Universidade Aberta

ProverX – Rewriting and Extending Prover9

Ivo Robert

Doctor's Degree in Computational Algebra

Doctoral dissertation supervised by:

Prof. Dr. João Araújo

Prof. Dr. Robert Veroff

2020

Resumo

O propósito principal deste projecto é tornar o demonstrador automático de teoremas Prover9 programável e, por conseguinte, extensível.

Este propósito foi conseguido acrescentando um interpretador de Python, uma linha de comandos e uma biblioteca de módulos, objectos e funções escritos em Python para interagir com ficheiros de Prover9 e Mace4. Foi também criada uma "interface" gráfica de utilizador (GUI) sob a forma de uma aplicação web para trazer aos utilizadores um meio mais eficiente e rápido de trabalhar com demonstrações automáticas de teoremas.

A nova biblioteca de "scripting" oferece aos utilizadores novas funcionalidades tais como correr várias sessões simultâneas de Prover9 parando automaticamente quando uma demonstração (ou um contraexemplo) é encontrada, elaborar estratégias para aumentar a velocidade com que as demonstrações são encontradas ou diminuir o tamanho das mesmas. Outro módulo permite interagir com o sistema de álgebra GAP.

Sobre esta biblioteca, muitas outras funcionalidades podem ser facilmente acrescentadas pois o objectivo principal é dar aos utilizadores a capacidade de acrescentar novas funcionalidades ao Prover9.

Resumindo, o objectivo deste projecto é oferecer à comunidade matemática um ambiente integrado para trabalhar com demonstração automática de teoremas.

Keywords: Prover9, Mace4, demonstração automática de teoremas, Python, GAP

Abstract

The primary purpose of this project is to extend Prover9 with a scripting language.

This was achieved by adding a Python interpreter, an interactive command line and a special scripting library to interact with Prover9 and Mace4 files. A user interface in the form of a web application was also created to help users achieve a more rapid and efficient way of working with automated theorem proving.

The new scripting library offers utilities that allows a user to run several Prover9 sessions concurrently and to create strategies for increasing the effectiveness of the proof search or to search for shorter proofs. Another module allows to interact with the algebra system GAP.

Based on the library, many more functionalities can be easily added, as the main goal is to give users the ability to extend the functionality of Prover9 the way they see fit.

In conclusion, the aim of this project is to offer to the mathematical community an integrated environment for working with automated reasoning.

Keywords: Prover9, Mace4, automated theorem proving, Python, GAP

Acknowledgements

I would like to express my gratitude to professors Robert Veroff and Michael Kinyon for all the feedback and support during the elaboration of this project, and specially professor João Araújo, not only for his wisdom and knowledge, but also for his never-ending optimism and moral support.

I also want to thank warmly my family for all their love and patience during this last two years.

Contents

Table of figures

Acronyms

API Application Programming Interface. A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.

DSL A domain-specific language is a computer language specialized to a particular application domain.

GAP Groups, Algorithms, Programming - a System for Computational Discrete Algebra, with particular emphasis on Computational Group Theory.

GUI Graphical User Interface. The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons instead of text-based user interfaces.

IDE Integrated Development Environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger.

REPL Read–Eval–Print Loop. Also termed an interactive toplevel or language shell, is a simple, interactive computer programming environment that takes single user inputs, evaluates them, and returns the result to the user.

VPS Virtual Private Server. A virtual private server is a virtual machine sold as a service by an Internet hosting service.

1 – Introduction

1.1 - A brief history of Automated Reasoning

In 1996 the New York Times [1] announced a milestone in automated theorem proving. The program EQP, developed by William McCune while he was part of the automated reasoning group at the Argonne National Laboratory had been used to solve a mathematical problem that had defeated famous mathematicians such as Tarski, Pigozzi, Robbins, McNulty, etc. for decades [2]. But this was the culmination of a long process that started long before.

1.1.1 – The beginning

The idea behind having a machine reasoning is a very old one [3]. Leibniz thought such a machine would be useful to settle all disagreements: it was only needed to find a universal language into which two different opinions could be translated and then a reasoning machine would decide who is right.

Many years after Leibniz's dream, George Boole [4] developed the first language to which formal rules could be applied leading to propositions whose logical value would be known, even when there is no interpretation for the starting "axioms". This first step was turned into a powerful tool with the introduction of symbols for quantifiers and the corresponding formal rules, shortly after leading to the concept of mechanical computability. The time between 1850 and 1950 was a time of great progresses in the area, coming from the work of a long list of mathematicians (DeMorgan, Russell, Whitehead, Peano, Frege, Skolem, Hilbert, Zermelo, Fraenkel, Herbrand, Godel, Church, Gentzen, Turing, etc.), finally providing a language as envisaged by Leibniz (even if far more modest than he conceived it) and the corresponding conceptual proving machine.

As Harrison describes on his 2007 paper, "A short survey of automated reasoning" [5] :

Boole [11] developed the first really successful symbolism for logical and set-theoretic reasoning. What's more, he was one of the first to emphasize the possibility of applying formal calculi to several different situations and doing calculations according to formal rules without regard to the underlying interpretation. In this way, he anticipated important parts of the modern axiomatic method. However, Boole's logic was limited to propositional reasoning (plugging primitive assertions together using such logical notions as 'and' and 'or'), and it was not until the much later development of quantifiers that formal logic was ready to be applied to general mathematics.

The introduction of formal symbols for quantifiers, in particular the universal quantifier 'for all' and the existential quantifier 'there exists', is usually credited independently to Frege, Peano and Peirce. Logic was further refined by Whitehead and Russell, who wrote out a detailed formal development of the foundations of mathematics from logical first principles in their Principia Mathematica [109]. In a short space of time, stimulated by Hilbert's foundational programme (of which more below), the usual logical language as used today had been developed.

English	*Symbolic*	*Other symbols*
false	$\bot$	*0, F*
true	$\top$	*1, T*
not p	$\neg p$	$\bar{p}$ *,* $-p$*,* $\sim p$
p and q	$p \wedge q$	*pq, p&q,* $p \cdot q$
p or q	$p \vee q$	$p + q$*,* $p \mid q$*, p or q*
p implies q	$p \Rightarrow q$	$p \rightarrow q$*,* $p \supset q$
p iff q	$p \Leftrightarrow q$	$p = q$*,* $p \equiv q$*,* $p \sim q$
for all x, p	$\forall x.\ p$	*(x)p*
there exists x such that p	$\exists x.\ p$	*(Ex)p*

At its simplest, one can regard this just as a convenient shorthand, augmenting the usual mathematical symbols with new ones for logical concepts. After all, it would seem odd nowadays to write 'the sum of a and b' instead of 'a + b', so why not write 'p ∧ q' in-stead of 'p and q'? However, the consequences of logical symbolism run much deeper: arriving at a precise formal syntax means that we

can bring deeper logical arguments within the purview of mechanical computation.

In 1956 Newell et al. wrote the first program (Logic Theorist) that could prove theorems of propositional logic, powerful enough to prove (sometimes with more elegant proofs) 38 of the 52 theorems of Part I Principia Mathematica [6].

The next natural step was a program to handle first order logic (FOL), but that process would turn out to be much more complex. In 1960 Paul Gilmore from IBM [7] produced the first such tool, introducing many ideas that are now standard such as the inclusion of the assumptions together with the negation of the goal, and having the program search for a contradiction. Even if some modern automated reasoning tools for FOL accept the undenied goal, internally they negate it and carry on with the search for a contradiction. This program, and many similar ones produced around the same time, were still too incipient to have any practical use. The quantum leap was Robinson's resolution method [8], appropriately published in the world's #1 computer science journal. The expectation was that a computational tool able to solve problems that defeated mathematicians would appear promptly. But again, that expectation proved to be overly optimistic. It took several more years of combined work by many experts to meaningfully tame the "explosion of the search space" and start extracting important mathematical results. These experts included several Turing Award recipients (John McCarthy (1971), Newell & Simon (1975), Hoare (1980), Milner (1991), Pnueli (1996), Clark, Emerson & Sifakis (2007)), and also an automated reasoning group gathered at the Argonne National Laboratory (for example, Larry Wos, SteveWinker, Ross Overbeek, Ewing Lusk, Brian Smith and Bill McCune) and external collaborators (for example, Bob Veroff, Kenneth Kunen, Michael Kinyon, R. Padmanabhan, Zac Ernst, etc).

Just as in the 60s the quantum leap was thanks to Alan Robinson, in the 90s it was thanks to McCune, thus managing to finally put computers to answer conjectures that have defeated mathematicians. His program EQP proved the Robbins Conjecture (posed by Herbert Robbins at Harvard University in the 1930s), a problem important enough to attract the attention of Tarski [9], McNulty [10], Pigozzi [11], Taylor [12], in addition to Robbins himself and many others.

The Robbins Conjecture and the story behind its "cracking" by a computer program is well covered by Wos, Veroff and Pieper in "Logical Basis for the Automation of Reasoning: Case Studies" [13]:

Robbins algebra

The following basis (expressed here in clause form) was presented for Boolean algebra by E. V. Huntington in the early 1930s.

(H1)	*x + y = y + x*	*% commutativity*
(H2)	*(x + y) + z = x + (y + z)*	*% associativity*
(H3)	*n(n(x) + y) + n(n(x) + n(y)) = x*	*% Huntington equation.*

Shortly thereafter, H. Robbins conjectured that H3 could be replaced by the following clause:

(R3) n(n(x + y) + n(x + n(y))) = x % Robbins equation

Algebras satisfying H1, H2, and R3 are called Robbins algebras. Since every Boolean algebra is a Robbins algebra, the question arose: Is every Robbins algebra Boolean?

The question intrigued some of the best minds, including the famed logician A. Tarski and his students [Henkin et al., 1971; Tarski, private communication 1980] to no avail. Subsequently, S. Winker attacked the problem using a combination of individual insight and an automated reasoning program. He was able to prove that certain conditions such to make a Robbins algebra Boolean [Winker, 1990, 1992]. But, with the reasoning programs available, he was unable to prove that any of these conditions follow from the axioms.

Then, in 1996, with the development of a new theorem prover called EQP [McCune, 1997b], the problem was cracked. The 133-step solution, which relied

> *on a technique known as associative-commutative unification [Stickel, 1981], required 20 hours using 18 megabytes on a Unix workstation. The answer:r all Robbins algebras are Boolean took the world by storm [McCune, 1997a]. It was covered by the New York Times [Kolata, 1996] and was cited as one of the major accomplishments in artificial intelligence by the NEC Research Institute and Computing Research Association [Waltz, 1997].*

It is worth observing that although it is one of the best known, the Robbins Conjecture was not the first mathematical problem solved by a computer. For example, more than 10 years before, Ewing Lusk started to work with Robert McFadden and John Howie to study some problems in semigroup theory [14]; however, the solution of the Robbins' problem was the first solution of an important open and old problem. To show the power of this new tool, R. Padmanabhan and McCune [15] wrote a book on cubic curves in which the human authors had to devise what would be interesting to prove or where the research should go, while the proofs were entirely left to the machine [15]. This is as if Padmanabhan and McCune just raised questions, leaving to a group of super-mathematicians to sort out the proofs.

A sad historical note: Even if Argonne National Laboratory was pivotal in the process that led to this long-awaited success, about ten years ago it was decided that Argonne should stick to its core business (Energy) and get rid of everything else. Therefore, the automated reasoning group was dismantled, with some members (such as Larry Wos) retiring or in some other way leaving the topic or becoming less active overall. As for Bill McCune, he took early retirement and moved to the University of New Mexico (where he joined his long-time collaborator Bob Veroff, one of Wos' former students), but died shortly after.

1.1.2 - The present: current actors on the field

First, we must clarify that theorem provers are not only about proving mathematical theorems. As remarked by Voronkov on his 2003 talk at the 18th International Joint Conferences on Artificial Intelligence Organization [16]:

> *The main application area of theorem provers has been, and continues to be, verification of software and hardware. Full applications of this kind usually cannot be directly represented in the first-order form, so provers are normally used to prove sub-goals generated by other systems, for example VHDL-to-first-order transformation systems or proof assistants based on higher-order logic or type theories. There are too many papers on this subject to be mentioned here. Finite-state model checkers and interactive proof assistants are currently prevailing in verification, but with the growing complexity of hardware first-order logic and its extensions are likely to play an increasingly important role.*

> *Theorem proving in mathematics has been the first application area for theorem provers. Provers are not very good at working in structured mathematical theories, but they are very efficient in fields of mathematics where combinatorial reasoning is required, for example, in algebra.*

Before talking about some of the leading projects in the field of AR it is important to differentiate between proof assistants and theorem provers:

A **proof assistant** or interactive theorem prover is a program that assists a human in the development of formal proofs in a collaborative way. This is usually accomplished with the aid of some sort of interactive proof editor, with which a human can guide the search for proofs.

On the other hand, a **theorem prover** is an independent system in which some goals (theorems to prove) are given alongside some axioms in a formal language. It is expected that the software will work autonomously to find a proof to (or refute) the theorem.

In turn, theorem provers are sometimes combined with counter-proof finders which can run in parallel to find a counterexample that refutes the theorem. This is done with a special software called a model finder with the task of finding finite models that fit some theory. Then, if this model finder finds some counter model, the theorem is disproved.

Another class of tools worth mentioning is **proof verifiers**. These are program that check the correctness of proofs provided by theorem provers. These kind of software tools has become more and more relevant as the proofs produced by most theorem provers are very hard to read by humans and can sometimes be hundreds (and even thousands) of pages long!

We will now, survey some of the most well-known projects on the field of proof assistants and theorem provers:

Proof Assistants:

We will just focus on two of the most well-known proof assistants: Coq and Isabelle:

Coq[1]

Coq implements a program specification and mathematical higher-level language called *Gallina* that is based on an expressive formal language called the *Calculus of Inductive Constructions* that itself combines both a higher-order logic and a richly typed functional programming language. Through a *vernacular* language of commands, Coq allows:

- to define functions or predicates, that can be evaluated efficiently;
- to state mathematical theorems and software specifications;
- to interactively develop formal proofs of these theorems;
- to machine-check these proofs by a relatively small certification "kernel";
- to extract certified programs to languages like Objective Caml, Haskell or Scheme.

[1] https://coq.inria.fr/

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a *tactic* language for letting the user define its own proof methods. Connections with external computer algebra system or theorem provers is available.

As a platform for the formalization of mathematics or the development of programs, Coq provides support for high-level notations, implicit contents and various other useful kinds of macros.

Coq comes with libraries for efficient arithmetics in N, Z and Q, libraries about lists, finite sets and finite maps, libraries on abstract sets, relations, classical analysis, etc.

Coq is released with:

- a graphical user interface based on gtk (CoqIDE) (see the chapter of the reference manual about CoqIDE),
- documentation tools (coqdoc and coq-tex) and a statistics tool (coqwc),
- dependency and makefile generation tools for Coq (coq_makefile and coqdep),
- a stand-alone proof verifier (coqchk).

Isabelle [2]

Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages.

The most widespread instance of Isabelle nowadays is Isabelle/HOL, which provides a higher-order logic theorem proving environment that is ready to use for big applications.

[2] https://isabelle.in.tum.de/index.html

Isabelle/HOL includes powerful specification tools, e.g. for (co)datatypes, (co)inductive definitions and recursive functions with complex pattern matching.

Proofs are conducted in the structured proof language Isar, allowing for proof text naturally understandable by both humans and computers.

For proofs, Isabelle incorporates some tools to improve the user's productivity. In particular, Isabelle's classical reasoner can perform long chains of reasoning steps to prove formulas. The simplifier can reason with and about equations. Linear arithmetic facts are proved automatically, and various algebraic decision procedures are provided. External first-order provers can be invoked through sledgehammer.

Abstract specifications are supported by a module system (known as locales), of which type classes are a special case.

Isabelle provides excellent notational support: new notations can be introduced using normal mathematical symbols. Definitions and proofs may include LaTeX source, from which Isabelle can automatically generate typeset documents (papers, books, theses).

Isabelle/HOL allows to turn executable specifications directly into code in SML, OCaml, Haskell, and Scala.

Isabelle comes with a large theory library of formally verified mathematics, including elementary number theory (for example, Gauss's law of quadratic reciprocity), analysis (basic properties of limits, derivatives and integrals), algebra (up to Sylow's theorem) and set theory (the relative consistency of the Axiom of Choice). Also provided are numerous examples arising from research into formal verification. A vast collection of applications is accessible via the Archive of Formal Proofs, stemming both from mathematics and software engineering.

Isabelle/jEdit is the default user interface and Prover IDE for Isabelle. It is based on jEdit and Isabelle/Scala. It provides a metaphor of continuous proof checking of a versioned collection of

theory sources, with instantaneous feedback in real-time and rich semantic markup for the formal text.

Isabelle may serve as a generic framework for rapid prototyping of deductive systems. These are formulated within Isabelle's logical framework Isabelle/Pure, which is suitable for a variety of formal calculi (e.g. axiomatic set theory). Instantiating the generic infrastructure to a particular calculus usually requires only minimal setup in the Isabelle implementation language ML. One may also write arbitrary proof procedures or even theory extension packages in ML, without breaking system soundness (Isabelle follows the well-known LCF system approach to achieve a secure system).

Theorem provers:

E [3]

E is a theorem prover for full first-order logic with equality. It accepts a problem specification, typically consisting of a number of first-order clauses or formulas, and a conjecture, again either in clausal or full first-order form. The system will then try to find a formal proof for the conjecture, assuming the axioms.

If a proof is found, the system can provide a detailed list of proof steps that can be individually verified. If the conjecture is existential (i.e. it's of the form "there exists an X with property P"), the latest versions can also provide possible answers (values for X).

Development of E started as part of the E-SETHEO project at TUM. The first public release was in in 1998, and the system has been continuously improved ever since. I believe that E now is one of the most powerful and friendly reasoning systems for first-order logic. The prover has successfully participated in many competitions.

[3] https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html

Prover9 / Mace 4 [4]

Prover9 is an automated theorem prover for first order and equational logic developed by William McCune. Prover9 is paired with Mace4, which searches for finite models and counterexamples. Prover9, Mace4, and many other tools are built on an underlying library named LADR to simplify implementation. Since they share the same input language, Prover9 and Mace4 can be executed simultaneously. Prover9 is the successor of the Otter prover[5]. In July 2006 the LADR/Prover9/Mace4 input language made a major change (which also differentiates it from Otter). The key distinction between "clauses" and "formulas" completely disappeared; "formulas" can now have free variables; and "clauses" are now a subset of "formulas". Prover9/Mace4 also supports a "goal" type of formula, which is automatically negated for proof by contradiction. Prover9 attempts to automatically generate a proof by default; in contrast, Otter's automatic mode must be explicitly set.

Mizar [6]

Mizar is more than just a theorem prover.

The Mizar system consists of a formal language for writing mathematical definitions and proofs, a proof assistant, which is able to mechanically check proofs written in this language, and a library of formalized mathematics, which can be used in the proof of new theorems.[1] The system is maintained and developed by the Mizar Project, formerly under the direction of its founder Andrzej Trybulec.

In 2009 the Mizar Mathematical Library was the largest coherent body of strictly formalized mathematics in existence.

Vampire [7]

[4] ://www.cs.unm.edu/~mccune/prover9/
[5] https://www.cs.unm.edu/~mccune/otter/
[6] http://mizar.uwb.edu.pl/

Vampire is an automatic theorem prover for first-order classical logic developed in the School of Computer Science at the University of Manchester by Andrei Voronkov together with Kryštof Hoder and previously with Alexandre Riazanov. So far it has won the "world cup for theorem provers" (the CADE ATP System Competition) in the most prestigious CNF (MIX) division eleven times (1999, 2001–2010).

Vampire's kernel implements the calculi of ordered binary resolution and superposition for handling equality. The splitting rule and negative equality splitting can be simulated by the introduction of new predicate definitions and dynamic folding of such definitions. A DPLL-style algorithm splitting is also supported. A number of standard redundancy criteria and simplification techniques are used for pruning the search space: tautology deletion, subsumption resolution, rewriting by ordered unit equalities, basicness restrictions and irreducibility of substitution terms. The reduction ordering used is the standard Knuth–Bendix ordering.

A number of efficient indexing techniques are used to implement all major operations on sets of terms and clauses. Run-time algorithm specialization is used to accelerate forward matching.

Although the kernel of the system works only with clausal normal forms, the preprocessor component accepts a problem in the full first-order logic syntax, clausifies it and performs a number of useful transformations before passing the result to the kernel. When a theorem is proven, the system produces a verifiable proof, which validates both the clausification phase and the refutation of the conjunctive normal form.

Along with proving theorems, Vampire has other related functionalities such as generating interpolants.

[7] http://www.vprover.org/

Waldmeister [8]

Waldmeister is a theorem prover for first order unit equational logic. It is based on unfailing Knuth-Bendix completion [17] employed as proof procedure. Waldmeister's main advantage is that efficiency has been reached in terms of time as well as of space.

In outline, the task Waldmeister deals with is the following: A theory is formulated as a set E of implicitly universally quantified equations over a many-sorted signature. It shall be demonstrated that a given equation s=t is valid in this equational theory, i.e. that it holds in all models of E. Equivalently, s is deducible from t by applications of the axioms of E, substituting equals for equals. In 1970, Knuth and Bendix presented a completion algorithm, which later was extended to unfailing completion, as described e.g. by Bachmair et al. Parameterized with a reduction ordering, the unfailing variant transforms E into a ground convergent set of rewrite rules. For theoretical reasons, this set is not necessarily finite, but if so, the word problem of E is solved by testing for syntactical identity after normalization. In both cases, however, if s=t holds, then a proof is always found in finite time. This justifies the use of unfailing completion as areduction semi-complete proof procedure for equational logic.

Accordingly, when searching for a proof, Waldmeister saturates the given axiomatization until the goals can be shown by narrowing or rewriting. The saturation is performed in a cycle working on a set of waiting facts (critical pairs) and a set of selected facts (rules). Inside the completion loop, the following steps are performed:

1. Select an equation from the set of critical pairs.
2. Simplify this equation to a normal form. Discard if trivial, otherwise orient if possible.
3. Modify the set of rules according to the equation.
4. Generate all new critical pairs.
5. Add the equation to the set of rules.

[8] "Waldmeister," [Online]. Available: https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/waldmeister/

The selection is controlled by a top-level heuristic maintaining a priority queue on the critical pairs. This top-level heuristic is one of the two most important control parameters. The other one is the reduction ordering to orient rules. There is some evidence that the latter is of even stronger influence.

It is also worth mentioning the CADE ATP System Competition (CASC)[9], a yearly competition of first-order systems for many important classes of first-order problems.

