LeanCoP is a very compact prover written in Prolog. Experiments were run on StarExec Miami, a spin-off of the original StarExec project [10]. The machines were equipped with 256GB of RAM and Intel Xeon CPUs running at 3.20GHz. The per-problem time-limit was set to 300s. For Prover9 and leanCoP, we used data included with the TPTP 7.2.0 distribution.

The *Best* configuration for PyRes enables forward and backward subsumption, negative literal selection (always select the largest literal by symbol count), uses indexing for subsumption and resolution, and processes given clauses interleaving smallest (by symbol count) and oldest clauses with a ratio of 5 to 1. The other configurations are modified from the *Best* configuration as described in the table. For the *Best* configuration (and the E results), we break the number of solutions into proofs and (counter-)saturations, for the other configurations we only include the total number of successes. All data (and the system and scripts used) is available at `http://www.eprover.eu/E-eu/PyRes1.2.html`.

It should be noted that Prover9, E, and leanCoP are all using an automatic mode to select different heuristics and strategies. leanCoP also uses strategy scheduling, i.e. it successively tries several different strategies

We present the results for different problem classes: UEQ (unit problems with equality), CNE (clausal problems without equality), CEQ (clausal problem with equality, but excluding UEQ), FNE (FOF problems without equality) and FEQ (FOF problems with equality).

Strategy	**UEQ**	**CNE**	**CEQ**	**FNE**	**FEQ**	**All**
Class size	(1193)	(2383)	(4442)	(1771)	(6305)	(16094)
Best (all)	116	1048	587	765	860	3376
Best (proofs)	113	946	499	632	725	2915
Best (sat)	3	102	88	133	135	461
No indexing	116	1042	567	736	829	3290
No subsumption	37	448	94	425	123	1127
Forward sub. only	115	1039	581	765	861	3361
Backward sub. only	40	541	106	479	143	1309
No literal selection	73	737	321	584	478	2193
E 2.4 auto (all)	813	1939	2648	1484	4054	10938
E 2.4 auto (proofs)	797	1621	2415	1171	3849	9853
E 2.4 auto (sat)	16	318	233	313	205	1085
Prover9-1109a (all)	728	1316	1678	709	2001	6432
LeanCoP 2.2 (all)	6	0	0	969	1826	2801

Table 1. PyRes performance (other systems for comparison)

A note on the UEQ results: Most of the problems are specified as unit problems in CNF. A small number are expressed in first-order format. While the original specifications are unit equality, the added equality axioms are non-unit. This explains the rather large decrease in the number of successes if negative literal selection is disabled.

Overall, in the *Best* configuration, PyRes solves 3376 of the 16094 problems. Disabling indexing increases run time by a factor of around 3.7 (for problems with the same search behaviour), but this translates to only about 90 lost successes. Disabling subsumption, on the other hand, reduces the number of solutions found by 2/3rd. However, if we compare the effect of forward and backward subsumption, we can see that forward subsumption is crucial, while backward subsumption plays a very minor role. If we look at the detailed data, there are about 10 times more clauses removed by forward subsumption than by backward subsumption. This reflects the fact that usually smaller clauses are processed first, and a syntactically bigger clause cannot subsume a syntactically smaller clause. Finally, looking at negative literal selection, we can see that this extremely simple feature increases the number of solutions by over 1100.

Comparing PyRes and E, we can see the difference between a rather naive resolution prover and a high-performance superposition prover. Maybe not unexpectedly, the advantage of the more modern calculus is amplified for problems with equality. Overall, PyRes can solve about 30% of the problems E can solve. But there is a clear partition into problems with equality (14% in UEQ, 22% in CEQ, 21% in FEQ) and problems without equality (54% in CNE, 52% in FNE). PyRes does relatively much better with the latter classes. Prover9 falls in between E and PyRes. LeanCoP, for the categories it can handle, is similar to Prover9, but like PyRes is relatively stronger on problems without equality, and relatively weaker on problems with equality.

5 Conclusion

We have described PyRes, a theorem prover developed as a pedagogical example to demonstrate saturation-based theorem proving in an accessible, readable, well-documented way. The system's complexity is orders of magnitude lower than that of high-performance provers, and first exposure to students has been very successful. We hope that the lower barrier of entry will enable more students to enter the field.

Despite its relative simplicity, PyRes demonstrates many of the same properties as high-performance provers. Indexing speeds the system up significantly, but only leads to a moderate increase in the number of problems solved. Simple calculus refinements like literal selection and subsumption (the most basic simplification technique) have much more impact, as have search heuristics.

It is tempting to extend the system to e.g. the superposition calculus. However, implementing term orderings and rewriting would probably at least double the code base, something that is in conflict with the idea of a small, easily understood system. We are, however, working on a Java version, to see if the techniques demonstrated in Python can be easily transferred to a new language by developers not intimately familiar with automated theorem proving.

References

1. Bachmair, L., Ganzinger, H.: Rewrite-Based Equational Theorem Proving with Selection and Simplification. Journal of Logic and Computation **3**(4), 217–247 (1994)
2. Korovin, K.: iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proc. of the 4th IJCAR, Sydney. LNAI, vol. 5195, pp. 292–298. Springer (2008)
3. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) Proc. of the 25th CAV, LNCS, vol. 8044, pp. 1–35. Springer (2013)

4. McCune, W.W.: Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/` (2005–2010), (acccessed 2016-03-29)
5. Otten, J.: leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proc. of the 4th IJCAR, Sydney. LNAI, vol. 5195, pp. 283–291. Springer (2008)
6. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM **12**(1), 23–41 (1965)
7. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications **15**(2/3), 111–126 (2002)
8. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. pp. 495–507. No. 11716 in LNAI, Springer (2019)
9. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Olivetti, N., Tiwari, A. (eds.) Proc. of the 8th IJCAR, Coimbra. LNAI, vol. 9706, pp. 330–345. Springer (2016)
10. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A Cross-Community Infrastructure for Logic Solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Proc. of the 7th IJCAR, Vienna. LNCS, vol. 8562, pp. 367–373. Springer (2014)
11. Sutcliffe, G., Schulz, S., Claessen, K., Gelder, A.V.: Using the TPTP Language for Writing Derivations and Finite Interpretations . In: Fuhrbach, U., Shankar, N. (eds.) Proc. of the 3rd IJCAR, Seattle. LNAI, vol. 4130, pp. 67–81. Springer (2006)
12. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R. (ed.) Proc. of the 22nd CADE, Montreal, Canada. LNAI, vol. 5663, pp. 140–145. Springer (2009)