The CASC Competition, with all its merits, worked as an attractor to a special type of system, namely those that were specifically tooled to win the competition (mainly focused on speed), therefore diverting the developments from what would be more useful to the working mathematician: small learning curve, nice input interface, built-in techniques and algorithms to perform the usual tasks of mathematicians (generalization, analogy, knowledge of the literature, etc.)

This was one of the things that most used to bother the late Bill McCune regarding the direction automated reasoning was taking.

1.1.3 - What about the future?

Predicting the future is a hard, if not impossible, task, but one can make educated assumptions based on the present. Let's see what some of the lead thinkers in this field have to say about this matter:

We'll start by quoting Voronkov [16]:

> ***Future Generation Theorem Provers***
>
> *Theorem proving is a very hard problem. The next generation of theorem provers will incorporate new theory, data structures, algorithms, and implementation*

[9] http://tptp.cs.miami.edu/~tptp/CASC/

techniques. Their development will be driven by the quest for flexibility and efficiency. Flexibility is required to adapt provers to new applications. Efficiency can be reformulated as controlling redundancy in large search spaces [Lusk, 1992].

It is unreasonable to expect future theorem provers to be much faster on all possible problems. However, if we can increase performance of provers by several orders of magnitude for a large number of problems coming from applications, many of these problems will be routinely solved, thus saving time for application developers. The development of next generation provers will require:

1. development of new theory,

2. addition of new features;

3. development of new algorithms and data structures;

4. understanding how the theory developed so far can be efficiently implemented on top of the existing architectures of theorem provers;

This development is impossible without considerable implementation efforts and extensive experiments.

Wos, Veroff and Pieper [13] have a similar prevision for the future:

We conjecture that the most significant contributions to automated reasoning will be in the area of strategy specifically, strategy for restricting the actions of a program. For example, the set of support strategy restricts the application of an inference rule by preventing it from being applied to various subsets of clauses. That strategy, still considered one of the most powerful strategies yet formulated, has the added advantage of requiring no CPU time to exercise. Nevertheless, the effect of the set of support strategy is limited. For a specific research problem to solve, one might study the possibility of extending or generalizing the set of

support strategy. For a second specific research problem to solve, one might study possible strategies to control the actions of paramodulation; uncontrolled, the inference rule produces far too many conclusions.

Regardless of the choice of problem, aspect, or area almost without exception what is required is experimentation. A sharp increase in experimentation was in fact one of the major forces for the rapid advance of automated reasoning. As evident from the material in the next section, the conditions for easy and substantial experimentation are far better than at any other time in the brief history of the field.

As a computer scientist, the author tends to agree with the notion that the future development of automatic reasoning should be focused on finding good strategies. But one must be careful not to incur in wishful thinking. It is, of course, very tempting and seductive to someone who has spent his all life working in algorithms to imagine a future where "intelligent" programs will be superior to brute force attacks.

Nowadays, with the abundance of resources (cheap and fast processors and vast amount of available memory) other fields of artificial intelligence have a tendency to show surprisingly good results using brute force approaches like Deep Blue, the ubiquitous Neural Networks (that now even run on smartphones for face and voice recognition, etc.).

On the other hand, we have seen recently some drastic speed improvements accomplished by Andrei Voronkov with AVATAR [18] (Advanced Vampire Architecture for Theories and Resolution) by coupling a SAT or a SMT solver to a theorem prover (VAMPIRE). This seems to prove his former prediction that the future of theorem proving lays on the field of strategies.

Probably the future will see a blend of the two (brute force and strategy) and maybe Irving Kaplansky prediction will become true (Kaplansy is credited with the prediction that automated

reasoning (AR) would enable 21st century mathematicians to tackle problems so sophisticated, deep and complex that a 20th century mathematician would likely find the work incomprehensible).

1.2 - A first look at ProverX

In a nutshell, ProverX can be summarized as a scriptable automated theorem prover with an IDE that runs in the cloud.

The first contact with ProverX (by pointing any modern browser to http://www.proverx.com and entering as guest) shows a classical IDE with a file explorer on the left, several tabbed editors for coding, and a toolbar on top. This will be a familiar environment for anyone used to modern IDE's.

One can immediately enter axioms and goals using Prover9 syntax and obtain a proof (or counter example). From this point of view, ProverX can be seen as web IDE to Prover9. But in fact, ProverX is much more than a simple IDE. Its major strength comes from the fact that it is programmable (scriptable) making it possible (and easy) to execute and automate task that were done by hand before. For example, running several instances of provers at once, interchanging results between them, interacting with axioms and models, etc.

Ironically, the motivations behind the ProverX project follows the exactly reverse order of this first glance. The primary concern that originated this project was to add a scripting language to Prover9. Only after a few experimentations, the need for an IDE emerged. It seemed then obvious to make this a cloud IDE instead of a desktop one. This choice was not dictated by an actual trend but because the author of this project is invested in aiding to teach automated reasoning and avoiding users the burden to install new software (especially if it's a UNIX-style command line) can be a huge benefit (for example, in prover9 master classes, more than half an hour is usually lost in helping attendees to install the software on their laptops; by contrasting this with pointing a browser

to a simple address and having everybody on the same page in minutes, one can immediately see the benefits of a cloud IDE).

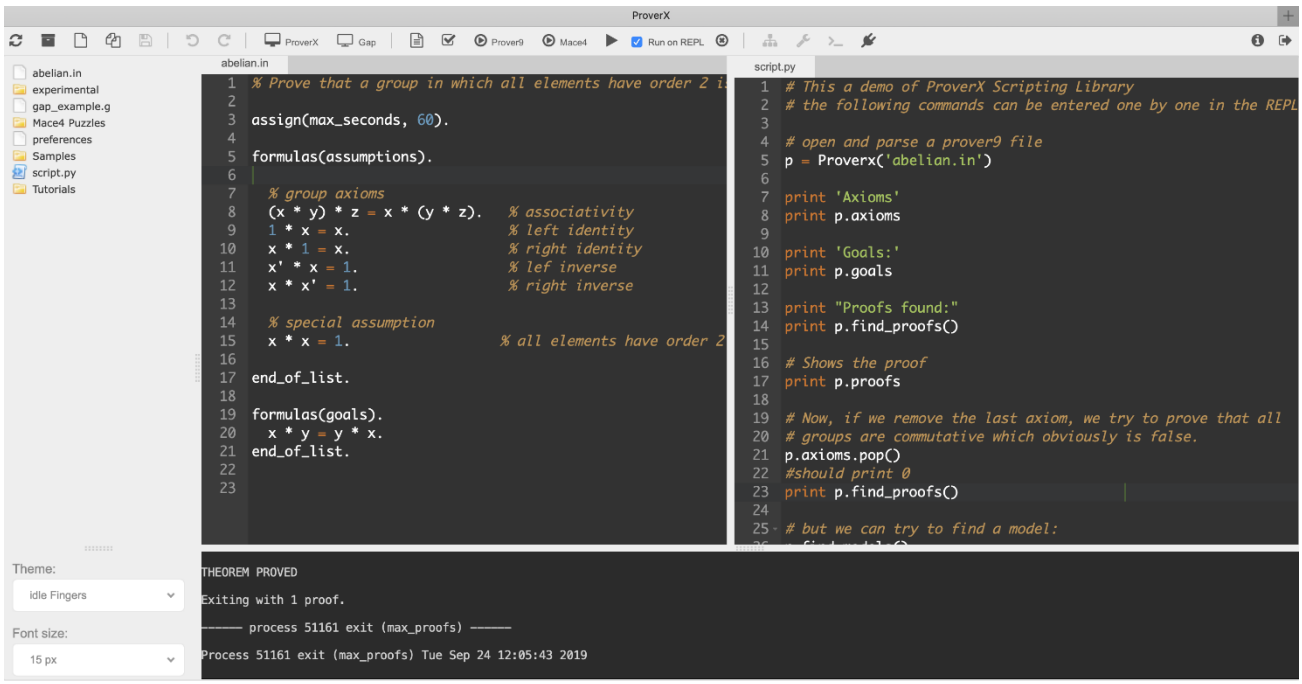


Figure 1- ProverX main screen view

1.3 - Objectives and motivation

1.3.1 – Primary goal: add a scripting language to Prover9

As already stated, the main goal of the proverX project is to take a theorem prover core and wrap it with a scripting language.

As we have seen in section 1.1.3, the future of theorem provers seems to rely on implementing good strategies and (perhaps even more important) experimentation.

It is with this goal in mind that we decided to add a scripting language to Prover9.

In fact, as he wrote in a private email in 2010 to João Araújo, Bob Veroff and Michael Kinyon, this was a project that William McCune had in mind (although the author of this project was not aware of this when he started this project; he was told later that Dr. McCune had even chose the same scripting language as he did: Python).

It is important to note that some of the algorithms that were implemented *ad hoc* in ProverX, have been used by W. McCune, J. Araújo, M. Kinyon, J. Konieczny and A. Malheiro to prove some important theorems [19] [20] [21] [22] [23].

As a side note, ProverX has been successfully used for teaching mathematics to more than 200 students in an academic environment (the chair of Matemática Discreta in FCT/UNL second semester of 2018/2019)

1.3.3.1 - Why a scripting language?

One of the first motivations for this project was to create something similar for theorem proving as GAP[10] is for algebraic computation. Following the GAP model, a command line was added. But a command line needs a language. Instead of creating a new language from scratch, it seemed more efficient to use an existing one. This decision both saves time and reduces the learning curve (assuming the user is already familiar with the chosen language). We chose Python for reasons explained on Section 2.1 (Design decisions).

Having a programming language brings many benefits:

- It allows users to tailor the system to fit their needs
- It allows users to expand the system with new functionality.

[10] http://www.gap-system.org/index.html

- It allows users to share extensions thus creating an ecosystem around the project.

But most importantly it makes experimenting easier (as we'll cover in the tutorial chapter, especially on **4.2 – Using strategies**). We can see this tendency of using scripting languages in product development in the gaming industry. Lua[11] is heavily used because it has a small footprint and is easy to couple with C/C++ gaming engines. And since it's a very simple and easy language to learn, it allows non-technical developers (storytellers and artists) to experiment with game development without touching the core engine. We hope that the same thing will happen with ProverX.

But this kind of architecture (a core with an open scripting language) is not limited to game programming. This is a very pervasive model that can be found in all areas of software development. Examples range from music creation system to CAD, photo editing, electronic simulation, spreadsheets, databases, and, of course, mathematical systems.

All these systems share (with some minor variations) the same architecture:

a) A kernel

This is the system core. The kernel is usually formed by the fundamental algorithms that must be executed as fast as possible. As so, the kernel is often written in a low-level language (most commonly C). The kernel is also constituted by all complex algorithms that encapsulate the area of expertise needed for that particular field. These are called the low-level routines (in opposition to the high-level routines written with the scripting language).

b) A scripting language coupled with an extensible library

In many cases the scripting engine (a language interpreter) is created from scratch but it is becoming more and more common to use "off-the-shelf"embeddable languages (most popular languages are Lua, Python or Javascript, but Lisp languages like Scheme are also often found) .

[11] https://www.lua.org/

The scripting language is interpreted which means that it has usually a more flexible syntax and is easy to learn. There is a downside: what we gain in speed of development, we lose in speed of execution.

Most systems include a set of pre-written routines in the form of one or more libraries. This approach brings a huge benefit to the developers who can program complex algorithms with the agility that an interpreted language brings, leaving for the kernel only what needs to run as fast as possible. As for the users, they have a huge amount of functionality ready at their fingertips, needing only to write the routines to solve their specific problems.

These libraries can be part of the system or provided by third party developers. When they are external, they are often called extensions, plug-ins or packages. It is also important to note that, in many systems, the packages can be a mix of external programs (written in C/C++ for speed) interfaced with the scripting language.

One of the most delicate (and time consuming) part of implementing such a system is to create a module that acts as an interface between the kernel and the scripting engine. This module is usually (but not always, as it can be external) embedded in the kernel.

c) A command line (or REPL)

One of the benefits of these kind of systems is that they allow quick experimentations. This is achieved by including a command line to enter commands and get immediate answers. The syntax of the commands is usually the same as the scripting language.

1.3.1.2 - The kernel: why Prover9?

This is a very important question. Why chose Prover9 as the core of this system? The first and obvious reason, is that the author of the project is very familiar with it. But this is not the only reason.

Prover9 has a very clean and intuitive syntax for declaring clauses and goals, making it a very popular theorem prover used by mathematicians worldwide. Again, the author is fully aware that this statement is probably biased, a familiar syntax will always seem more intuitive and clean than an unfamiliar one.

One other important reason is the availability of prover9 source code. More important is that the source code is written in the C language which makes adding a scripting engine easier since, usually, the instructions and examples to embed most scripting languages are also written in C.

Another nice feature of Prover9 is that it automatically negates goals.

One downside is that the proofs it produces are not easy to read by humans (although translators have been written for this purpose). Also, resulting proofs can be double-checked by Ivy[12], a proof-checking tool that has been separately verified using ACL2[13].

The primary purpose of ProverX project is to extend Prover9 with a scripting language, hence giving the users the ability and freedom to extend the functionality of Prover9 the way they see fit.

This is achieved by adding a Python[14] interpreter, a REPL (read, eval, print, loop) and a special scripting library to work with Prover9 and Mace4 files.

Some extensions to Prover9 syntax were also added like exponentiation, for loops and defining functions by induction. A new useful feature is the “include” keyword for including files (like most programming languages) and opens the possibility to create libraries of axioms (see the section: “New directives” in chapter 4).

[12] https://www.cs.unm.edu/~mccune/papers/ivy/
[13] http://www.cs.utexas.edu/users/moore/acl2/
[14] https://www.python.org/

1.3.3.3 - Domain specific language

To better understand what benefits a scripting language can bring to a project like ProverX it is important to talk about domain specific languages.

A domain-specific language (DSL) is a computer language specialized to a particular application domain. Martin Fowler [24] divides DSL's in two categories, internal or external:

- *An external DSL is a language separate from the main language of the application it works with. Usually, an external DSL has a custom syntax, but using another language's syntax is also common (XML is a frequent choice). A script in an external DSL will usually be parsed by a code in the host application using text parsing techniques. The Unix tradition of little languages fits this style. Examples of external DSLs that you probably have come across include regular expressions, SQL, Awk, and XML configuration files for systems like Struts and Hibernate.*

- *An internal DSL is a particular way of using a general-purpose language. A script in an internal DSL is valid code in its general-purpose language, but only uses a subset of the language's features in a particular style to handle one small aspect of the overall system. The result should have the feel of a custom language, rather than its host language. The classic example of this style is Lisp; Lisp programmers often talk about Lisp programming as creating and using DSLs. Ruby has also developed a strong DSL culture: Many Ruby libraries come in the style of DSLs. In particular, Ruby's most famous framework, Rails, is often seen as a collection of DSLs.*

From this definition, we can see that, in a way, ProverX has both an external and an internal DSL. By this we mean that Python can be seen as the external DSL whereas the ProverX Scripting Library is more akin an internal DSL.

We emphasize this distinction because we feel that it is of utmost importance that the ProverX Scripting Library (in our opinion the most important part this project) can mimic and use the same terms as a mathematician using prover9 “by hand” would.

Although is not yet perfect and is still a work in progress we thrive to make the scripting library so intuitive to use that in the end a user might forget that it is using Python.

A small example might be in order to clarify this point.

Before presenting this example, we introduce the notion of hints in Prover9. As stated on the Prover9/Mace4 website [15]:

> *Hints frequently consist of proofs, perhaps many, of related theorems.*
>
> *Bob Veroff developed the concept, installing code for hints in an early version of Otter, to experiment with his method of proof sketches [Veroff-hints, Veroff-sketches]. In the proof sketches method, a difficult conjecture is attacked by first proving several (or many) weakened variants of the conjecture, and using those proofs as hints to guide searches for a proof of the original conjecture.*

In our example, and following the example on the website, let’s suppose we have file called “hard.in” with a non-trivial theorem to prove and another file called “easy.in” with a weakened variant.

So, what is needed is to run Prover9 on the easy file, read the proofs form the result, parse the text file to extract the proofs, transform the proofs into hints, add the hints to the hard file and finally run Prover9 again on this file.

To accomplish this task in pure Python would take several hundred lines of code, but using ProverX we only need five lines:

[15] https://www.cs.unm.edu/~mccune/prover9/manual/2009-02A/hints.html

```
easy = Proverx('Tutorials/easy.in')
hard = Proverx('Tutorials/hard.in')
easy.find_proofs()
hard.hints.add(easy.proofs.hints)
hard.find_proofs()
```

In addition of the reduced amount of code needed, the code is easy to read even for a non-programmer (note that the code can almost be read as English).

As a matter of fact, William McCune did write a utility called *prooftrans* that does just that: extract the hint from a proof making it trivial to feed it to prover9. It is important to note that the fact that this utility exists doesn't make the above example irrelevant. On the contrary, it just underlines the need for a scripting language as Prover9 comes only with a few utilities which are not trivial to write and there is a growing need for many more of these utilities (that, for now on, will be written more easily using ProverX Scripting Library).

1.3.3.4 - Similar projects

We will now examine two famous mathematical software tools that adhere to this architecture, GAP a System for computational Discrete Algebra and the R Project for Statistical Computing[16]:

GAP

GAP is a system for computational discrete algebra, with particular emphasis on Computational Group Theory. GAP provides a programming language, a library of thousands of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, combinatorial structures, and more. GAP and its sources, including packages (sets of user contributed

[16] https://www.r-project.org/

programs), data library (including a list of small groups) and the manual, are distributed freely, subject to "copyleft" conditions. GAP runs on any Linux system, under Windows, and on Macintosh systems. The user contributed packages are an important feature of the system, adding a great deal of functionality. GAP offers package authors the opportunity to submit these packages for a process of peer review, hopefully improving the quality of the final packages, and providing recognition akin to an academic publication for their authors.

R

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of Linux platforms, Windows and MacOS.

R is an implementation of the S programming language combined with lexical scoping semantics, inspired by Scheme.

R and its libraries implement a wide variety of statistical and graphical techniques, including linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, and others. R is easily extensible through functions and extensions, and the R community is noted for its active contributions in terms of packages. Many of R's standard functions are written in R itself, which makes it easy for users to follow the algorithmic choices made. For computationally intensive tasks, C, C++, and Fortran code can be linked and called at run time. Advanced users can write C, C++, Java, .NET or Python code to manipulate R objects directly. R is highly extensible through the use of user-submitted packages for specific functions or specific areas of study. Due to its S heritage, R has stronger object-oriented programming facilities than most statistical computing languages. Extending R is also eased by its lexical scoping rules.

Another strength of R is static graphics, which can produce publication-quality graphs, including mathematical symbols. Dynamic and interactive graphics are available through additional packages.

The source code for the R software environment is written primarily in C, Fortran and R itself.

Although R has a command line interface, there are several graphical user interfaces, such as RStudio, an integrated development environment.

As we can see by the examples above, the strength of these kind of systems rely heavily on their ability to be extended, creating a strong community of users and extension developers.

To conclude: the main goal of ProverX is to bring to automated deduction what GAP and R bring respectively to symbolic algebra and statistics.

Before concluding this section, we would like to address a project that although is not similar to ProverX in the sense of having a kernel wrapped with a scripting language, is often referred or compared to ProverX:

SAGE

Sagemath[17], previously SAGE, is a free open-source mathematics software system built on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. It can access their combined power through a common, Python-based language or directly via interfaces or wrappers.

Since the ProverX project started, a question often heard is "why is ProverX better than SAGE?".

There is no real answer to this question since they are two different things and so, are not comparable.

The Sagemath goal is to integrate several mathematical systems under one unified interface, while ProverX aims to be an autonomous theorem deduction system (although it can be interfaced with the external world, as, for example, with its built-in GAP interface).

[17] http://www.sagemath.org/

In a nutshell, in the future, some parts of ProverX could be embedded in SAGE (since most of ProverX is written in Python this should be trivial).

To conclude this false debate between ProverX and SAGE, it is important to note that is becoming more and more common (and easy) to extend software and to implement interfaces to integrate different system. It is then, only normal, that software systems are getting bigger and bigger and that their functionalities tend to overlap each other.

1.3.2 - Secondary goal: and IDE on the cloud

Prover9 is a command-line program meaning that it is invoked from a terminal window receiving as input a text file containing the theorem to prove and it prints the result to the screen.

Since ProverX shares the same source code as Prover9 (with the addition of the Python engine), at its core it is also a command-line application. It usually runs in REPL mode, but it can be called exactly like Prover9 (for legacy reasons) or with a python script.

To clarify here are the options displayed by calling Proverx -h:

```
Usage: proverx [-proverx file | -macex file | -prover9 | -mace4] args | -script  [-i] file]

Options:
-proverx or -px : runs proverx preprocessor on file then calls prover9 with args
-macex or -mx : runs proverx preprocessor on file then calls mace4 with args
-prover9 or -p9 : runs like prover9 with usual arguments (ex: -f file)
-mace4   or -m4 : runs like mace4 with usual arguments
-script  or -s  : run python script in file (-i enters interactive mode)
```

```
If no option is given, proverx enters in interactive mode (REPL).
```

Before going any further, it is important to pinpoint that ProverX was developed with two different users in mind: researchers and students. We believe that both class of users would greatly benefit from having an IDE.

As a matter of fact, an IDE for Prover9 already exists [18]. It runs on MS-Windows, MacOS and Linux, but unfortunately it has not been updated from several years and it doesn't run on the last windows10 versions.

Although it is true that some of Prover9 "power users" use it on the command line writing the theorems on a text editor, most programmers find that an IDE makes their workflow more expedite.

The truth is that from the moment the kernel was completed, and we started working on the ProverX Scripting Library, we felt the need to have a decent IDE to help complete the job faster and in a more pleasant way.

But an IDE is not only an important tool in a programmer's arsenal, it can also be a valuable didactic tool. In our own experience teaching "newbies" how to program we often came to the conclusion that IDE's are far less intimidating to newcomers than a black screen and a blinking line. And since our expected userbase will consist of mathematical researcher (with little or none programming skills) and students, developing an IDE became mandatory for this project.

Furthermore, after having struggled for years with students not being able to install Prover9 on their particular hardware/software configuration, it became clear that the solution would be to have the IDE running on the cloud as a web application.

Having an IDE as a web application brings several advantages:

[18] https://www.cs.unm.edu/~mccune/mace4/gui/v05.html

1. There is no need to install software (non-technical users usually have a hard time using the command line).
2. It eliminates the problem of conflicting libraries and versions.
3. Users are always certain of using the latest version.
4. It can be a time saver for teachers in a classroom environment.
5. ProverX could be installed on a cluster of fast servers giving researchers the possibility to launch huge jobs and get quicker answers.

1.4 - Conclusion

To summarize, we could say that the aim of this project is to have an ubiquitous and integrated environment for working with automated reasoning.

1.5 - Thesis outline

- Chapter 1 starts with an overview of what is automated reasoning and its history. It then proceeds to explain the motivations and goals for ProverX.
- Chapter 2 explains the design decisions made during the development of the ProverX project.
- Chapter 3 explains the architecture of the project both on the server side as on the client side.
- Chapter 4 shows how ProverX works in a form of increasing complexity tutorials. We also include an article co-authored by the author of this project, illustrating a real-case example where ProverX was used to prove a mathematical theorem in semigroups theory.
- Chapter 5 introduces a submitted paper that illustrates the use of ProverX in proving a theorem by exhaustion.

- Chapter 6 concludes with a summary of the results achieved so far and also highlights some ProverX's limitations, indicating future work and giving credit to all software tools used in this project.

Finally, these appendices were added:

- Appendix A gives a brief overview of the Graphical User Interface.
- Appendix B is a quick start tutorial for new users.
- Appendix C is an extensive manual to the ProverX Scripting Library

2 - Design decisions

2.1 - ProverX

2.1.1 - Main prover engine

The first and most important decision was to leave Prover9 and Mace4 source code untouched and create a wrapper around it written in C language (the original language of Prover9/Mace4). This thin wrapper would be responsible for calling instances or Prover9 or Mace4 (by forking), interface this calls with the scripting engine and manage the command line interface.

This strategy was chosen mainly to avoid introducing new bugs in a working piece of software that has already passed the test of time. The downside of it is that, since the original functions of prover9/mace4 only accept files as input, the scripting language must create temporary files to pass data to the prover engine. Although this doesn't create an efficiency problem in terms of speed as most of the computational time is spent in the prover engine.

As a side note, there is another Ph.D. project on the works by a DAC (Doutoramento em Álgebra Computacional) student at Universidade Aberta that aims at rewriting the LADR (and thus Prover9 and Mace4) library in C++. In a near future these two projects may be merged, and this could lead to more elegant solution that avoids using temporary files.

One important decision was to make sure that source code would compile seemingly on Linux and MacOS systems, but no effort was made to make it so for windows systems. This decision was largely due to the fact that ProverX will run on a Linux web server and keep the compatibility of source code with windows would complicate the development seriously (furthermore the new Linux

Subsystem for Windows [19] available on Windows 10 will make the installation on windows machines as easy as on Linux).

2.1.2 - Choice of the scripting language

The second fundamental decision was the choice of the scripting engine.

We first tried to identify typical users of the project and found two:

1) Mathematical researchers
2) Logic and algebra students.

What these users have in common is a high probability of having none to little prior programming knowledge.

So, with that in mind, we arrived at this list of desired features:

1) The scripting language should be easy to learn (clean and intuitive syntax)
2) Be popular enough so the probably of being known by users with prior programming knowledge would be high.
3) Have a strong base of users and third-party libraries.
4) Speed was not the most important factor considered because most of the "heavy" work is done by the prover engine.

Some more technical requirements would also need to be met:

1) Having an easy integration with the C language (for integration with prover9/mace4 source code), and a thorough documentation on how to do so.
2) Have built-in capabilities to ease the creation of a DSL (or mini-language) like operator overloading, introspection, metaprogramming, etc.

[19] https://docs.microsoft.com/en-us/windows/wsl/install-win10

3) Although not mandatory, being object-oriented would be preferable since this is the predominant programming paradigm nowadays.

The first set of features immediately ruled out our first idea of creating a custom-made language with a LISP syntax specifically for this project. The reasons were:

1) Lack of time, as creating a small language is a time-consuming endeavour.
2) Beginners usually find prefix notations counter-intuitive and LISP (or Scheme) syntax, although extremely simple, is not very user-friendly.

For popularity, we consulted the IEEE page "The Top Programming Languages 2019" [20] where we can see that Python retains its leadership from previous years:

Figure 2 - The Top Programming Languages 2019 according to IEEE ranking

[20] https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

After that we turned to the technical aspect and searched for embeddable programming languages (meaning languages that can be embedded in another language – preferably C – and serve as scripting language). After considering several options, four of them were retained: Lua, Ruby, Javascript and Python.

Lua [21]:

Lua has a very small footprint and is almost trivial to embed. It has a very easy and pleasant syntax and was a strong contender. Unfortunately, it is not object oriented (although this can be simulated with tables) which doesn't make it very well suited for creating a large library. Furthermore, it is not as popular as the other three,

Ruby [22]:

Ruby has a strong tradition of being well suited to create DSL's (Ruby on Rails [23] being the paradigmatic example). It has a cleaner syntax than Python specially for classes (no annoying self references in method definitions) but it is not easy to embed in C programs (at least, not as easy as Lua or Python).

Javascript:

Javascript was designed as a scripting language for web pages where it gained popularity amongst web developers. Its is not easy to embed in C programs and its object-oriented features are quite peculiar to grasp. Although is a very popular language, it doesn't meet the needed criterias.

Python [24]:

Python is the oldest language of the four. As we have seen it is also the most popular. It has a simple and easy syntax to learn (although the indentation can be sometimes confusing) and it is very easy to embed in a C program (although, in our opinion, Lua is easier).

---

[21] https://www.lua.org/
[22] http://www.ruby-lang.org/en/
[23] https://rubyonrails.org/
[24] https://www.python.org/

Although speed was not a fundamental requirement, for completeness sake we searched for some benchmarks comparing these languages for speed. We are aware that comparing languages is a controversial matter, subject to biases, and easily prone to fall in "language wars". Nevertheless, we present here a benchmark [25] that clearly shows Python a fast language:

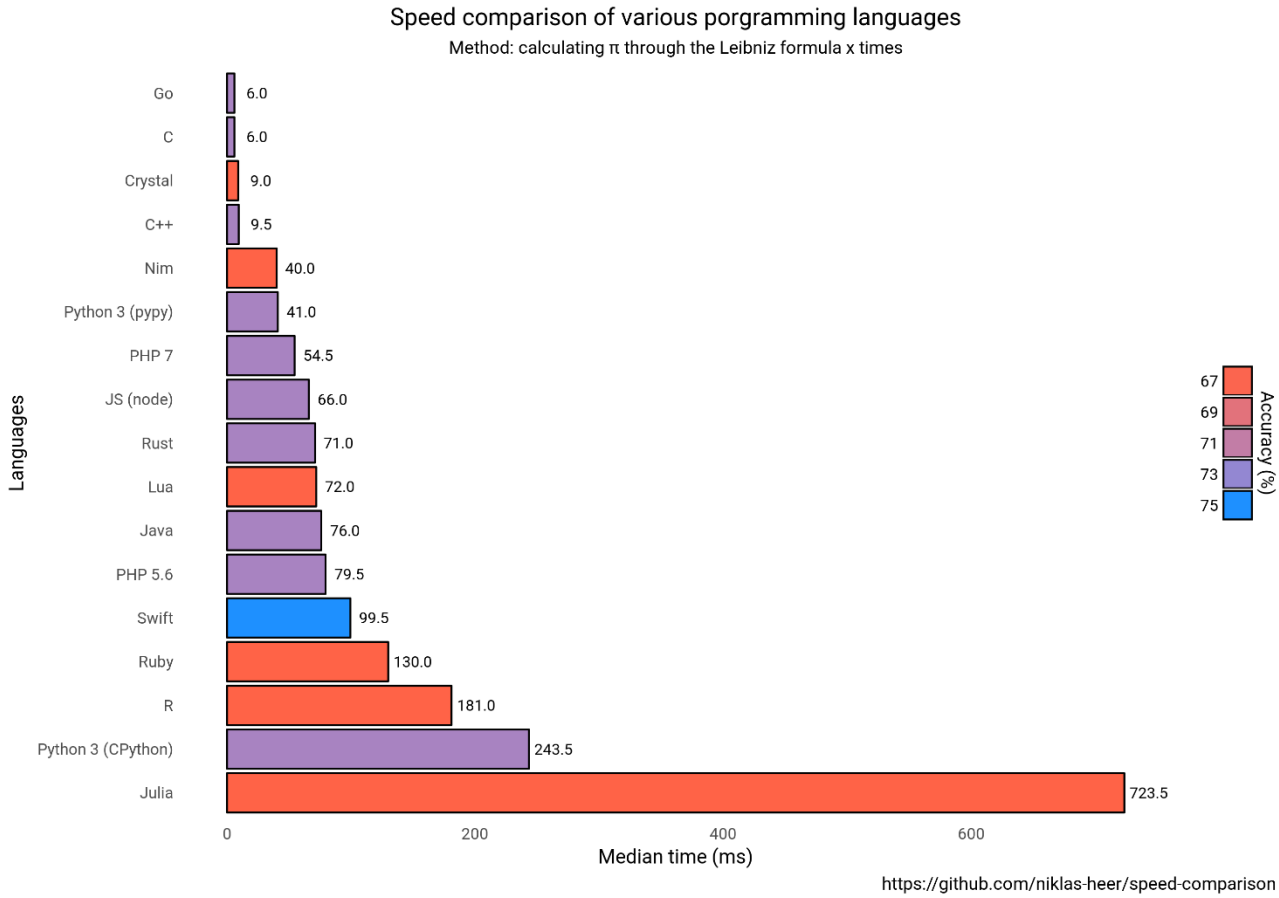


Figure 3- Speed comparison of programming languages

Finally, we chose Python for the following reasons:

1. Robust and well tested (26 years)
2. Easy to learn.
3. Used prominently by the scientific community.
4. Very good built-in library and lot of 3rd party modules.

[25] https://github.com/niklas-heer/speed-comparison

5. Has a large user base and enthusiast community.
6. Easy to embed in a C program.
7. Object oriented.
8. Good features for DSL's (like "magic methods", operator overloading, decorators, etc.)

2.1.3 - Scripting Library

By the time the C wrapper was written, it became clear that the true strength of this project would lie on his Python library. This is where the most time and effort were spent (alongside with the user interface).

2.1.3.1 - Parsing engine

The first and more important decision was how to manage prover9 input and output. Since the interaction between Python and prover9/mace4 is made through text files, some kind of parsing is necessary.

To extract the results, we decided to parse the output files using simple regular expressions.

But for the input, we soon realized that parsing the prover9 syntax (which presents some of the challenges found in programming languages like parentheses matching, etc.) was a little more involved and a parsing engine was a better choice.

The whole point of having a scripting language resides not only in the possibility of extracting the axioms (or goals) from the input file, but also, we must be able to modify them, add new ones, etc. In most advanced cases the user may not even use an input file but will want to create the axioms and goals programmatically.

So, it became clear that a more sophisticated data structure was needed as shown on Figure 4:

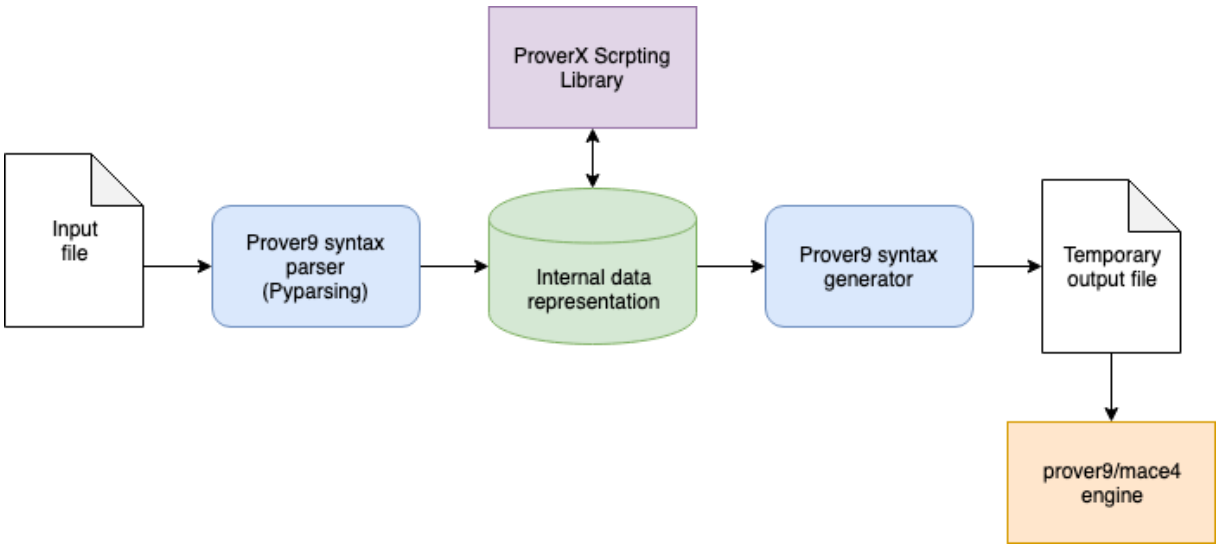


Figure 4- Parsing mechanism

Writing a parser by hand, even a recursive descent parser, is a time-consuming task, so we tried to find a parser engine for Python. Luckily, there is a vast choice of parser engines available for Python (as matter of fact, luck has nothing to do with it, since a rich ecosystem of third-party libraries was one of the main reasons we chose Python as the scripting language).

Being used to work with YACC/FLEX [26], we first looked for Context-Free parser generators. We started with ANTLR [27] and Lark [28] , but in both cases the grammar must be coded in an external file and a generator will create a parser in python (furthermore ANTLR is written in java). This approach may be fine to create full-fledged languages but can become very cumbersome for our simpler need.

We then moved to PLY [29] wich replicates YACC and LEX but lets the programmer build the lexer and tokenizer inside a Python file. We found that this feature can speed the programming workflow considerably.

[26] http://dinosaur.compilertools.net/
[27] https://www.antlr.org/
[28] https://github.com/lark-parser/lark
[29] https://github.com/dabeaz/ply

For completeness sake, we also investigated Arpeggio [30] which is based on PEG formalism and looked very promising. One of its main advantages is that the parser grammar can be expressed with python methods. For example:

```
from arpeggio import Optional, ZeroOrMore, OneOrMore, EOF
from arpeggio import RegExMatch as _

def number():     return _(r'\d*\.\d*|\d+')
def factor():     return Optional(["+","-"]), [number, ("(", expression, ")")]
def term():       return factor, ZeroOrMore(["*","/"], factor)
def expression(): return term, ZeroOrMore(["+", "-"], term)
def calc():       return OneOrMore(expression), EOF
```

This led us to the discovery of Pyparsing [31] which we finally adopted as our parser engine. This choice was dictated by several criteria:

- It was written in 2003 but is still actively developed (an older library usually translates into better tested, less bugs and larger user community).
- Simple to use (like Arppegio, the grammar can be written using python methods in a very intuitive and "pythonic" manner).
- Simple to insert in a project (just one file, no need to install a module on the computer).

2.1.3.2 – Class hierarchy

The second concern was either to create two different classes for prover9 and mace4 objects or not.

At first, the more obvious solution was to have to separate classes with an ancestor class containing the common functionality (reading files, parsing, etc).

It then became clear that input files containing axioms and goals, can either generate a proof or a model (which sometimes can only be known at runtime) which led to the design decision of having a unique class for finding both models and proofs.

[30] https://textx.github.io/Arpeggio/stable/
[31] https://github.com/pyparsing/pyparsing

This class, called proverX, is in fact the main class of the scripting library and can be instantiated with an input file or a python multiline string. The proverX class can then find models or proofs and store the results in a model class or a proof class.

A clearer picture is presented in the class diagram in the architecture chapter.

2.2 - Web app

As explained on section 1.3.2, after the need for an IDE was felt, we decided to create a web application that would try to emulate as close as possible a desktop IDE.

This kind of project brings several challenges namely:

1) Web pages are by nature stateless and some kind of mechanism must be implemented to maintain data storage.
2) The server cannot access the user file system and a solution must found to transfer files back and forth from the user to the server.
3) HTML was designed to be a simple interface akin to a word processor with text and images flowing from top to bottom. Creating graphical elements and user interfaces that mimics the ones found in modern desktops can be a daunting task.
4) Security issues.
5) Psychological resistance from users who prefer to keep their data on their own drives. (We will not address this issue here).

It is impossible for a single programmer to create a complex web application from scratch in a few months without using already made frameworks or libraries.

We will next discuss our choices in selecting those libraries/frameworks.

It is important to note that we only searched for free and open source solutions.

2.2.1 -The front-end

The front-end relates to what runs in the user browser and is responsible for the user interface and how the user interacts with the application.

2.2.1.1 – The core

For a web app is almost mandatory to use javascript for the front-end because of its ubiquity amongst all modern browsers. One can use another language for its syntax facility or language paradigm like TypeScript [32] or ClojureScript [33], but these are ultimately translated in javascript.

So, the real question that a web developer first faces is: what framework to use on top of javascript?

There is a very large offer of javascript frameworks but, by experience, we firmly believe that is preferable to choose amongst the most popular ones because it always pays off when the need for help arises.

With that in view, we first contemplated using Meteor [34], AngularJS [35], the simpler VueJS [36] frameworks or even more simply: Bootstrap [37] (strictly speaking, Bootstrap is also a HTML and CSS framework).

Although most of modern frameworks like these ones promise "build apps faster", the truth is that there is learning curve associated to it and as the version number increases, the steeper this learning curve becomes.

Furthermore, the workflow associated with some of these frameworks is becoming increasingly complex forcing the programmer to install several tools like transpilers and CLI tools.

[32] https://www.typescriptlang.org/
[33] https://clojurescript.org/
[34] https://www.meteor.com/
[35] https://angularjs.org/
[36] https://vuejs.org/
[37] https://getbootstrap.com/

By examining the examples that accompany these frameworks, we arrived at the conclusion that this project was quite atypical and, more important, the javascript involved is more of a "glue" between "off the shelf" components.

So, in the end, we decided to use the venerable jQuery [38] a javascript library that can be seen as a "swiss knife" for javascript. It is a simple library to use, ubiquitous, and most of all: very familiar.

As cited on the jQuey web site:

> *What is jQuery?*
>
> *jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.*

At this point, it's important to acknowledge that this is a one person project and so, it is only normal that a programmer will have some bias and some tendency to use tools that are already familiar (this certainly also holds true for the choice of PHP for the back-end).

2.2.1.2 – Other libraries and components

The choice of jQuery created an additional constraint on the choice of the other libraries: they should be compatible with jQuery.

The first and most important component was a code editor. We looked at the best-known code editor, ACE [39] which seemed to have all the features we needed and more:

[38] https://jquery.com/
[39] https://ace.c9.io

- Syntax highlighting for over 110 languages (TextMate/Sublime Text.tmlanguage files can be imported)
- Over 20 themes (TextMate/Sublime Text .tmtheme files can be imported)
- Automatic indent and outdent
- Handles huge documents (four million lines seems to be the limit!)
- Fully customizable key bindings including vim and Emacs modes
- Search and replace with regular expressions
- Highlight matching parentheses
- Toggle between soft tabs and real tabs
- Displays hidden characters
- Drag and drop text using the mouse
- Line wrapping
- Code folding
- Multiple cursors and selections
- Cut, copy, and paste functionality

Some of the features listed above are not currently exploited (like code folding or regular expressions for searching) but probably will in future versions of ProverX.

After looking at the very thorough documentation, we concluded that this was an easy component to install and also very easy to work with jQuery.

This is also the code editor used by some of famous sites like AWS, Wikipedia, Khan Academy, etc. Therefore, we felt there was no need to look any further. This was probably the easiest decision.

After that, we the searched for a GUI library that would mimic a desktop IDE's.

Unluckily, all the libraries that looked promising were paid software. This led to an important decision: instead of having multiple resizable windows like on a desktop, we chose to simplify the IDE design by having only two code editor windows (or panes) side by side, each one containing

one or more tabs. This design allows to compare a theory with its proof, or a script with its result, side by side.

We then took some more decisions that not only simplified even further the design, but also made the interaction with the user simpler and more intuitive:

- No menus (all commands are called by pressing buttons on an icon bar)
- All parts of the GUI are resizable and collapsible panes

We thus arrived at the design on Figure 5.

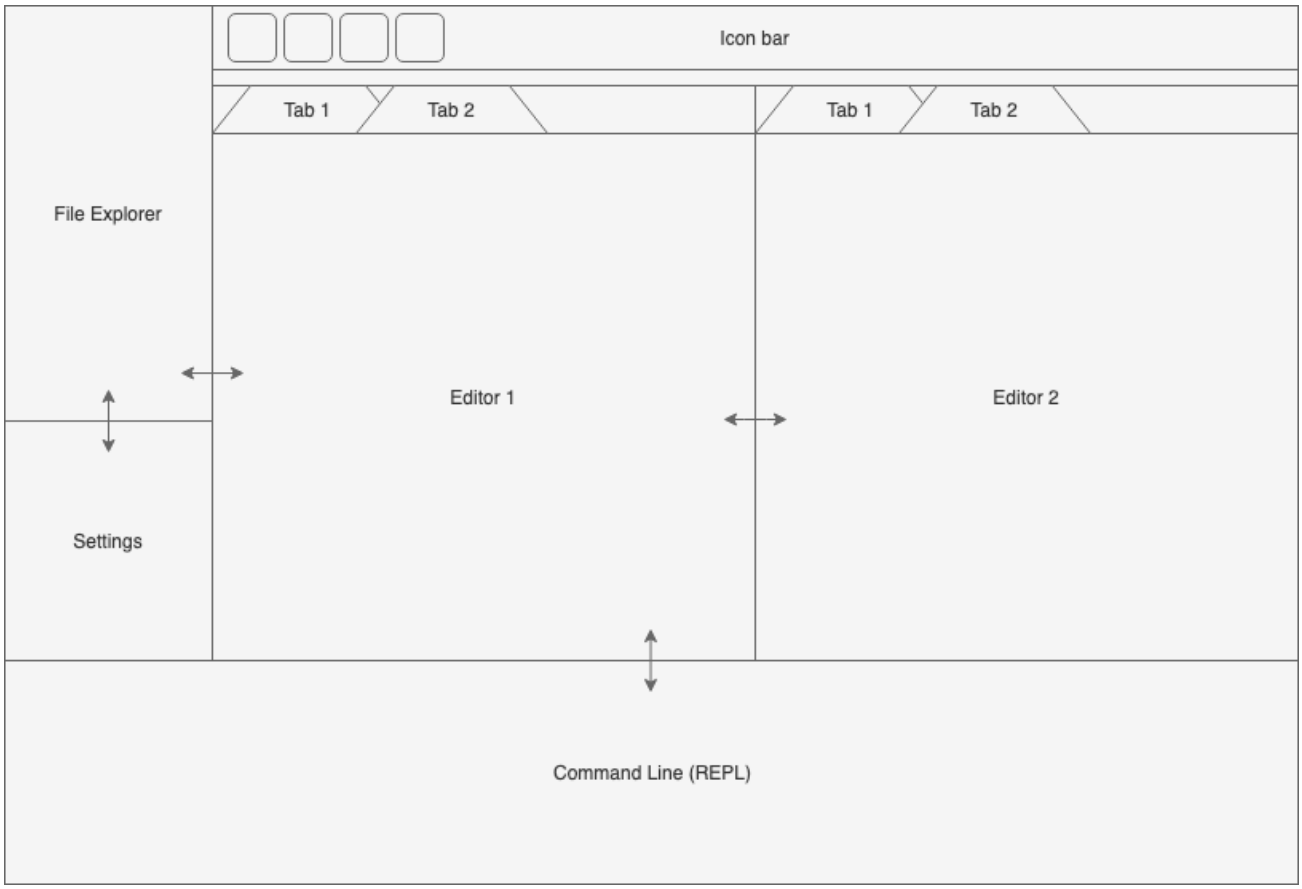


Figure 5- GUI design

This design allowed us to code the GUI system by hand only using jQuery and the help of a small library called Split.js [40] which proved very useful to create split resizable views.

[40] https://split.js.org/

The last piece to have a working IDE, was to find a suitable file explorer. Most components we surveyed were not compatible with jQuery (or very hard to integrate), lacked good documentation, or were visually very unpleasant.

We finally settled on jQuery File Tree [41] that has all the features we needed.

Note that this file explorer serves only to open files and navigate the tree hierarchy of the file system; for deleting, uploading or renaming files, a file manager residing on the server side is needed.

2.2.2 - The back-end

For several years, web development has been dominated by what is called the LAMP stack. LAMP is an acronym for a Linux Server, an Apache web server, a MySQL database and PHP language. This was (and in many cases still is) the cornerstones of most sites found on the web.

Although there is a lot more choice nowadays, we are almost sure to find this stack on a linux system. And, since we didn't know at design-time were ProverX would be deployed, it looked a safe bet to use a traditional approach and so we chose PHP as the language for the back-end.

Since ProverX runs on a terminal and can emulate prover9 and mace4, it made sense to have a command line window on our system. It turned out that interacting with a command line program running on a server through the web is very complicated to achieve with plain PHP.

The answer to this problem was to use web sockets and we started to code a web socket server in node.js [42]. Fortunately, we found early in the process that such a server emulating a terminal over the web, already exists and is called TTYD [43].

[41] https://github.com/jqueryfiletree/jqueryfiletree
[42] https://nodejs.org/
[43] https://github.com/tsl0922/ttyd

Finally, the problem of uploading user files to server (as well as renaming and deleting files) arose and led us to choose Roxy Fileman [44] which is a file manager written in PHP that runs on the server.

[44] http://www.roxyfileman.com/

3 - Architecture

In this chapter we present different diagrams that illustrate the architecture of the ProverX project.

We start with a global view of the project:

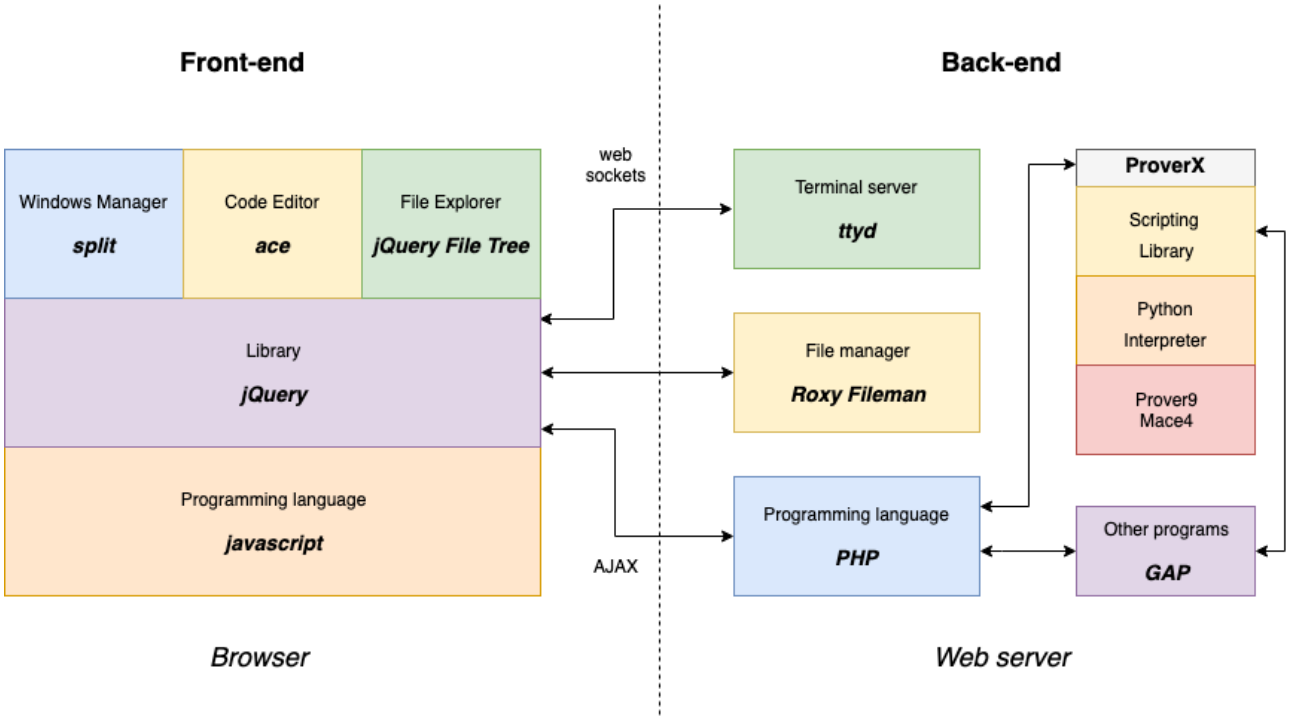


Figure 6- Overall architecture

3.1. - ProverX

ProverX consists of a thin layer of C code compiled against the Python Library and the LADR library (which is the main library of prover9 and mace4) to obtain the ProverX executable.

There are only three files:

proverx.c is the main file that parses the command line arguments, initialize the Python interpreter, adds the function that call prover9 and mace4 to the scripting engine and run the REPL if needed.

provernine.c and **macefour.c** contains the code to emulate prover9 and mace4 respectively by calling functions in the LADR library.

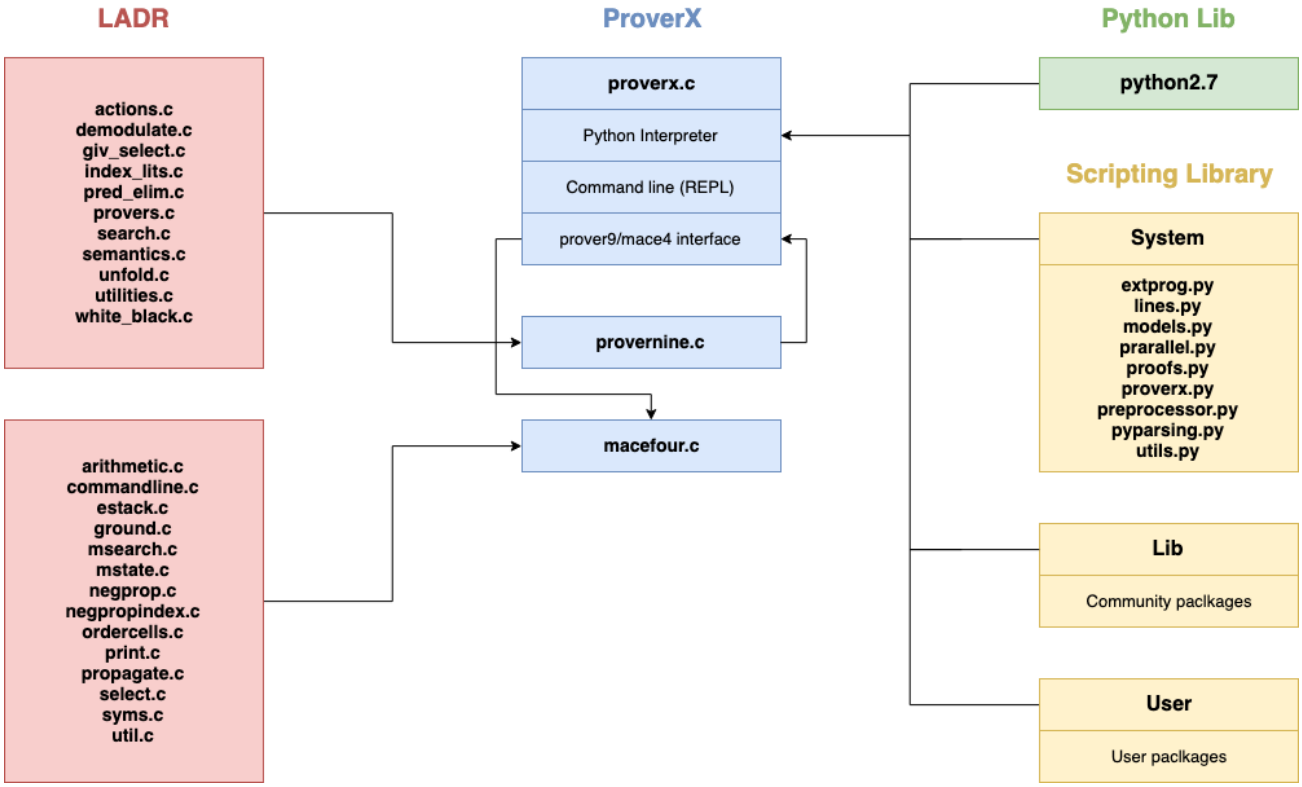


Figure 7- ProverX source code diagram

3.2 - Scripting Library

The scripting library has several Python files, modules and classes which are distributed by three different folders:

- **System:** here we can find the main Scripting Library classes and modules.
- **Lib:** here is the place to store all contributed packages developed by the community.
- **User:** this folder is only meant to be used by user which have ProverX installed locally on their machines. For all purposes, it works just as the Lib folder.

The system folder is composed of the following files:

- **proverx.py:** this file contains the main proverx class which does the heavy work of parsing prover9/mace4 files, calling prover9 to find proofs or mace4 to find interpretations (also

models or counter-examples) or even both in parallel (killing the other process as soon as proof or a model is found).

- **parallel.py:** This class allows to run several prover9/mace4 jobs in parallel. This class is used internally by Proverx class for its find_both() method.
- **preprocessor.py:** This class is automatically called by Proverx on initialisation. it will preprocess the input file makes it possible to add new features to prover9 syntax like an exponentiation operator, include directive, for loops and define functions by induction.
- **models.py:** This module contains the class Models that acts as an array of interpretations and the class interpretations that encapsulates all attributes about a model like functions, arity, size, etc.
- **proofs.py:** This module contains the following classes:
    - **Proofs:** which acts as an array of proofs.
    - **Proof:** This class encapsulates one proof. It acts as an ordered dict of clauses (see ProofClause class) indexed by the id of the clause.
    - **ProofClause:** This class encapsulates all the attributes of a clause.
    - **GivenClause:** This class encapsulates all the attributes of a given clause.
    - **ProofStat:** This class gives statistic informations about each proof.
    - **ProofStats:** This class gives statistic informations about all the proofs.
- **utils.py:** contains some utility functions.
- **extprog.py:** This module contains the class Exec that executes an external program and interacts with its stdin, stdout streams, check if it's still running, kills it, etc. It also includes Gap which inherits form Exec and is used to execute instances of the GAP algebra system.
- **lines.py:** is a utility class for handling text lines. It is used by axioms, goals, givens, etc. It subclasses list and adds methods for saving and opening text files, set methods (union, intersection etc.) and a very useful method to generate all subsets. It can also backup itself (create an internal copy) and restore.

The full description of the ProverX Scripting Library can be found in appendix D (or on the online documentation).

Figure 8 shows a simplified (without attributes and methods) class diagram:

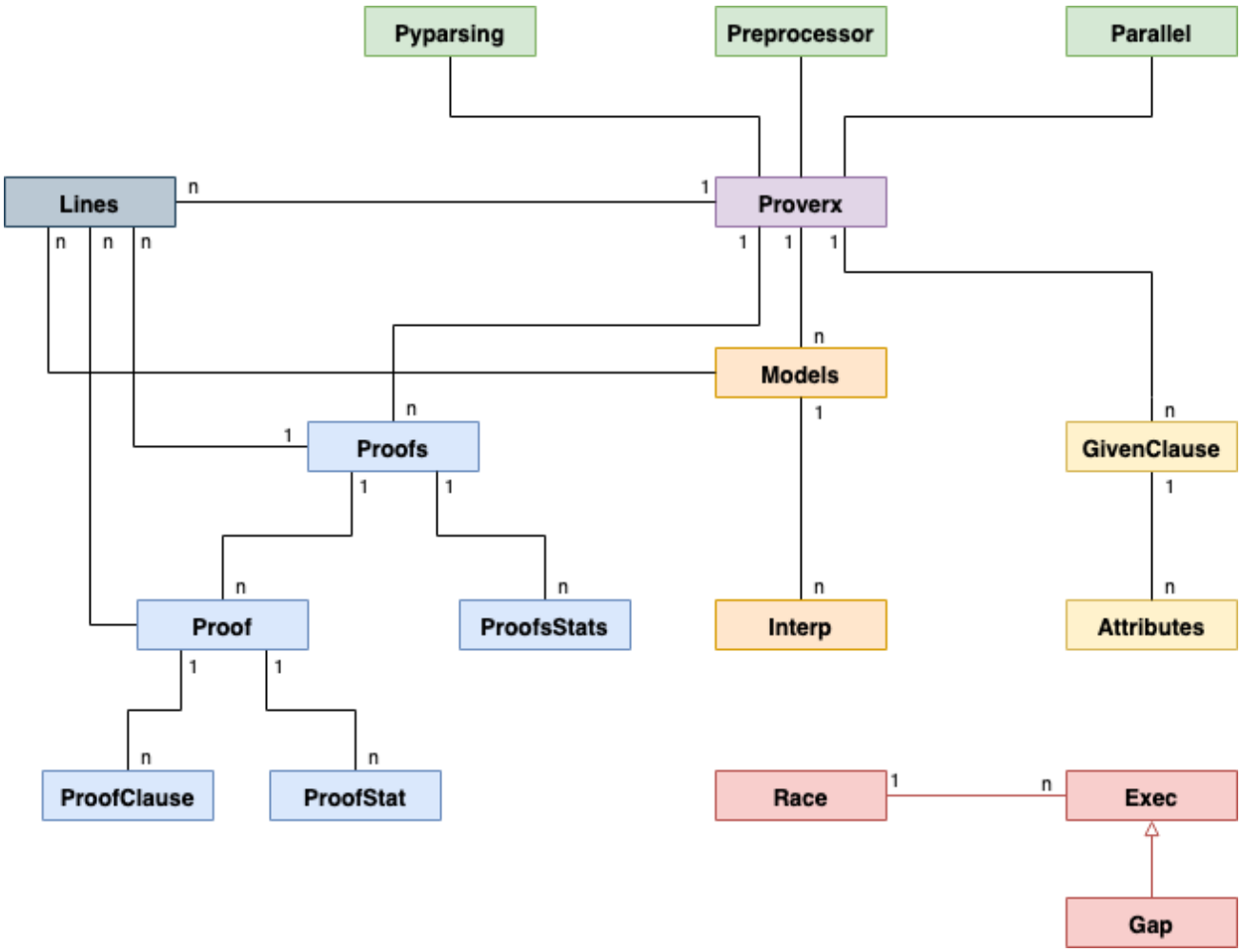


Figure 8- Scripting Library class diagram

3.3 - Web App

3.3.1 - Server side

On the server side we have the following files:

- **fileserver.php:** this is where most of the work is done. This code is responsible for receiving the POST requests and, based on them, execute ProverX, run scripts, open files, call TTYD, etc.
- **guest.php:** is responsible for creating a temporary filesystem for guest users (automatically deleted after 24 hours).
- **pass.php:** is responsible for retrieving the passwords from the database.
- **login.php:** calls **pass.php** to check the login credentials and if accepted uses tells **fileserver.php** to use that user file system.
- **logout.php:** is responsible for closing the session.

3.3.2 - Client side

On the client side we have only two files:

- **app.js**: this is the entire app. The functions on this file manage all the interaction between the GUI and the user like loading files into the editors, managing windows, tabs and button commands, posting requests to fileserver, etc.
- **doc.js:** translates the documentation files form Markdown to html.

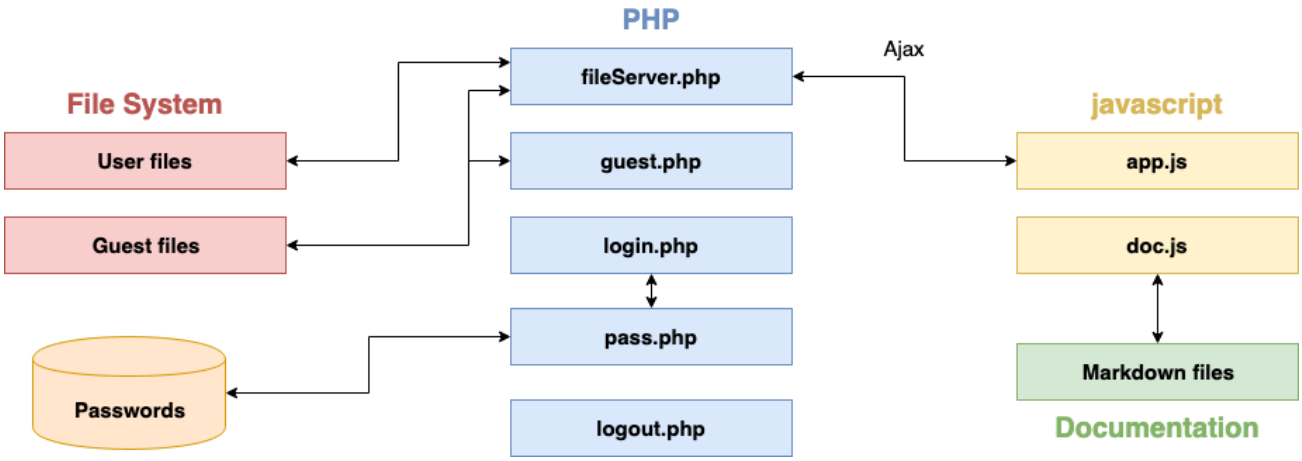


Figure 9- Web App source code diagram

4 - Practical usage of ProverX

This chapter is written as a succession of small tutorials with an increasing complexity that will show the most important features of ProverX.

4.1 - First steps

This section covers the basic aspects and the most common tasks of ProverX.

Create a Prover9 input file and find a proof

Press the New Blank Prover9 File button

ProverX will automatically write some of the text for you:

```
include(preferences).

formulas(assumptions).
% Write the axioms here:

end_of_list.

formulas(goals).
% and the goals here:

end_of_list.
```

Notice on the file explorer there is a file called "preferences" with some common settings. You can, of course, modify this file to suit your preferences.

Now let's try to prove that a group in which all elements have order 2 is commutative.

So, enter the following lines:

```
include(preferences).

formulas(assumptions).
% group axioms
  (x * y) * z = x * (y * z).    % associativity
  1 * x = x.                    % left identity
  x * 1 = x.                    % right identity
  x' * x = 1.                   % lef inverse
  x * x' = 1.                   % right inverse
```

```
  % special assumption
  x * x = 1.                  % all elements have order 2

end_of_list.

formulas(goals).
  x * y = y * x. % commutativity end_of_list.
```

press the Save File button  and call this file abel1.in.

Then press the Run with Prover9 button 

You should get a new window on the right with the result:

```
============================== PROOF =================================

% Proof 1 at 0.04 (+ 0.06) seconds.
% Length of proof is 11.
% Level of proof is 4.
% Maximum clause weight is 11.000.
% Given clauses 12.

   1  x * y = y * x # label(non_clause) # label(goal).  [goal].
   2  (x * y) * z = x * (y * z).  [assumption].
   3  1 * x = x.  [assumption].
   4  x * 1 = x.  [assumption].
   7  x * x = 1.  [assumption].
   8 c2 * c1 != c1 * c2.  [deny(1)].
   13 x * (x * y) = y.  [para(7(a,1),2(a,1,1)),rewrite([3(2)]),flip(a)].
   14 x * (y * (x * y)) = 1.  [para(7(a,1),2(a,1)),flip(a)].
   20 x * (y * x) = y.  [para(14(a,1),13(a,1,2)),rewrite([4(2)]),flip(a)].
   24 x * y = y * x.  [para(20(a,1),13(a,1,2))].
   25 $F.  [resolve(24,a,8,a)].

============================== end of proof ==========================
```

Using the preprocessor

Now, let's do exactly the same thing but using some of the preprocessor directives.

Create the following file and save it (call it abel2.in):

```
include(preferences).

formulas(assumptions).

 include(group).

  % special assumption
  x ** 2 = 1.

end_of_list.

formulas(goals).
  x * y = y * x.
end_of_list.
```

Notice that all the group theory axioms have been replaced by the directive include(group). This file contains the group axioms and is stored on the server; you can also create your own include files (if you create a file called "group" it will take precedence over the default one). Notice also that the special assumption now uses the exponentiation directive.

Press the Run with Prover9 button and you should get exactly the same result.

Now, if you look at the file tree on the left, you will notice a new file called abel2.in.tmp (this is the actual file that is passed to ProverX). Click on that file to reveal an identical file to abel1.in (note that .tmp files are not editable).

Syntax checking

To check what the preprocessor did to your input file, just press the Syntax Check button: 

A new pane will open side-by-side with your original code, revealing the .tmp file that will be passed to ProverX.

But the main purpose of this button is to find errors in your input file. For instance, remove the last dot in "x * y = y * x" and press the button.

You should now see the offending line highlighted in the .tmp file:

```
15
16  formulas(goals).
17
18  x * y = y * x
19
20  end_of_list.
21
```

Using Mace4

Now remove the special assumption ('x ** 2 = 1.') and try to run Prover9 again.

By removing this line, we are asking the program to prove that all groups are commutative. This is obviously false and you will get the message:

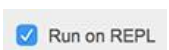
```
No proof found !
```

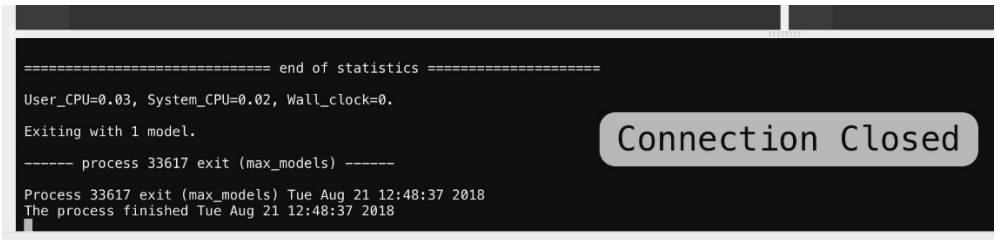
So, instead, press the Run with Mace4 button to find a model that satisfies this theory (a non-commutative group).

You should get:

```
============================== MODEL =================================

interpretation( 6, [number=1, seconds=0], [

        function(c1, [ 0 ]),

        function(c2, [ 2 ]),

        function('(_), [ 0, 1, 2, 4, 3, 5 ]),

        function(*(_,_), [
               1, 0, 3, 2, 5, 4,
               0, 1, 2, 3, 4, 5,
               4, 2, 1, 5, 0, 3,
               5, 3, 0, 4, 1, 2,
               2, 4, 5, 1, 3, 0,
               3, 5, 4, 0, 2, 1 ])
]).

============================== end of model ==========================
```

which is the smallest non-commutative group.

To see this result in the REPL instead of a window, just check the Run on REPL 



This can be useful when there are errors on your input file / script.

Using the command line

Open the ProverX REPL 

and type:

```
p = Proverx('abelian.in')
```

then type:

```
p.find_proofs()
```

and:

```
p.proofs[0]
```

you should see:

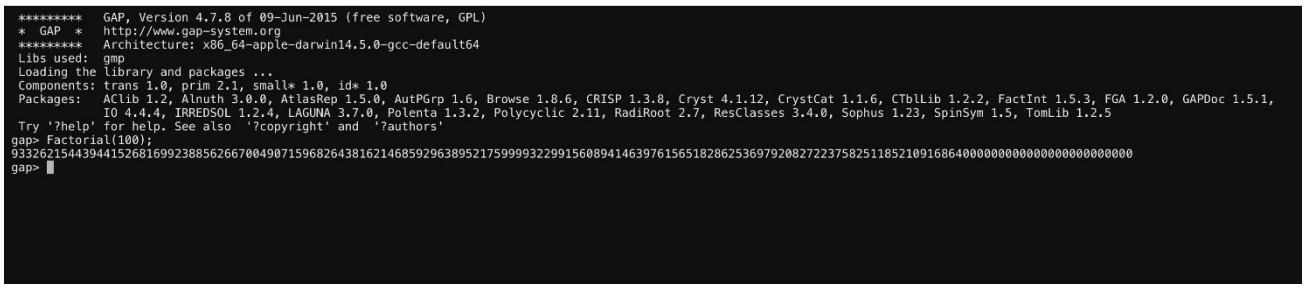
```
ProverX 1.6.2 - Universidade Aberta, Yves Robert - 2018

Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>p = Proverx('abelian.in')
>p.find_proofs()
1
>p.proofs[0]
1 x * y = y * x # label(non_clause) # label(goal).  [goal].
2 (x * y) * z = x * (y * z).  [assumption].
3 1 * x = x.  [assumption].
4 x * 1 = x.  [assumption].
7 x * x = 1.  [assumption].
8 c2 * c1 != c1 * c2.  [deny(1)].
13 x * (x * y) = y.  [para(7(a,1),2(a,1,1)),rewrite([3(2)]),flip(a)].
14 x * (y * (x * y)) = 1.  [para(7(a,1),2(a,1)),flip(a)].
20 x * (y * x) = y.  [para(14(a,1),13(a,1,2)),rewrite([4(2)]),flip(a)].
24 x * y = y * x.  [para(20(a,1),13(a,1,2))].
25 $F.  [resolve(24,a,8,a)].
>
```

IMPORTANT: when you are not using the command line, to reduce bandwidth on the server, you should disconnect the REPL 

(The first time, you do this on a session, you might get an alert message from your browser saying: "Are you sure you want to leave this page?" , just press "Leave page" and ignore it.)

If you want to hide the REPL, just press: 

You can also interact with gap 



Creating scripts

We will do the same as above, but this time using a Python script.

Start by creating a new file with the text below and save it with a .py extension (for instance abel.py)

Enter this text:

```
p = Proverx('abelian.in')

print 'Axioms:\n'
print p.axioms

print 'Goals:\n'
print p.goals

print 'Proofs found:\n'
print p.find_proofs()

# Shows the proof
print p.proofs
```

and press the Run Script button 

You should get this answer:

```
Axioms:
(x * y) * z = x * (y * z)
1 * x = x
x * 1 = x
x' * x = 1
x * x' = 1
x * x = 1
Goals:

x * y = y * x
Proofs found:

1
============================== Proof 1 ==============================

Proof 1 in 0.01 seconds .
Length of proof is 11.
Level of proof is 4
Maximum clause weight is 11.0.
Given clauses 12.

1 x * y = y * x # label(non_clause) # label(goal). [goal].
2 (x * y) * z = x * (y * z). [assumption].
3 1 * x = x. [assumption].
4 x * 1 = x. [assumption].
7 x * x = 1. [assumption].
8 c2 * c1 != c1 * c2. [deny(1)].
13 x * (x * y) = y. [para(7(a,1),2(a,1,1)),rewrite([3(2)]),flip(a)].
14 x * (y * (x * y)) = 1. [para(7(a,1),2(a,1)),flip(a)].
20 x * (y * x) = y. [para(14(a,1),13(a,1,2)),rewrite([4(2)]),flip(a)].
24 x * y = y * x. [para(20(a,1),13(a,1,2))].
25 $F. [resolve(24,a,8,a)].
```

A neat trick is to use Python's triple-quoted strings. This way, you can have your Prover9 input and your script in the same file:

```
abel = """

formulas(assumptions).

% group axioms
(x * y) * z = x * (y * z).   % associatitivity
1 * x = x.                   % left identity
x * 1 = x.                   % right identity
x' * x = 1.                  % lef inverse
x * x' = 1.                 % right inverse

% special assumption
x * x = 1.                  % all elements have order 2
```

```
end_of_list.


formulas(goals).

x * y = y * x.

end_of_list.
"""

p = Proverx(abel)
p.find_proofs()
print p.proofs[0].stats.max_weight
```

Run it and you will get:

```
11.0
```

As you keep running scripts, the result windows begins to accumulate. If you try to close them, you will get a warning that you are trying to close an unsaved file. This can become annoying when there are a lot of them. So, if you want to close all results without warnings, just press the Close all results button 

GAP scripts

You are not limited to Python, you can also write GAP scripts.

For instance, suppose you want to know the order as well as the elements of the symmetric group S4.

Just create a file a save it with the extension .g

Enter the following gap instructions:

```
S4 := SymmetricGroup(4);
Print("Order of S4: ", Order(S4), "\n\n");
Print("Elements:\n");
Display(Elements(S4));
```

Then press the Run Script button  . This time the GAP interpreter will be invoked (because of the .g extension in the file name). The result will be:

```
Order of S4: 24

Elements:
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3),
  (1,2,3,4), (1,2,4,3), (1,2,4), (1,3,2), (1,3,4,2), (1,3), (1,3,4),
  (1,3)(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), (1,4,3), (1,4), (1,4,2,3),
  (1,4)(2,3) ]
```

4.2 - Using strategies

Having a programming language means that we can extend ProverX anyway we see fit.

One of the first needs that comes to mind is to use the power of Python to create strategies that allows us to attack complex proofs (see the module strategies).

Note that the files for these examples are in the Tutorials folder.

Using hints

Let's now take an input file (hard.in) with some hard theorems to prove:

```
% assign(new_constants, 1).

assign(eq_defs, fold).

set(restrict_denials).

formulas(assumptions).

% Veroff's 2-basis for BA in terms of the Sheffer stroke.

  f(x,y) = f(y,x).
  f(f(x,y),f(x,f(y,z))) = x.



% Define a new operation (which turns out to be complement).
% The "assign(eq_defs, fold)" above causes this definition to be
% oriented as a rewrite rule so that the operation is introduced
% whenever possible.

x' = f(x,x).

end_of_list.

formulas(goals).

  % Sheffer basis for Boolean Algebra

  f(f(x,x),f(x,x)) = x  # label(Sheffer_1).
  f(x,f(y,f(y,y))) = f(x,x)  # label(Sheffer_2).
  f(f(f(y,y),x),f(f(z,z),x)) = f(f(x,f(y,z)),f(x,f(y,z))) # label(Sheffer_3).

end_of_list.
```

And the file easy.in has and easier proof of the same theorems (by adding an extra assumption):

```
% assign(new_constants, 1).

assign(eq_defs, fold).

set(restrict_denials).

formulas(assumptions).

% Veroff's 2-basis for BA in terms of the Sheffer stroke.

  f(x,y) = f(y,x).
  f(f(x,y),f(x,f(y,z))) = x.

  f(x,f(y,f(x',z))) = f(x,y').  % extra assumption

% Define a new operation (which turns out to be complement).
% The "assign(eq_defs, fold)" above causes this definition to be
% oriented as a rewrite rule so that the operation is introduced
% whenever possible.

x' = f(x,x).

end_of_list.

formulas(goals).

  % Sheffer basis for Boolean Algebra

  f(f(x,x),f(x,x)) = x  # label(Sheffer_1).
  f(x,f(y,f(y,y))) = f(x,x)  # label(Sheffer_2).
  f(f(f(y,y),x),f(f(z,z),x)) = f(f(x,f(y,z)),f(x,f(y,z))) # label(Sheffer_3).

end_of_list.
```

If we run Prover9 on these two files, hard.in will take a lot longer to obtain a proof than easy.in.

A strategy to find a quicker proof for the hard case would be to extract the hints from the easy proof and use them to prove the hard one.

This can be easily done (and tested) with this script (remember to give this file the .py extension to be runnable):

```
easy = Proverx('Tutorials/easy.in')
hard = Proverx('Tutorials/hard.in')

easy.find_proofs()
print "easy proof:", easy.proofs.stats.time, "secs."

hard.find_proofs()
print "hard proof:", hard.proofs.stats.time, "secs."

hard.hints.add(easy.proofs.hints)
hard.find_proofs()
print "hard with hints:", hard.proofs.stats.time, "secs."
```

You can verify that the hard proof with the hints should be a lot faster to obtain:

```
easy proof: 0.24 secs.
hard proof: 1.91 secs.
hard with hints: 0.27 secs.
```

Using interpretations lists

Suppose that you have this file (LT-82-2.in):

```
if(Prover9).
    assign(order, kbo).
    assign(max_weight, 25).
end_if.

formulas(sos).

% lattice theory

x v y = y v x.
(x v y) v z = x v (y v z).
x ^ y = y ^ x.
(x ^ y) ^ z = x^ (y ^ z).
x ^ (x v y) = x.
x v (x ^ y) = x.

end_of_list.

formulas(sos).
(x ^ y) v (x ^ z) = x ^ ((y ^ (x v z)) v (z ^ (x v y))) # label(H82).
end_of_list.

formulas(goals).
x ^ (y v (x ^ z)) = x ^ (y v (z ^ ((x ^ (y v z)) v (y ^ z)))) # label(H2).
end_of_list.
```

If you run Prover9, it will take some time to find a proof (more than 15 minutes, depending on your hardware).

One strategy would be to find a model of a subset of the axioms and add it to an interpretation list. This way, Prover9 can discard some clauses quicker.

Just by observing the input, we can see that it should be easy to find a model for the lattice theory.

So let's try this script:

```
p = Proverx('LT-82-2.in')
last = p.axioms.pop()       # remove the last axiom (H82)
if p.find_models():
    print p.models[0]
```

the answer is:

```
interpretation( 6 , [number = 1, seconds=0], [

    function(c1, [0]),

    function(c2, [1]),

    function(c3, [2]),

    function(^(_,_), [
                      0, 3, 3, 3, 0, 3,
                      3, 1, 5, 3, 1, 5,
                      3, 5, 2, 3, 2, 5,
                      3, 3, 3, 3, 3, 3,
                      0, 1, 2, 3, 4, 5,
                      3, 5, 5, 3, 5, 5 ]),

    function(v(_,_), [
                      0, 4, 4, 0, 4, 4,
                      4, 1, 4, 1, 4, 1,
                      4, 4, 2, 2, 4, 2,
                      0, 1, 2, 3, 4, 5,
                      4, 4, 4, 4, 4, 4,
                      4, 1, 2, 5, 4, 5 ])
]).
```

So, if we add this interpretation to the input file, we should get a quicker proof.

We can do it this way:

```
p = Proverx('LT-82-2.in')
last = p.axioms.pop()
p.find_models()
p.interps.add(str(p.models[0]))
p.axioms.append(last)
p.find_proofs()
print p.proofs
```

And we get a proof in a few seconds.

What's happening is that the model falsifies the goal theorem in the simplified axiom set. Proofs of the original theorem must include some clauses that evaluate to false in this model. (Note that two clauses that evaluate to true in the model can only produce clauses that follow from the simplifed axioms and therefore cannot deduce the goal clause.) Prover9's default clause selection strategy will choose these "false in model" clauses sooner than it would without having the model as input.

This is why this strategy can help (but doesn't necessarily help) to find a proof.

4.3 - More scripting

Proving with interpretations

Now we can generalize the previous example and wrap it in a reusable function:

```
def prove_with_interps(fname, time_out = 5):
    p = Proverx(fname)
    p.axioms.backup()
    p.set_option('Mace4', 'assign', 'max_seconds, {}'.format(time_out))
    p.set_option('Prover9', 'assign', 'max_seconds, {}'.format(time_out)) for
    axioms in p.axioms.subsets(order = 'desc'):
        p.axioms.replace(axioms)
        if p.find_models():
            p.interps.add(str(p.models))
            p.axioms.restore()
            if p.find_proofs():
                return copy(p)
    return None
```

How does it work?

What this function does is to take all subsets of the axioms (a generalisation of the previous script) and checks if a model exists for that subset. If it does it will add into the interpretation list and tries to find a proof. If a proof is found, a Proverx instance is returned, otherwise it goes on to the next subset. Note that this function already exists in the module strategies.

Writing a graph viewer for proofs

Sometimes it is very useful to see the proofs in a graphical form (for example with a directed graph). Wouldn't it be nice if we could do that with ProverX?

Not only can we do that, but most important, it is quite an easy task to accomplish!

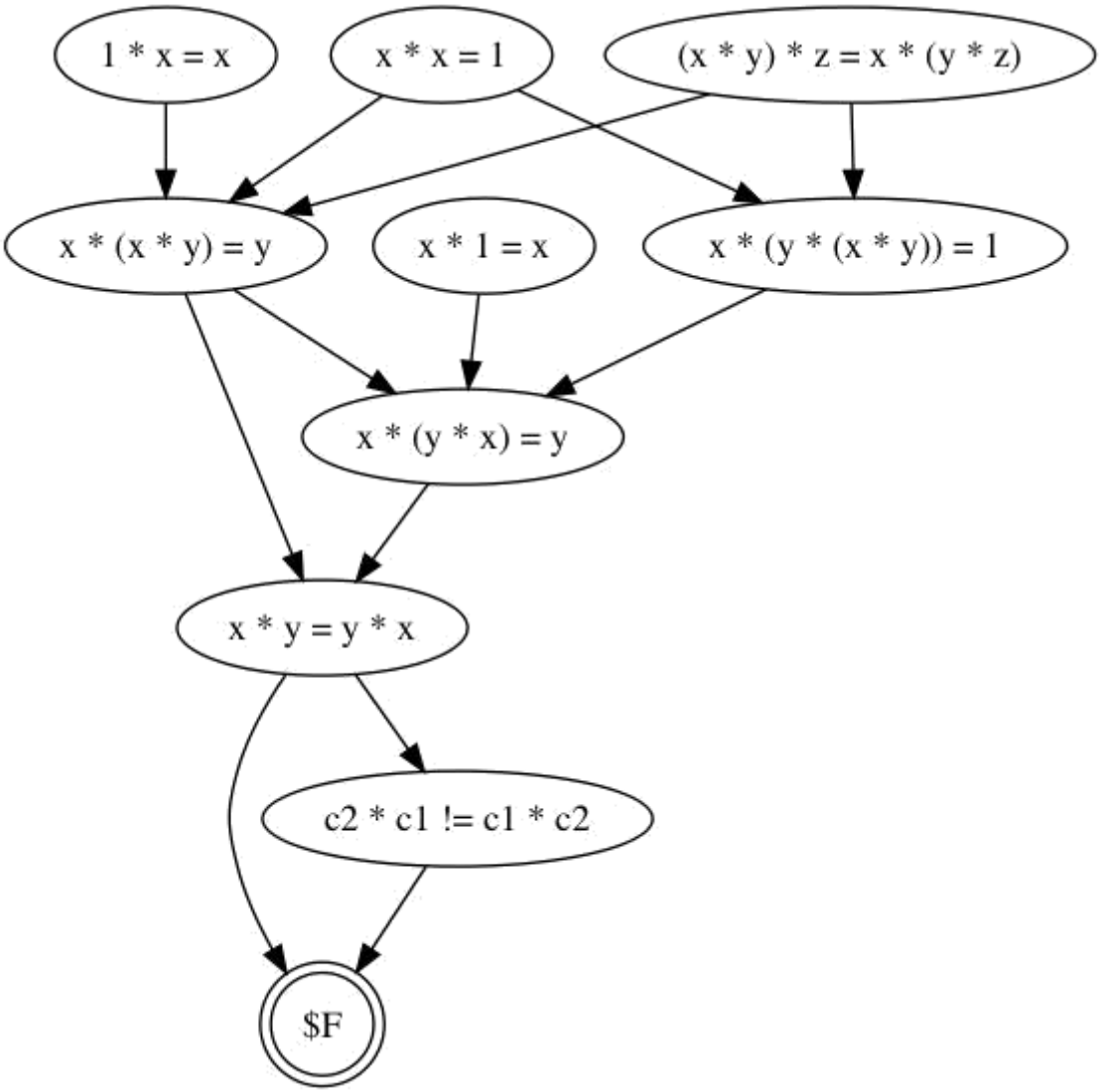
One of the advantages of using Python is that we have access to its huge third-party library.

For this project, we will use the graphviz module from Graphviz (https://www.graphviz.org/).

```
import graphviz


  def depends_from(clause):
      ids = []
      rewrites = re.findall(r"rewrite\(\[.*?\]\)", clause, re.I) #find
  rewrite justifications
      for rewrite in rewrites:
          ids += re.findall(r"([0-9]+[A-Z]*)\(", rewrite, re.I) #get clauses ids from
  rewrite
          clause = clause.replace(rewrite, '')
      paras = re.findall(r"para\(.*?\)\)", clause, re.I) #find para
      justifications for para in paras:
          ids += re.findall(r"([0-9]+[A-Z]*)\(", para, re.I) #get clauses ids from
          para clause = clause.replace(para, '')
      ids += re.findall('[0-9]+[A-Z]*', clause, re.I) # get remaining
      ids return list(set(ids)) #remove duplicate ids

  def graph(proof, filename):
      graph = graphviz.Digraph(format='png', filename = filename + '.gv')
      graph.node('$F',shape='doublecircle')
      for id, clause in proof.clauses.iteritems(): depends =
          depends_from(clause.justifications) if depends:

              for dep_id in depends:
                  graph.edge(proof.clauses[dep_id].literals, proof.clauses[id].literals)
      graph.render()

  p = Proverx('abelian.in')
  p.find_proofs()
  graph(p.proofs[0], 'graph')
```

Run this file and you should see two new files on the file tree on the left (if you don't see them, just press the Refresh button): 'graph.gv' and 'graph.gv.png'.

If you press the .png file you will see:



How does it work?

For this script, we use the ***clauses*** attribute from the class Proof.

First we create a function ***depends_from()*** that takes a justification from a clause and returns all the id's this clause depends from (rewrite, para, etc.). This is accomplished with regular expressions.

Then, the main function instantiates a graphviz. Digraph instance and for each clause in the proof, it gets the justifications from that clause and passes them to **depends_from()**. It creates the edges from all this clauses (using the literals as the name) and then it renders the graph.

5 -Varieties of regular semigroups with uniquely defined inversion

In the following pages, we introduce the paper: ***"Varieties of regular semigroups with uniquely defined inversion"***. This paper that has been accepted for publication in the journal ***"Portugaliae Mathematica"*** [45], illustrates a practical example of how Proverx was used to prove a theorem by exhaustion.

[45] https://www.ems-ph.org/journals/journal.php?jrn=pm

VARIETIES OF REGULAR SEMIGROUPS WITH UNIQUELY DEFINED INVERSION

JOÃO ARAÚJO*, MICHAEL KINYON†, AND YVES ROBERT

Abstract. Inverse semigroups and completely regular semigroups share some nice properties and in a certain sense, they are the top success stories in semigroup theory. Therefore a reasonable goal is to find other classes of regular semigroups with the same nice shared properties and try to replicate the successful structure theories achieved for those two classes. Perhaps surprisingly, the semigroup literature is mute on examples of other classes of regular semigroups with the same nice properties, while semigroupists, for decades, voiced the metamathematical conviction of the hopelessness of such a goal. Our guess is that many have tried that approach, but always failed, thus unable to publish (*the muteness of the literature*) and convinced that the path leads to nowhere (*the metamathematical conviction*).

The aim of this paper is to provide some mathematical content for that metamathematical conviction. In his celebrated theorem, K. Arrow wrote down the nice properties a voting system should have and then went on to prove that no system has those properties. We follow a similar path by writing down the nice common properties of inverse semigroups and of completely regular semigroups, and then show that, under some constraints, any variety having those nice properties is contained in one of the other two (if not in both). The proof of this theorem is by *exhaustion*: all possible varieties (under certain constraints) are written down, and each one is tested regarding the possession of the nice properties. Many pass the test, but then all of them turn out to be varieties of inverse or completely regular semigroups. This proof requires thousands of lemmas and was only possible because we used ProverX, a system aimed at playing in automated reasoning the same role GAP or MAGMA plays in symbolic computation.

The paper ends with some problems for experts in semigroups, equational logic and computer science.

* Partially supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through projects UID/MAT/00297/2019 (Centro de Matemática e Aplicações) and CEMAT-CIÊNCIAS UID/Multi/04621/2013, and also by the Fundação para a Ciência e a Tecnologia through project n. PTDC/MAT-PUR/31174/2017.

†Partially supported by Simons Foundation Collaboration Grant 359872. Partially supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through projects UID/MAT/00297/2019 (Centro de Matemática e Aplicações) and CEMAT-CIÊNCIAS UID/Multi/04621/2013, and also by the Fundação para a Ciência e a Tecnologia through project n. PTDC/MAT-PUR/31174/2017.

1. Introduction

According to J.M. Howie we value an area of pure mathematics by its power to prompt deep and elegant theories, and by its interconnections with other parts of mathematics. Among the many classes of semigroups there are, some are better at fulfilling Howie's criteria than other. Suppose thus that we want to find some new classes of semigroups fully satisfying Howie's criteria. To have better chances of identifying an interesting new class of semigroups we should first inspect those that the *giants* of the past considered interesting. Therefore our first question should be this one: *which varieties of semigroups have been considered the most interesting?*

Setting aside groups, arguably the most interesting varieties are inverse semigroups and completely regular semigroups. The first class is very interesting because it contains *real world semigroups* which arise naturally in other branches of mathematics [37, 43]; completely regular semigroups are interesting because their rich structure theory has an abstract beauty. Since such inner beauty cannot be artificial, we offer the conjecture that it is just a matter of time until completely regular semigroups will start appearing as the right mathematical tool to study *natural* objects.

Both classes of semigroups are among the special classes awarded the honor of a book fully dedicated to them. Inverse semigroups have rightfully deserved several books [37, 38, 43, 44]. Two books were originally intended for completely regular semigroups, but unfortunately only one has appeared so far [46].

Let us start by introducing formally these two classes of semigroups. They both lie inside the class of *regular* semigroups, that is, semigroups S such that for all $a \in S$ there exists $b \in S$ such that $a = aba$. These two classes are often defined semantically: inverse semigroups as regular semigroups with commuting idempotents; completely regular semigroups as unions of groups (that is, semigroups in which every element belongs to a subgroup). The semigroups in these two classes can be seen as algebras $(S, \cdot, ')$ of type $\langle 2, 1\rangle$ since they admit a natural *unary regular operation*, usually called an *inversion map*, $x \mapsto x'$ such that $x' \in V(x) = \{y \mid xyx = x \ \& \ yxy = y\}$. Both are also *pre-involuted* semigroups, that is, the unary operation satisfies $x'' = x$, where $x'' = (x')'$. (Recall that an *involuted* semigroup is a pre-involuted semigroup satisfying the additional identity $(xy)' = y'x'$.)

Finally, each class of semigroups satisfies an additional identity which distinguishes them from each other. Completely regular semigroups satisfy $xx' = x'x$, while inverse semigroups satisfy $(xx')(y'y) = (y'y)(xx')$. These are customarily taken to be part of their equational definitions.

Summarizing, from an equational point of view, the most popular classes of regular semigroups are:

(1) **inverse semigroups**, defined by associativity together with

$$x = xx'x \qquad x'' = x \qquad xx'y'y = y'yxx';$$

(2) **completely regular semigroups**, defined again by associativity and

$$x = xx'x \qquad x'' = x \qquad xx' = x'x.$$

The given identities defining inverse semigroups are due to B.M. Schein [50, Theorem 1.4.]; for the given identities for completely regular semigroups, see [32, 46].

Therefore we can move to our second question: *why are these classes of semigroups so special?* Of course there are the many deep connections with other parts of mathematics, and for this, we defer to the various standard references. Our concern now is to identify the properties possessed by these classes that led them to prompt deep and elegant theories, and build interconnections with other parts of mathematics.

From the universal algebraic point of view, they are varieties of unary semigroups and hence we can immediately apply the classical Birkhoff machinery. Also, from an equational logic point of view, the two varieties are *uniform*, meaning a variety with a base of identities such that all variables appearing on one side of an identity also appear on the other side. The equational logic folklore tells us that any set of identities equivalent to a uniform set of identities is also uniform, and this is often very useful.

Another useful fact is that in both varieties we have $x'' = x$, thus avoiding the prime explosion $x' \cdots '$, and this certainly helps to deal with the word problem on their free object. It is worth mentioning that the word problem for free inverse semigroups and for free completely regular semigroups has been solved (for free inverse semigroups see [37]; for completely regular semigroups see [35] and the references therein).

In addition, both varieties obviously contain the variety of groups. If one is going to study a new class of semigroups, it is comforting to know that groups are contained in the class. Furthermore, as each contains a non-trivial unary operation, they allow the standard group theory trick of looking at them as algebras of type $\langle 2, 2\rangle$ or $\langle 2, 2, 2\rangle$, with left and/or right divisions in the signature: $x/y = xy'$ and/or $x\backslash y = x'y$. (For group theory results arising from this approach, see the bibliography of [8]. For applications to semigroups, see [5, 8]. The use of division operations is a standard tool in quasigroup theory [13].)

Another nice property of these two classes is that the unary operation is *natural*. This is a subtle point that requires a careful analysis. Saying that *a class of semigroups $\mathcal{C}$ can be realized as a variety of unary semigroups $\mathcal{U}$* means that the semigroup reduct of every unary semigroup in $\mathcal{U}$ belongs to $\mathcal{C}$, and there are no two different unary semigroups in $\mathcal{U}$ sharing the same semigroup reduct; conversely, every semigroup in $\mathcal{C}$ is the reduct of a unary semigroup in $\mathcal{U}$.

Let us examine the case of inverse semigroups where instead of interpreting the operation $'$ as an inversion mapping, we merely interpret it as

sending each element to a corresponding *inner* inverse (or sometimes *associate*), that is, so that $xx'x = x$ holds, but not necessarily $x'xx' = x'$. Again, the semantic definition of an inverse semigroup is that it is a regular semigroup in which idempotents commute. To capture this adding a unary operation to the signature, but using just inner inverses, we have associativity ($x(yz) = (xy)z$) and regularity ($xx'x = x$), and we write the commuting of idempotents as $(xx')(y'y) = (y'y)(xx')$, since both xx' and $y'y$ are idempotents. In fact, to check that the semigroup reduct of any unary semigroup satisfying these identities is inverse, it is enough to show that the expressions xx' and $y'y$ represent all idempotents. Thus for $e^2 = e$, we have

$$\begin{aligned} ee' = (ee'e)e' = ee' \underbrace{e}\, e' = e\underbrace{(e'e)(ee')} = eee'e' \underbrace{e} \\ = e\underbrace{(ee')(e'e)}\, e = \underbrace{ee'e}\,\underbrace{ee'e} = ee = e\,. \end{aligned}$$

The argument that $e'e = e$ is dual to this. It follows that in the variety of unary semigroups defined by the three identities, idempotents do indeed commute: for $e^2 = e$, $f^2 = f$, we have $ef = (ee')(f'f) = (f'f)(ee') = fe$.

Therefore, the variety of unary semigroups axiomatized by $x(yz) = (xy)z$, $x = xx'x$ and $(xx')(y'y) = (y'y)(xx')$ has the class of inverse semigroups as its semigroup reduct. Conversely, every inverse semigroup $(S, \cdot)$ induces a unary semigroup $(S, \cdot, ')$ satisfying the three identities above, by taking $'$ to be the natural inversion mapping.

The subtile problem is that this correspondence between the variety of unary semigroups satisfying $x = xx'x$ and $(xx')(y'y) = (y'y)(xx')$ and the class of inverse semigroups is not one-to-one because in the former, $'$ need not coincide with the natural inversion and hence the same inverse semigroup might induce two different unary semigroups in the unary variety defined above. For example, in a 2-element chain with top element 1, we can take $x' = x$ (the natural inversion) or $x' = 1$. Both unary operations satisfy $x = xx'x$ and $(xx')(y'y) = (y'y)(xx')$, and hence in this variety there are two different unary semigroups with the same inverse semigroup reduct.

This means that the class of inverse semigroups is not realized as the variety of unary semigroups defined by $x = xx'x$ and $(xx')(y'y) = (y'y)(xx')$ because in the latter there are different semigroups with the same semigroup reduct.

For inverse semigroups, the class can be realized as a variety of unary semigroups by the inclusion of the identity $x'' = x$.

For completely regular semigroups, the one-to-one correspondence between the variety of unary semigroups and their semigroup reducts is guaranteed by the identity $xx' = x'x$, that is, that the inversion mapping $'$ assigns to each element a *commuting* inverse. The uniqueness follows because if, say, b, c are commuting inverses of a, then $ab = ba = baca = baac = abac = ac = ca$, and so $b = bab = cab = cac = c$ (*cf.* [46], Proposition II.1.3(ii)).

Thus both inverse semigroups and completely regular semigroups have an important feature: they can be realized as a variety of unary semigroups

because each has a *natural* choice of inversion mapping based on uniqueness of some property. In the case of inverse semigroups, the unique property is merely that of being an inverse, while for completely regular semigroups, the unique property is that of being a commuting inverse.

Other important classes of regular semigroups, such as orthodox, locally inverse or E-solid, all lack the preceding feature. In general, if we want to consider a given regular semigroup as a unary semigroup, we have to arbitrarily fix an inversion map. This idea appears in regular $*$-semigroups, introduced by Nordahl and Scheiblich [42]. However that has a drawback: a regular subsemigroup $(T, \cdot)$ of a regular semigroup $(S, \cdot)$ is not necessarily a $\langle 2, 1\rangle$-subalgebra of the chosen unary semigroup $(S, \cdot, ')$. This idea of arbitrarily fixing a unary regular operation in a regular semigroup, from the point of view of universal algebra makes sense, but does not seem to lead anywhere in semigroup theory.

To get around this problem Hall [28] introduced the concept of *existence varieties* or *e-varieties*, classes of regular semigroups closed under direct products, homomorphic images and regular subsemigroups. This is an elegant approach and is in some sense unavoidable, although an alternative to e-varieties is to consider classes of regular semigroups as those satisfying systems of simultaneous equations as suggested by Higgins and Jackson [30]. In any case, the e-variety approach adds complications as one must take care to consider only those identities which characterize an e-variety for any choice of inversion mapping. Here we rather seek to preserve the feature discussed above: we want varieties of regular unary semigroups for which there is a natural choice for the inversion mapping.

To summarize our goals, we seek to find new varieties of unary regular semigroups (that is, varieties defined by identities written on the language of a associative binary operation and of a unary operation) that share some of the nice common features of both inverse and completely regular semigroups, but are not varieties of either of them; that is we want to find new varieties such that:

(1) the following identities hold: $x = xx'x$ and $x'' = x$;
(2) there is an additional identity $u = v$ ensuring that on every semigroup in the variety there can be defined one and only one unary operation satisfying the identities of the variety;
(3) we do not want our new variety to be a subvariety of inverse or completely regular semigroups.

A variety of unary semigroups satisfying condition (1) and an additional identity $u = v$ will be denoted by $U(u = v)$; any variety $U(u = v)$ satisfying (1)–(2) above will be called an *Howie variety*, and will be denoted by $H(u = v)$. Two obvious examples of Howie varieties are inverse semigroups $H(xx'y'y = y'yxx')$ and completely regular semigroups $H(xx' = x'x)$.

Finding Howie varieties is a clear goal; and they should be very interesting varieties of semigroups as explained/suggested in the considerations above;

they should be the best candidates to emulate the very successful structure theories of $H(xx'y'y = y'yxx')$ and $H(xx' = x'x)$. There is only one detail left: are there any varieties $H(u = v)$ satisfying (3)?

Regarding goals and outcomes, this paper is analogous to Arrow's [9] search for a voting system. He started by introducing some basic properties any *fair* voting system should possess (as we did above listing nice properties an interesting variety of unary semigroups should possess), and then realized that no system can satisfy those properties.

This paper can also be seen as similar to Preston's [48] search for possible ways of defining semidirect products of inverse semigroups. He also wrote down all possible words (under certain constraints) and then used a computational tool developed at the Argonne National Laboratory to check every case.

2. Howie varieties defined by group congruent identities

When analyzing the properties of the equational definition of completely regular semigroups and inverse semigroups, we observe that each identity is group congruent, that is, is composed of words that are congruent when seen as elements of the free group. Nontrivial varieties of regular semigroups, defined by identities which are congruent in the free group and admitting a unique regular pre-involution should be the best candidates to emulate the theories of completely regular and/or inverse semigroups. However, to the best of our knowledge, the literature is mute on these varieties. In their seminal book, Clifford and Preston [14, Section 1.11, p.39] drop a metamathematical hint to remove this perplexity:

> *Considerable work has been done on axiomatics of groups. If A is a system of axioms including closure and associativity, then the proposition $P(A)$, asserting that any semigroup satisfying A is a group, or the negation of $P(A)$, logically belongs to the theory of semigroups. If $P(A)$ is false, the system A may possibly define an interesting class of semigroups. Questions of independence of axioms lead to assertions about the inclusion pattern of various classes of semigroups.*
>
> *In practice, such axiomatic studies have seldom produced interesting material for the theory of semigroups.*

A mathematical explanation for this *strange* situation (strange that an obvious idea, apparently, does not seem able to produce interesting material) might be partially provided by the following 'baby' theorem. (For an alphabet A denote by A^+ the free semigroup generated by A. Given a word $u \in \{x, y, x', y'\}^+$, the length of u is its length in the free semigroup $\{x, y\}^+$once we delete from u every occurrence of the unary operation.)

Theorem 2.1. *Let $H(u = v)$ be an Howie variety of unary semigroups defined by the following identities*

$$x = xx'x \qquad x'' = x \qquad u = v\,,$$

where $u = v$ is a uniform identity such that $u \equiv v$ in the free group. If $u, v \in \{x, y, x', y'\}^+$ and the lengths of u and v (as words in the free semigroup) are at most 4 (as happens with completely regular semigroups and inverse semigroups), then V is a variety of completely regular semigroups or a variety of inverse semigroups.

Someone trying to find a variety under the conditions of the theorem would find a few hundred relevant identities $u = v$ (with $u, v \in \{x, y, x', y'\}^+$) of which about 30% imply that within the corresponding variety of unary semigroups, $'$ is the unique regular pre-involution. Thus it is certainly possible to stumble upon one such identity, but, then it turns out that the corresponding variety lies within the variety of completely regular semigroups or the variety of inverse semigroups. On the other hand, a 'by hand' proof that all those examples lead to uninteresting varieties (if ever conjectured) is beyond any reasonable hopes of success. Fortunately, the situation now is totally different as we can use computational tools such as ProverX [49] that can handle in a couple of minutes a project that not long ago seemed totally unattainable (at least for one person alone).

This baby theorem is a particular case of the main 'monster' theorem (appearing in the next section). Since the proof of the latter subsumes the proof of the former, we will not provide an independent proof for the baby theorem.

3. The main theorem

The main result in this paper, which subsumes Theorem 2.1, is the following.

Theorem 3.1. *Let $H(u = v)$ be an Howie variety of unary semigroups defined by the following identities*

$$x = xx'x \qquad x'' = x \qquad u = v.$$

If $u, v \in \{x, y, x', y'\}^+$ and the lengths of u and v (as words in the free semigroup) are at most 4 (as happens with completely regular semigroups and inverse semigroups), then V is a variety of completely regular semigroups or a variety of inverse semigroups.

Proof. We produced 23639 identities which exhaust all possible identities fulfilling the conditions of the theorem. To prove the theorem it was necessary to prove 45198 lemmas. The proof that in $H(xyx'x' = y'x'y'y')$ the unary operation is uniquely defined is the longest proof filling more than 2000 pages.

We used a special package included in ProverX to generate words and identities and created a text file with all 23639 possible identities.

We then created a script using ProverX's scripting language (based on Python) to create a script to test if each identity defines a variety of completely regular semigroups or a variety of inverse semigroups. The test

returns false if we can find an identity that makes V not a variety of completely regular semigroups nor a variety of inverse semigroups or if it runs out of a predetermined time.

After a few tests and some fiddling, we adopted the strategy of running the script first with a maximum time of 1 second to eliminate most candidates and create a new file with the ones that didn't pass the test. We then ran it again on this new file with the remaining candidates but this time augmenting the timeout to 5 seconds. We proceeded this iterative process in the hope of eliminating all possible identities.

After about 30 hours we were left with only two identities (row 19 of the following table): $xxx'x' = y'y'yy$ and $xxx'x' = yyy'y'$. A that point the situation was the following:

		Howie	CR	Inverse	$x' = x$	Number of varieties
1	$U(u = v)$	no	no	no	no	5552
2	$U(u = v)$	no	undecided	undecided	undecided	5201
3	$H(u = v)$	yes	no	yes	no	174
4	$H(u = v)$	yes	yes	no	no	477
5	$H(u = v)$	yes	yes	no	yes	2442
6	$H(u = v)$	yes	yes	no	undecided	1
7	$H(u = v)$	yes	yes	yes	no	466
8	$H(u = v)$	yes	yes	yes	yes	2962
9	$H(u = v)$	yes	yes	undecided	yes	13
10	$H(u = v)$	yes	yes	undecided	undecided	6151
11	$H(u = v)$	yes	undecided	no	yes	3
12	$H(u = v)$	yes	undecided	yes	no	13
13	$H(u = v)$	yes	undecided	yes	undecided	2
14	$H(u = v)$	yes	undecided	undecided	yes	2
15	$H(u = v)$	undecided	no	yes	no	54
16	$H(u = v)$	undecided	yes	yes	undecided	10
17	$H(u = v)$	undecided	yes	undecided	undecided	7
18	$H(u = v)$	undecided	undecided	yes	undecided	107
19	$U(u = v)$	undecided	undecided	undecided	undecided	2

(Note that $x = x'$ immediately implies $xx' = x'x$; nevertheless we decided to keep the computer's output on rows 11 and 14.) Therefore, even if many questions were undecided, the decided ones were enough to settle all cases, except the two on row 19. We tested these two identities for more than 24 hours each and finally decided to handle them by hand as follows. Let c denote the constant defined by the identity $xxx'x' = yyy'y'$, that is, $c = xxx'x'$. Experiments with ProverX showed that up to order 16, semigroups satisfying the identity $xxx'x' = yyy'y'$ have c as an identity element. At order 17, a semigroup having c as a zero appeared. However, in this particular semigroup, the unary operation is still uniquely determined. We

let ProverX search for another semigroup of higher order having c as a zero. The first one it found has order 37. We then fed the multiplication table of this semigroup back to ProverX to have it search for distinct unary operations, and it quickly produced them. Therefore in the semigroups of this variety, the unary operation is not uniquely defined. This semigroup of order 37 has the following six generators (*rest* stands for all the points in which the transformation has not been defined):

$$a = \begin{pmatrix} \text{rest} & \{22,30\} & \{23,31\} & \{24,29\} & \{25,27\} & \{26,28\} & \{33,37\} \\ 1 & 2 & 16 & 4 & 8 & 18 & 35 \end{pmatrix}$$

$$b = \begin{pmatrix} \text{rest} & \{11,29\} & \{12,27\} & \{13,30\} & \{17,31\} & \{19,28\} & \{36,37\} \\ 1 & 3 & 6 & 5 & 14 & 20 & 34 \end{pmatrix}$$

$$c = \begin{pmatrix} \text{rest} & \{7,24\} & \{9,22\} & \{10,25\} & \{15,23\} & \{21,26\} & \{32,33\} \\ 1 & 7 & 9 & 10 & 15 & 21 & 32 \end{pmatrix}$$

$$d = \begin{pmatrix} \text{rest} & \{3,7\} & \{5,9\} & \{6,10\} & \{14,15\} & \{20,21\} & \{32,34\} \\ 1 & 7 & 9 & 10 & 15 & 21 & 32 \end{pmatrix}$$

$$e = \begin{pmatrix} \text{rest} & \{2,13\} & \{4,11\} & \{8,12\} & \{16,17\} & \{18,19\} & \{35,36\} \\ 1 & 22 & 24 & 25 & 23 & 26 & 33 \end{pmatrix}$$

$$f = \begin{pmatrix} \text{rest} & \{2,5\} & \{3,4\} & \{6,8\} & \{14,16\} & \{18,20\} & \{34,35\} \\ 1 & 22 & 24 & 25 & 23 & 26 & 33 \end{pmatrix}$$

This shows that the two varieties in class 19 of the table above are not Howie. This completes the proof of the theorem. □

4. The computations

The procedure to prove the main theorem will be outlined in the following subsections but first an introduction to ProverX is needed.

ProverX is a web application running on a server, its primary role is, given some axioms and goals, to find proofs or counter-examples. One of its greater strengths is it can be programmed (scripted) with the python language.

To access ProverX, just point any browser to `http://www.proverx.com` and enter as a guest.

The app behaves similarly to any IDE: the 'New' button (third icon on the left) opens a new editor. It is recommended to save the file first (5th icon) to get syntax highlighting based on the file extension (for example create_ident.py).

The 'Play' button (black triangle after Mace4) is used to run the script. In long interactions, it is recommended to check the 'run on REPL' option.

4.1. **Generating the words.**

For a word w in the free semigroup generated by a non-empty set X, denote by $|w|$ its length, denote by $C(w)$ its content, that is the variables in X that occur in w, and by $h(w)$, the *head* of w, denote the first letter occurring in w.

Consider the following sets:

(1) $W_x := \{w \in \{x, y, x', y'\}^+ \mid 1 \leq |w| \leq 4 \text{ and } C(w) \subseteq \{x, x'\}\}$;

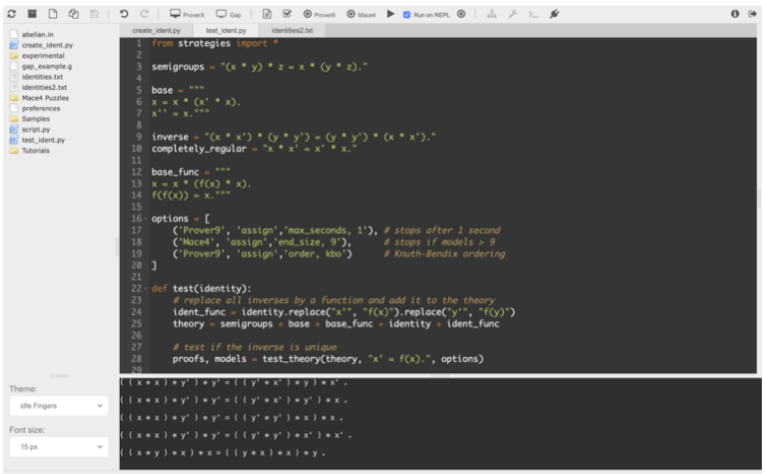


FIGURE 1. PROVERX screenshot.

(2) $W_{x,y} := \{w \in \{x, y, x', y'\}^+ \mid 1 \leq |w| \leq 4,\ C(w) \cap \{x, x'\} \neq \emptyset \neq C(w) \cap \{y, y'\}\}$;
(3) $W_{h(x)} := \{w \in W_x \cup W_{x,y} \mid 1 \leq |w| \leq 4 \text{ and } h(w) = x\}$.

We could take $(W_x \cup W_y \cup W_{x,y}) \times (W_x \cup W_y \cup W_{x,y})$ as our candidates. However some reductions of the search space are possible. Since $yx = w(x, y)$ is equivalent to $xy = w(y, x)$, and since, in our varieties, $x'u(x, y) = w(x, y)$ is equivalent to $xu(x', y) = w(x', y)$, then we can assume that in our identities the first letter of the word on the left is x. So rather than take the cartesian product of all words contained in $W_x \cup W_y \cup W_{x,y}$, we pick $I := W_{h(x)} \times (W_x \cup W_{x,y} \cup W_y)$.

But a further reduction is possible. As two of the identities in all our varieties are $x = xx'x$ and $x'' = x$, it follows that $x' = x'xx'$; thus the set I could be purged from the identities containing words of the form $uzz'zv$ and $uz'zz'v$, where $z \in \{x, y\}$ and $u, v \in \{x, x', y, y'\}^*$. After removing from I the identities of the form $u = u$, we ended up with a set containing 23639 identities. Of course this set could be simplified (for example, $xy' = yx$ is equivalent to $xy = y'x$ —just replace y with y' and use $y = y''$—), but a trade-off was necessary here. Increasing the number of simplifications implied increasing the complexity of the computer code and hence an increased chance of unwanted mistakes. Thus we worked with this I, a set with 23639 identities.

The above definitions were directly transcribed in PROVERX scripting language in a straightforward manner by using the Words_semigroup class from the words package:

```
from words import *

# create all words with varable x, y (with inverses) of size 4
u = Words_semigroups("x,y,x',y'", 4)

# filter all words with patterns zz'z or z'zz'
u.not_contains("x * x' ) * x ")
u.not_contains("x ) * x' ) * x ")
u.not_contains("x' * x ) * x' ")
u.not_contains("x' ) * x ) * x' ")
u.not_contains("y * y' ) * y ")
u.not_contains("y ) * y' ) * y ")
u.not_contains("y' * y ) * y' ")
u.not_contains("y' ) * y ) * y' ")

# create an identical set of words for v
v = u.clone()

# and from u take only words starting with x (ignoring parentheses)
u.startswith("x ")

# create file (23639 identities wil be created)
Identities(u, v).save("identities.txt")
```

4.2. **The search loop.** After creating the file with 23639 identities (which takes less than a second), we wrote the main search script. This script opens the identities file (identities.txt), and tests each one using the test function.

This function was quite easy to write by using the built-in function `is_variety` that ProverX provides in the strategies package.

The code and the comments below are self-explanatory:

```
from strategies import *

semigroups = "(x * y) * z = x * (y * z)."

base = """
x = x * (x' * x).
x'' = x."""

inverse = "(x * x') * (y * y') = (y * y') * (x * x')."
completely_regular = "x * x' = x' * x."

base_func = """
x = x * (f(x) * x).
```

```
f(f(x)) = x."""

options = [
    ('Prover9', 'assign','max_seconds, 1'), # stops after 1 second
    ('Mace4', 'assign','end_size, 9'),      # stops if models > 9
    ('Prover9', 'assign','order, kbo')      # Knuth-Bendix ordering
]

def test(identity):
    # replace all inverses by a function and add it to the theory
    ident_func = identity.replace("x'", "f(x)").replace("y'", "f(y)")
    theory = semigroups + base + base_func + identity + ident_func

    # test if the inverse is unique
    proofs, models = test_theory(theory, "x' = f(x).", options)

    # if not (model found) return
    if models: return True

    # if there is no model nor proof then we found our rare gem !
    if not proofs: return False

    # else test if it's a variety of inverse of completely regular semigroups
    return is_variety(identity,semigroups+base,[inverse,completely_regular],options)

with open("identities.txt", 'r') as identities:
    for identity in identities:
        if not test(identity):
            print identity
            out = open("identities2.txt",'a')
            out.write(identity)
            out.close()
```

After the first iteration (which can take more than 10 hours depending on the server), the file names in the main loop were changed (the input file became "identities2.txt" and the output became "identities3.txt") and the max_seconds was changed from 1 to 5 seconds (line 14). This takes about 30 hours and the proofs generate about one million pages.

We then ran the program again several times (increasing max_seconds each time) until only two identities were left which were dealt by hand.

5. Final remarks and problems

Before concluding this paper, we would like to make some observations.

5.1. **Correctness.** The correctness of these automatic proofs can be questioned (human proofs can be as well). However, each of the individual proofs were checked by an independent program (IVY) that was subjected to software verification [40]. Thus the degree of confidence in the correctness of these proofs is very high. There is no example of a false proof being approved by IVY. Therefore, if one false proof could be found in the companion website, that would be in itself a major discovery. We should note that IVY cannot check the counterexamples found by MACE4. In addition, IVY did not (and cannot) check the correctness of the reduction of the proof to special cases, although that these exhaust all possibilities is indeed clear.

In addition, the correctness issue is not restricted to automated reasoning. Mathematicians are now increasingly using symbolic computation systems that have much more sources of potential error/mistakes. Just for the sake of illustration, many theorems in primitive permutation groups have the general form of *All primitive groups have property P, except for a finite number of exceptions listed below.* Typically, these list of exceptions is found using an algebra system and involves the verification of millions of cases, using code written by the user (sometimes very large code relying on complex non-public algorithms), code written in the system (usually many different routines written by many different people with non-public algorithms), and libraries pre-compiled using the available literature. Each knot in this process adds a new risk of unintended error.

To summarize, the amount of code written and computations carried for this project is much smaller than other jobs in symbolic computation, the confidence in the correctness of PROVERX output checked by IVY is very high, and the algorithms here are few, very transparent, yielding a robust process.

5.2. **Computational issues.** Like many automated deduction programs, PROVER9 uses term orderings to decide how to orient equalities and to decide which terms in an equality are available for making inferences. There are two main choices for this, lexicographic path ordering (LPO) and Knuth-Bendix ordering (KBO). LPO orders terms based on symbol precedence, which in turn is selected by PROVER9. KBO first orders terms using a weight function and then uses symbol precedence to break ties. PROVER9 uses LPO as its default term ordering because in the experience of its developer, Bill McCune, LPO was a bit faster for many problems, especially in lattice theory.

The problem with LPO is that it can sometimes rewrite small terms with larger ones, eventually steering the search so that PROVER9 is unable to find a proof. Here is an example. In attempting to prove that the identity $xy'xx' = yxx'y'$ gives a completely regular variety, PROVER9 with the LPO term ordering very quickly derives the identity $x' = x^{47}$, where we write x^{47} as a shorthand for $x \cdots x$ with 47 occurrences of x. But now because of the term ordering, PROVER9 rewrites *all* instances of x' in its set of identities

available for inferences with x^{47}. This pushes the sizes of all available identities beyond the default limit (which is a total of 100 symbols per identity), so PROVER9 discards them. Thus PROVER9 runs out of identities with which to make inferences and quits.

By contrast, if we run PROVER9 on the same identity with the KBO term ordering, it derives that it implies that the variety is completely regular (in fact, the variety of semilattices) in less than a second.

The point here is that as with many automated deduction problems involving equality, this particular problem is very sensitive to the choice of settings, particularly term ordering. The discovery that for this project we should work with KBO rather than LPO was the real turning point that made everything possible.

5.3. **Corroborating evidence.** There is a sense in which our result is not entirely unexpected. Some grounds for the metamathematical conviction that there is very little beyond inverse semigroups and completely regular semigroups was already given in the work of Yeh [54] within the context of Hall's notion of e-varieties of regular semigroups. Yeh showed that an e-variety $\mathcal{V}$ of regular semigroups contains e-free objects if and only if $\mathcal{V}$ is an e-variety of locally inverse semigroups or an e-variety of E-solid semigroups. E-solid semigroups can be characterized as regular semigroups in which the idempotents generate a completely regular subsemigroup, and so both inverse and completely regular semigroups are E-solid. On the other hand, obviously, locally inverse semigroups and inverse semigroups are close-knit. Thus an informal interpretation of Yeh's result is that if we want to have some type of free object, we cannot really move very far beyond inverse semigroups and completely regular semigroups.

5.4. **Are there any nontrivial Howie varieties?** This paper exhausts the case of identities $u = v$ such that $|u|, |v| \leq 4$. The cases of word lengths above 4 involve more computations, many of which we have carried out. After millions of attempts, we have not managed to find a nontrivial Howie variety. However, even the case of length 5 identities has not been exhausted as several identities remained undecided. Therefore the main problem here is

Problem 5.1. *Are there any nontrivial Howie varieties?*

Another direction in which to expand our search is to remove the restriction on the identities to those where $u, v \in \{x, y, x', y'\}^+$, allowing the unary operation $'$ to be applied to arbitrary terms; also we should consider more variables than only x and y.

5.5. **Joins.** A different approach to seeking out new varieties is to look for a reasonable common generalization of inverse and completely regular semigroups. For example, both varieties satisfy the identity

$$xx'x'x = x'xxx', \tag{5.1}$$

so one might stipulate that a reasonable generalization should satisfy this. Similarly, both inverse and completely regular semigroups are E-solid, so one could adjoin e-identities for E-solidity to one's axioms.

The furthest one can go in this direction is to consider the *join* of the varieties of inverse and completely regular semigroups. The only paper known to the authors on this topic is that of Trotter [53], who gave an infinite system of identities axiomatizing the variety. It is conjectured that the join variety is not finitely based.

It is also worth mentioning that inverse and completely regular semigroups share another important feature which partly explains the richness of their respective theories: *every semigroup homomorphism of inverse (completely regular) semigroups is a homomorphism of* $\langle 2, 1\rangle$*-algebras.* This is actually *not* true in the join variety; details will appear elsewhere. Besides seeking nontrivial Howie varieties, one might also ask if there are new varieties in which this homomorphism property holds.

5.6. **The abundant world.** Inverse semigroups are semigroups such that for every x, there exists exactly one (necessarily) $\mathcal{D}$-related inverse. Completely regular semigroups are semigroups such that for every x, there exists exactly one $\mathcal{H}$-related inverse. (For Green's relations $\mathcal{D}$, $\mathcal{R}$, $\mathcal{L}$ and $\mathcal{H}$, see [32, Ch. 2].) Of course, one might guess that an easy way to find an Howie variety is to follow the same pattern and consider the semigroups such that for every x, there exists exactly one, say, $\mathcal{L}$-related inverse. However it is very easy to prove that if a regular element a has a unique inverse a' in its $\mathcal{L}$-class, then a' is, in fact, $\mathcal{H}$-related to a so that $a'a = aa'$ (see, for example, [45, Lemma 3.2]), and hence this class of semigroups is contained in the class of completely regular semigroups.

A different (and very successful) approach consists of defining new relations of Green's type and then emulating the traditional classes. For example, elements x, y in a semigroup S are said to be $\mathcal{R}^*$-related if there exists a semigroup T containing S and such that $x \mathcal{R} y$ in T. The relation $\mathcal{L}^*$ is dually defined.

The rationale for studying classes of semigroups defined by these relations goes beyond mere formalism, but for this paper, let us just proceed by analogy. Recall that a semigroup is regular if and only if each $\mathcal{R}$-class contains an idempotent if and only if each $\mathcal{L}$-class contains an idempotent. With respect to the starred relations, the analogous concepts are no longer equivalent. A semigroup is said to be *left abundant* if each $\mathcal{R}^*$-class contains an idempotent, and *right abundance* is defined dually. A semigroup is *abundant* if it is both left and right abundant [17].

Suppose that for each x in a left abundant semigroup, we make some arbitrary choice of idempotent x^+ in the $\mathcal{R}^*$-class of x. This defines a unary

semigroup which will turn out to satisfy the following axioms:

$$(xy)z = x(yz) \tag{A1}$$

$$x^+x^+ = x^+ \tag{A2}$$

$$x^+x = x \tag{A3}$$

$$yx = zx \quad \Rightarrow \quad yx^+ = zx^+ \tag{A4}$$

We note for later use that these axioms imply

$$xy = y \quad \Rightarrow \quad xy^+ = y^+ \,. \tag{A4'}$$

Indeed, if $xy = y = y^+y$ (by (A3)), then $xy^+ = y^+y^+ = y^+$ (by (A4) and (A2)).

Unary left abundant semigroups suffer from the same drawback as unary regular semigroups: fixing the unary operation arbitrarily means that in general, a left abundant subsemigroup may contain elements x such that x^+ is not in the subsemigroup

A semigroup is inverse if and only if each $\mathcal{R}$-class and each $\mathcal{L}$-class contain a unique idempotent. A semigroup is said to be $\mathcal{R}$-unipotent if each $\mathcal{R}$-class contains a unique idempotent; $\mathcal{L}$-unipotence is defined dually. Thus inverse semigroups are semigroups which are both $\mathcal{R}$-unipotent and $\mathcal{L}$-unipotent. As it turns out, $\mathcal{R}$-unipotence is not a strong enough condition to define the unary inverse operation uniquely, so one considers $\mathcal{R}$-unipotent semigroups from the e-variety perspective.

Back in the abundant world, a semigroup is *left amiable* if each $\mathcal{R}^*$-class contains exactly one idempotent and *right amiable* is defined dually. A semigroup is *amiable* if it is both left and right amiable [6]. The class of left amiable semigroups can be axiomatized as a quasi-variety of unary semigroups by (A1)–(A4) and

$$xx = x \;\&\; yy = y \;\&\; xy = y \;\&\; yx = x \;\Rightarrow x = y \,. \tag{A5}$$

Axiom (A5) says that no idempotent is $\mathcal{R}$-related to any other idempotent. Regular elements are $\mathcal{R}^*$-related if and only if they are $\mathcal{R}$-related, so (A5) is what gives the uniqueness of the idempotent x^+ in the $\mathcal{R}^*$-class of x.

We omit the easy proof that a semigroup is amiable if and only if it is abundant and every regular element has a unique inverse.

The preceding discussion defines left amiable semigroups semantically. We will call an algebra of type $\langle 2, 1\rangle$ satisfying (A1)–(A5) a unary left amiable semigroup. Thus we temporarily forget the semantic origin of left amiability and treat unary left amiable semigroups as forming a quasi-variety.

Proposition 5.2. *Let S be a semigroup. Then there exists at most one unary operation $x \to x^+$ defined on S such that all the quasi-identities (A1)–(A5) hold. In other words: in the class of unary left amiable semigroups, the unary operation is uniquely determined.*

Proof. Suppose a semigroup S has two unary operations, $x \mapsto x^+$ and $x \mapsto f(x)$, satisfying (A1)–(A5). Since $x^+x = x = f(x)x$ by (A3), we have

$x^+ = x^+x^+ = f(x)x^+$ by (A2) and (A4'), and $f(x) = f(x)f(x) = x^+f(x)$ by (A2) and (A4'). Now applying (A5), we get $x^+ = f(x)$ as claimed. □

Inverse semigroups are characterized or defined as regular semigroups with commuting idempotents. In the abundant world, a semigroup is *left adequate* if it is left abundant and has commuting idempotents. *Right adequate* is defined dually, and a semigroup which is both left and right adequate is simply said to be *adequate* [16]. Unary left adequate semigroups form a quasi-variety defined by (A1)–(A4) and

$$x^+y^+ = y^+x^+ . \tag{A5'}$$

A semigroup is (left) adequate if and only if it is (left) abundant and its set of regular elements is an inverse subsemigroup [16]. Thus every (left) adequate semigroup is (left) amiable. Unlike the analogous situation in the regular world, the converse is not true [3]. Adequate semigroups can be characterized among amiable semigroups as avoiding certain forbidden subsemigroups [6].

Again, the following corollary is obvious from the semantic definitions of these various classes, but here we think of them formally as quasi-varieties of algebras.

Corollary 5.3. *Let $(S, \times)$ be a semigroup. Then, on S, it is possible to define at most one unary operation $x \to x^+$ such that $(S, \times, ^+)$ satisfies all the quasi-identities of (left) amiable or (left) adequate semigroups*

The abundant analogue of complete regularity is called superabundance [17], that is, a semigroup is *superabundant* if it has an idempotent in every $\mathcal{H}^*$-class. As with $\mathcal{H}$-classes, if an $\mathcal{H}^*$-class contains an idempotent, it contains only one, so this again defines a unique unary operation semantically. Once more we consider this from the quasi-variety point of view. Unary superabundant semigroups are defined by (A1)–(A4) and the duals

$$xx^+ = x \tag{A3*}$$

$$xy = xz \quad \Rightarrow \quad x^+y = x^+z . \tag{A4*}$$

Besides (A4′), we also have

$$yx = y \quad \Rightarrow \quad y^+x = y^+ . \tag{A4*′}$$

Proposition 5.4. *Let $(S, \times)$ be a semigroup. Then, on S, it is possible to define at most one unary operation $x \to x^+$ such that $(S, \times, ^+)$ satisfies all the quasi-identities of a unary superabundant semigroup*

Proof. The proof is similar to that of Proposition 5.2. Suppose $x \mapsto x^+$ and $x \mapsto f(x)$ are two such operations. Since $x^+x = x = f(x)x$ by (A3), we have $x^+ = x^+x^+ = f(x)x^+$ by (A2) and (A4'). Dually, we have $x^+ = x^+f(x)$ by (A3*), (A2) and (A4*'). On the other hand, we can reverse the roles of $f(x)$ and x^+ in these calculations. Thus $x^+ = f(x)$. □

Observing that the proofs of Propositions 5.2 and 5.4 depend only on (A4′) and (A4*′), we note we can generalize still further. Define $x \mathrel{\widetilde{\mathcal{R}}} y$ if and only if x and y share the same sets of left identities, that is, for all $e = e^2$, $ex = x \Leftrightarrow ey = y$. Now the notions of (right) *weak abundance* (sometimes *semiabundance*), *weak amiability* and *weak adequacy* are defined in the obvious ways. From the quasi-variety point of view, one simply uses (A4′) or (A4*′) instead of (A4) or (A4*). As the proofs above show, the expected uniqueness results follow. Similar remarks apply if we start with the relation $\widetilde{\mathcal{R}}_E$, which is defined like $\widetilde{\mathcal{R}}$, but with respect to a distinguished subset E of the idempotents rather than the set of all idempotents.

5.7. **Generalized restriction semigroups.** In our discussion of uniquely determined unary operations in the (weakly) abundant world of the preceding subsection, we (necessarily) moved away from varieties to quasi-varieties. To bring things back to varieties, and thus closer to the spirit of the rest of the paper, we will briefly discuss here one further generalization. Again, this can be interpreted in terms of generalized Green's relations, but this time we just proceed formally.

A generalized left restriction semigroup [23] is a unary semigroup satisfying the following identities:

(1) $x(yz) = (xy)z$ (2) $x^+x = x$ (3) $x^+(xy)^+ = (xy)^+$
(4) $(x^+y^+)^+ = x^+y^+$ (5) $x^+(y^+x^+) = x^+y^+$ (6) $x^+x^+ = x^+$
(7) $(x^+)^+ = x^+$.

Note that (6) and (7) are dependent. To prove (7), observe that

$$x^{++} = (x^+)^+ = (x^{++}x^+)^+ = x^{++}x^+,$$

where the second equality follows from (2) and the last from (4). By $x^{++} = x^{++}x^+$ and (2), we get $x^{++} = x^{++}x^+ = x^+$, as desired. Regarding (6),

$$x^+ = x^{++}x^+ = x^+x^+,$$

where the first equality follows from (2) and the second from (7).

Note also that axioms (4), (5) and (6) imply that in a generalized left restriction semigroup S, the set $U_+ = \{x^+ \mid x \in S\}$ is a left regular band.

A right generalized restriction semigroup is a unary semigroup satisfying the dual identities:

(11) $x(yz) = (xy)z$ (12) $xx^* = x$ (13) $(xy)^*y^* = (xy)^*$
(14) $(x^*y^*)^* = x^*y^*$ (15) $(x^*y^*)^*x^* = y^*x^*$ (16) $x^*x^* = x^*$
(17) $(x^*)^* = x^*$.

By symmetry, (16) and (17) are redundant, and in a generalized right restriction semigroup S, the set $U_* = \{x^* \mid x \in S\}$ is a right regular band.

An algebra of type $\langle 2, 1, 1\rangle$ is said to be a *generalized restriction semigroup* [22] if it is generalized left restriction for one unary operation, generalized right restriction for the other and satisfies two connecting identities

$$(18)\quad (x^+)^* = x^+ \qquad \text{and} \qquad (19)\quad (x^*)^+ = x^*.$$

In this case, we have $U_+ = U_*$ and this set of idempotents, which we denote simply by U, is a semilattice. In particular, the identity

$$x^+y^* = y^*x^+ \quad (20)$$

also holds.

Theorem 5.5. *Neither generalized left nor generalized right restriction semigroups have their unary operations uniquely defined. In a generalized restriction semigroup, each unary operation is uniquely defined once the other is fixed.*

Proof. To show that left generalized restriction semigroups do not have a uniquely defined unary operation, consider a 2-element right zero semigroup $S = \{a, b\}$ with two constant unary operations: $f(x) = a$ and $g(x) = b$. Then $(S, \times, f)$ and $(S, \times, g)$ are generalized left restriction semigroups with the same semigroup reduct. The dual of this example handles generalized right restriction semigroups.

Suppose $(S, \cdot, ^+, ^*)$ is a generalized restriction semigroup with $(S, \cdot, ^+)$ being generalized left restriction and $(S, \cdot, ^*)$ being generalized right restriction. Suppose further that $(S, \cdot, ^\circ, ^*)$ is another generalized restriction semigroup structure on S sharing the same generalized right restriction unary operation. Then we also have $U = \{x^\circ \mid x \in S\}$ and so $x^\circ y^* = y^* x^\circ$ and $x^\circ y^+ = y^+ x^\circ$ for all $x, y \in S$. In addition, each of $^+$, * and $^\circ$ act as the identity mapping when restricted to U. In particular, $(x^+)^\circ = x^+$ for all $x \in S$.

Now $x^\circ = (x^+x)^\circ = (x^+)^\circ(x^+x)^\circ = x^+x^\circ$, using (2) in the first equality, (3) (applied to $^\circ$) in the second, and (2) together with the last observation of the preceding paragraph in the third. By symmetry, we must also have $x^+ = x^\circ x^+$. Since $x^+x^\circ = x^\circ x^+$, we obtain $x^+ = x^\circ$ for all $x \in S$. □

For many more classes of semigroups with varied signatures, see [3, 6, 10, 11, 12, 15, 21, 22, 24, 25, 26, 27, 34, 51] and the references therein.

References

[1] J. Araújo and J. Duarte, XPlain. `http://www.ciul.ul.pt/~mjoao/xplain.zip`

[2] J. Araújo, P.J. Cameron, J.D. Mitchell and M. Neunhöffer, The classification of normalizing groups, *J. Algebra*, **373** (2013), 481–490.

[3] J. Araújo and M. Kinyon, On a problem of M. Kambites regarding abundant semigroups, *Comm. Algebra* **40** (2012), 4439–4447

[4] J. Araújo and M. Kinyon, An elegant 3-basis for inverse semigroups, *Semigroup Forum* **82** (2011), 319–323.

[5] J. Araújo and M. Kinyon, Axioms for unary semigroups via division operations, *Comm. Algebra* **40** (2012), 719–737.

[6] J. Araújo, M. Kinyon and A. Malheiro, A characterization of adequate semigroups by forbidden subsemigroups, *Proc. Roy. Soc. Edinburgh Sect. A* **143** (2013), 1115–1122.

[7] J. Araújo, M. Kinyon and R. Padmanabhan. A 2-base for inverse semigroups, `arxiv.org/abs/1210.3285`

[8] J. Araújo and W. McCune, Computer solutions of problems in inverse semigroups, *Comm. Algebra* **38** (2010), 1104–1121.
[9] K.J. Arrow, A difficulty in the concept of social welfare. *J. Political Economy* **58:4** (1950), 328-3466.
[10] K. Auinger, G.M.S. Gomes, V.A.R. Gould and B. Steinberg, An application of a theorem of Ash to finite covers, *Studia Logica* **78** (2004), 45–57.
[11] M. Branco, G. Gomes and V.A.R. Gould. Extensions and covers for semigroups whose idempotents form a left regular band, *Semigroup Forum* **81** (2010), 51–70.
[12] M. Branco, G. Gomes and V.A.R. Gould, Left adequate and left Ehresmann monoids, *Int. J. Algebra Comp.* **21** (2011), 1259–1284.
[13] R. H. Bruck, *A Survey of Binary Systems*, Springer-Verlag, 1971.
[14] A.H. Clifford and G.B. Preston, *The algebraic theory of semigroups*, Vol. I. Mathematical Surveys, No. 7, American Mathematical Society, Providence, R.I. 1961.
[15] C. Cornock and V.A.R. Gould, Proper restriction semigroups and partial actions, *J. Pure Applied Algebra* **216** (2012), 935–949.
[16] J.B. Fountain, Free right h-adequate semigroups, *Semigroups, theory and applications (Oberwolfach, 1986)*, 97–120, Lecture Notes in Math. **1320**, Springer, Berlin, 1988.
[17] J.B. Fountain, Abundant semigroups. *Proc. London Math. Soc.* **44** (1982), 103–129.
[18] J.B. Fountain, G. Gomes and V.A.R. Gould, The free ample monoid, *Int. J. Algebra Comp.* **19** (2009), 527–554.
[19] Fountainfest, "Semigroups, categories and automata," A conference celebrating John Fountain's 65th birthday and his mathematical achievements, 12–14 October 2006, University of York, `http://maths.york.ac.uk/www/Fountainfest`
[20] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008.
[21] G. Gomes and V.A.R. Gould, Left adequate and left Ehresmann monoids II, *J. Algebra* **348** (2011), 171–195.
[22] V.A.R. Gould, Restriction and Ehresmann semigroups, *Proceedings of the International Conference on Algebra 2010*, 265–288, Advances in Algebraic Structures, World Sci. Publ., Hackenack, NJ, 2012.
[23] V.A.R. Gould, Notes on restriction semigroups and related structures, `http://www-users.york.ac.uk/~varg1/restriction.pdf`
[24] V.A.R. Gould and C.Hollings, Partial actions of inverse and weakly left E-ample semigroups, *J. Australian Math. Soc.* **86** (2009), 355–377.
[25] V.A.R. Gould and C.Hollings, Restriction semigroups and inductive constellations, *Comm. Algebra* **38** (2010), 261–287.
[26] V.A.R. Gould and C. Hollings, Actions and partial actions of inductive constellations, *Semigroup Forum*, **82** (2011), 35-60.
[27] V.A.R. Gould and M. Kambites, Faithful functors from cancellative categories to cancellative monoids, with an application to abundant semigroups, *Int. J. Algebra Comp.* **15** (2005), 683–698.
[28] T.E. Hall, A concept of variety for regular semigroups, *Monash Conference on Semigroup Theory (Melbourne, 1990)*, 101–115, World Sci. Publ., River Edge, NJ, 1991.
[29] L. Henkin, J. Monk and A. Tarski, Cylindric algebras, Part I, Studies in Logic and the Foundations of Mathematics, Vol. **64**, North-Holland Publishing Co., Amsterdam-London, 1971.
[30] P. Higgins and M. Jackson, Algebras defined by equations, `arXiv:1810.13012`
[31] T. Hillenbrand, *Waldmeister*, `http://www.mpi-inf.mpg.de/~hillen/waldmeister/`
[32] J. M. Howie, *Fundamentals of Semigroup Theory*, Oxford Science Publications, Oxford, (1995).
[33] J.M. Howie and J. L. Selfridge, A semigroup embedding problem and an arithmetical function, *Math. Proc. Cambridge Philos. Soc.* **109:2** (1991), 277–286

[34] M. Jackson and T. Stokes, Modal restriction semigroups: towards an algebra of functions, *Internat. J. Algebra Comput.* **21** (2011), 1053–1095.
[35] J. Kad'ourek and L. Polák, On the word problem for free completely regular semigroups, *Semigroup Forum* **34** (1986), 127–138.
[36] G. Kolata, Computer Math Proof Shows Reasoning Power, *The New York Times*, December 10, 1996, `www.nytimes.com/library/cyber/week/1210math.html`
[37] M. V. Lawson, *Inverse semigroups. The theory of partial symmetries*, World Sci. Publ. Co., Inc., River Edge, NJ, 1998.
[38] S. Lipscomb, *Symmetric inverse semigroups*, Mathematical Surveys and Monographs, **46**. American Mathematical Society, Providence, RI, 1996.
[39] W. McCune, *Prover9 and Mace4*, `https://www.cs.unm.edu/~mccune/prover9/`.
[40] W. McCune and O. Shumsky, Ivy: A preprocessor and proof checker for first-order logic, Ch. 16, in M. Kaufmann, P. Manolios and J Moore (eds.), *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic, 2000.
[41] G. F. McNulty, A field guide to equational logic, *J. Symbolic Comput.* **14** (1992), 371–397.
[42] T. E. Nordahl and H. E. Scheiblich, *Regular ?-semigroups*, Semigroup Forum 16 (1978), no. 3, 369–377.
[43] A. L. T. Paterson, *Groupoids, inverse semigroups, and their operator algebras*, Progress in Mathematics **170**, Birkhäuser Boston, Inc., Boston, MA, 1999.
[44] M. Petrich, *Inverse semigroups*, Pure and Applied Mathematics (New York). A Wiley-Interscience Publication. John Wiley & Sons, Inc., New York, 1984.
[45] M. Petrich, Onesided inverses for semigroups. *Acta Math. Univ. Comenianae*, Vol. LXXV, **1** (2006), 1–19.
[46] M. Petrich and N. Reilly, *Completely regular semigroups*. Canadian Mathematical Society Series of Monographs and Advanced Texts, **23**. A Wiley-Interscience Publication. John Wiley & Sons, Inc., New York, 1999.
[47] D. Pigozzi, Equational logic and equational theories of algebras, Mimeographed at Iowa State University, 1970, and reissued at Purdue University, 1975.
[48] G.B. Preston, Products of inverse semigroups, *Collect. Math.* **46** (1995), 151–157.
[49] Y. Robert, *ProverX*, `http://proverx.com/login.php`
[50] B. M. Schein, On the theory of generalized groups and generalized heaps, *Amer. Math. Soc. Transl.*, (2) **113**, (1979), 89–122.
[51] T. Stokes, Comparison semigroups and algebras of transformations, *Semigroup Forum* **81** no. 2, (2010), 325–334.
[52] W. Taylor, Equational logic, *Houston J. Math.* **37** (1979) (survey), 1–83.
[53] P. G. Trotter, Unary semigroup joins of varieties of inverse and completely regular semigroups, *Monash Conference on Semigroup Theory (Melbourne, 1990)*, 296–305, World Sci. Publ., River Edge, NJ, 1991.
[54] Y. T. Yeh, The existence of e-free objects in e-varieties of regular semigroups, *Int. J. Algebra Comput.* **2** (1992), 471–484.

(Araújo) Departamento de Matemática, Centro de Matemática e Aplicações (CMA), Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829–516 Caparica, Portugal, and CEMAT-CIÊNCIAS Universidade de Lisboa, Portugal
Email address: jj.araujo@fct.unl.pt

(Kinyon) Department of Mathematics, University of Denver, 2360 S Gaylord St, Denver, Colorado 80208 USA
Email address: mkinyon@du.edu

(Robert) Universidade Aberta, R. Escola Politécnica, 147, 1269-001 Lisboa, Portugal
Email address: yvrobert@gmail.com

6 - Conclusion

Before going any further, it is important to acknowledge that, since this is a PhD project, the goal was to create a functional prototype, not a commercial product ready to be released to the public.

So, at this point, it is important for us to reflect upon the results and ask ourselves some fundamental questions: was this goal achieved, or did it fail? What are the limitations of the project? What could have been done better? And what are the plans for the future?

ProverX was successfully implemented and installed on a small VPS (virtual private server) running linux ubuntu 16.04 more than a year ago. Although this is a small and inexpensive entry level VPS, ProverX ran surprisingly fast (faster than the development machine) and is still working flawlessly with results that exceeded our expectations.

This implementation can be found at http://www.proverx.com

6.1 - Success cases

- ProverX was successfully used by the author to present a lecture on the use of software tools to teach propositional logic: "Lógica no Ensino" at "Encontro Nacional da Sociedade Portuguesa de Matemática"[46] held at *Instituto Politécnico de Bragança* in July 2018.
- As stated previously in 1.2.1, it was used to teach discrete mathematics in two classrooms with more than 100 students by professor J. Araújo at FCT/UNL with no slowdown and perfect results. The students feedback was very enthusiastic with some of them readily volunteering to create new packages for the project.
- ProverX was also used successfully to prove a theorem in semigroup theory as documented in the paper ***"Varieties of Regular Semigroups with Uniquely Defined Inversions"*** by João

[46] http://www.enspm2018.ipb.pt/PT_index.html

Araújo, Michael Kinyon and Yves Robert. This article shows that this kind of result that can only be obtained through the use of an algorithm, are reduced to a few lines of code in ProverX thus speeding immensely such research projects. A copy of the paper can be found on chapter 4 (section 4.4: A real-word example).

6.2 - Limitations

6.2.1 - Security

We consider the weakest point of this project to be security. There are at least two main flaws:

1) Since users have access to whole Python library, it is easy to write a script that can access the file system. Although this script can only access the files in the web directory (and not the entire server), a malicious user could view/change other users files.
2) The registered user database is implemented only for demonstrations purposes and it is simulated by a PHP dictionary of usernames / passwords.

These weaknesses are not of great concern since, for now, the program is used only by guests with a temporary file system that is erased after 24 hours. That said, it is obviously of primary importance to address the security issues in the next version.

6.2.2 - ProverX

1) When the project started, it was developed on a MacOS system with Python 2.7 already installed. And since the author of the project worked with version 2 for years, it seemed natural to use Python 2. Unfortunately, by the time the project was completed, it was

announced that Python 2 will be deprecated in 2020. For the moment, this not a real problem, but it could become a major concern in a very near future.

2) The way ProverX interacts with prover9/mace4 is by including the source of both programs in the main executable and by forking and passing temporary files as input and output. This solution has been working properly even with large problems that open and close thousands of files (see paper in 4.4). Nevertheless, it is not a very efficient solution and we are concerned that for massive parallel experiments, a bottleneck with the filesystem could be found rapidly.

6.2.3 - GUI

There are also some minor annoyances with the GUI:

1) The user must be careful not to refresh the browser or click the button to go back because this will mean losing any unsaved work.
2) Only one REPL window can be opened. It could be useful to have, for instance, a GAP session running side by side with a ProverX session in REPL mode.
3) The font size of the REPL is a little too small and cannot be changed.

6.3 - Future work

This project can potentially grow infinitely, but here are some areas that must be addressed in a very near future:

6.3.1 - Security

As the first concern is security, it would be important to add some container technology with an on-demand server isolating each user's filesystem. For now, we are contemplating the use of Docker[47] and some kind of on-demand server technology like Microsoft Azure[48] or Amazon Cloud Services[49].

This approach will bring two benefits: on one hand it would solve the security issue by maintaining each file system isolated from other users, on the other hand it would bring a high scalability to the project.

6.3.2 - ProverX

1) Upgrade the scripting engine to Python 3.
2) Rewrite the original prover9/mace4 source code so the functions could accept string instead of files. This will bring the benefit of having all the interaction between main program and the scripting engine all done in memory and not with temporary files. Some parts of the ProverX Scripting Library will have to be rewritten as well.

6.3.3 - GUI

1) There is a need to create a more robust windowing GUI system. The javascript framework that handle panes should be replaced (or rewritten). This also should allow for a more versatile arrangement of panes.
2) Modify or rewrite the **`ttyd`** module so that more than one REPL pane could be opened at the same time. Having control over the source code, will allow to tweak certain issues like having a bigger font size etc.

[47] https://www.docker.com/
[48] https://azure.microsoft.com
[49] https://aws.amazon.com

3) Users should be able to configure and assign buttons (or menu options) to execute automatically their own Python functions.
4) Update the File Manager library to the newer version which now allows file drag and drop.

6.3.4 - Outside World

1) Create an easy mechanism for other user to create packages (like natural language processors, proof humanizers, etc.). Although new packages need to be installed by the system administrators, some guidelines should be available as some template for the packages documentation.
2) Add other automated theorem proving engines like Vampire, Waldmeister and be able to seamlessly use one or another. For this purpose, a package is being written that translates the Prover9 syntax to other languages like TPTP.

6.4 – Dependencies

Nowadays, it is almost impossible to create a software project without depending heavily on other people's projects. All my gratitude goes to the generous and bright persons who have created these wonderful tools:

- Prover9, by Professor William McCune, https://www.cs.unm.edu/~mccune/prover9/.
- Ace, The High Perfomance Code Editor for the Web, https://ace.c9.io.
- Split.js, by Nathan Cahill, https://split.js.org/.
- jQuery File Tree, https://github.com/jqueryfiletree/jqueryfiletree.
- Nice Select, by Hernán Sartorio, http://hernansartorio.com/jquery-nice-select/.
- Roxy Fileman, http://www.roxyfileman.com/.
- Font Awesome Icons, https://fontawesome.com/v4.7.0/icons/.
- ImageViewer, by Sudhanshu Yadav, http://ignitersworld.com/lab/imageViewer.html.
- CSS Element Queries, by Marc J. Schmidt, http://marcj.github.io/css-element-queries/.
- Showdown, a Markdown to HTML converter, http://showdownjs.com.
- TOC extension for Showdown, by Jan Löbel, https://github.com/JanLoebel/showdown-toc.
- Highlight.js, Syntax highlighting for the Web, https://highlightjs.org/.
- Github Markdown CSS, by Sorhus, https://github.com/sindresorhus/github-markdown-css.
- ttyd – share your terminal over the web, https://github.com/tsl0922/ttyd

Appendix A - The User Interface

In this chapter you will learn how to interact with the user interface of ProverX.

Main Window

ProverX is a single page web app, so the first thing to do is to get acquainted with the main window.

The main window is divided in several areas and panes. Each pane can be resized vertical or horizontally. Furthermore, some of these panes can be collapsed or revealed by clicking an icon on the toolbar (show/hide icons on the right of the toolbar: , and).

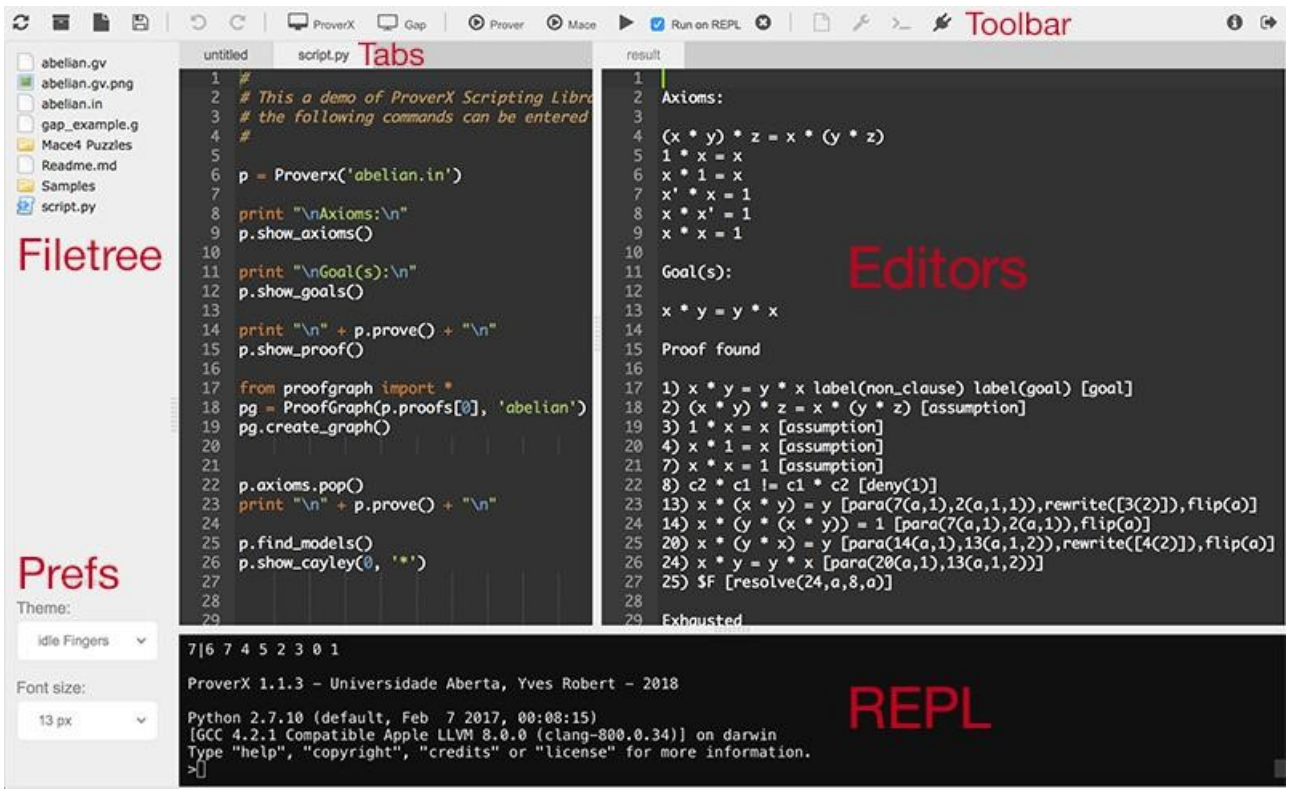


Figure 10- The Graphic User Interface

File tree

This is where you can quickly select files.

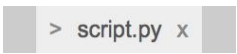
Notice that there is a File Manager (click the  icon) for more sophisticated actions like renaming, deleting, moving and uploading files to the server.

The File Tree is for quickly opening files and changing the active folder (just click on any folder; the name will turn bold to indicate that it is now the active folder).

The tree hierarchy is foldable (just click on any folder to reveal/hide the content).

Sometimes, a file may not appear on the File Tree (especially if it was created by a running script). If this is the case, you can refresh it manually by clicking this icon: 

Editing files

You can have several editor windows opened at the same time and switch between them using the tabs: 

To close a tab, click on the “x” sign.

To move a tab to the right, click on the “>” sign (or "<" to move it to the left).

Be aware that some actions will create new editor windows automatically. The obvious case is when you run a Prover9/Mace4 or a script (without the REPL option checked): the result will be inserted in a new editor window (on the right if the source was on the left and vice-versa).

As the result is editable, if you try to close it, you will receive a warning asking if you want to save first. Of course, after a few runs, your workplace will begin to get cluttered with result windows. You can speed up the process by clicking the "Close all Results" icon: 

Another case is when you want to check the syntax of your Prover9 input file (or just see how the preprocessor expanded your code) by clicking the "Syntax check" icon: 

In this case, the new editor will be a file with the same name but with the extension ".tmp". Be careful not to mistake the tmp files with the original as sometimes they are absolutely identical (if the preprocessor had nothing to change and the syntax is correct). This kind of files are not editable.

Sometimes it is more practical to create a new file by changing an existing one.

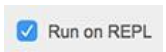
You can quickly duplicate a file by clicking the "Duplicate" icon: 

Running code

You can run two types of code: Prover9/Mace4 files or Python scripts.

and 

will run Prover9 or Mace4.

 will run the Python interpreter.The result of any of these operations will appear in a new editor window or in the REPL (see below) depending on the status of this checkmark: 

The REPL (or command line)

This pane works as a command line on a terminal logged to the server.

Here you can run GAP session: 

You can also open a ProverX session: 

A ProverX session is very similar to using a Python interpreter. Here you can, for example, open files, find proofs, models, and interact with these by executing Python commands. see the scripting section for more details.

Remember to disconnect the REPL if you are not using it to save bandwidth and CPU time. You can do this by clicking: 

Remember that you will not disconnect the REPL by just hiding it 

Preferences

Here, you can change the theme and font size of the editors.

The toolbar icons:

Most of the actions are performed by clicking on the toolbar buttons:

Refresh Filetree

 Refreshes the file tree. This can be useful when running scripts that create files (only necessary if the script run on the REPL, otherwise the file tree is refreshed automatically).

Open File manager

 This is where you can upload and download and preview files and also create, rename and delete folders or files.

New File

 Creates a new blank file. This file will be created on the active side of the window.

Duplicate File

 Creates a copy of the current file.

Save File

 Saves the active editor content.

Undo/Redo

 Undo or redo the last action.

Opens ProverX in REPL

Opens the REPL pane and starts a ProverX session.

Opens GAP in REPL

Opens the REPL pane and starts a GAP session.

New Blank Prover9 File

 Creates a new template file for use with Prover9 or Mace4.

Syntax Check

 Checks prover syntax and runs the preprocessor. The preprocessed file is saved with a .tmp extension for checking.

Run Prover9

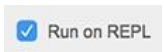
Runs Prover9 with the code on the active editor as an input file.

Run Mace4

Runs Prover9 with the code on the active editor as an input file.

Run Script

 Runs the script on the active editor. This script needs to be a Python or GAP file; you need to save it first with the right extension (.g or .py).

Note: The results of the above actions can be shown on the REPL 

or

(if unchecked) on the opposite editor pane (right if the source is on the left and vice versa).

Close all results

 This button closes all tabs called "result". This is can be very useful as every time you run a script, the results will open in a new tab.

Show / Hide File Tree

 Shows or hides the file tree.

Show / Hide Preferences

Shows or hides the preferences pane.

Show / Hide File REPL

Shows or hides the REPL.

Disconnect the REPL

This will close the remote connection with the session running on the server. It is recommended that you do this if not using the REPL for saving bandwidth and cpu time on the server.

Documentation

Opens a new window with online documentation.

Logout

Log out from the server. This is the recommended way to stop using the program as it will ask you to save unsaved files and will also save your preferences (if you have an account).

Appendix B - Quick Start

Start by directing your web browser to: http://www.proverx.com

You will be presented with this screen:

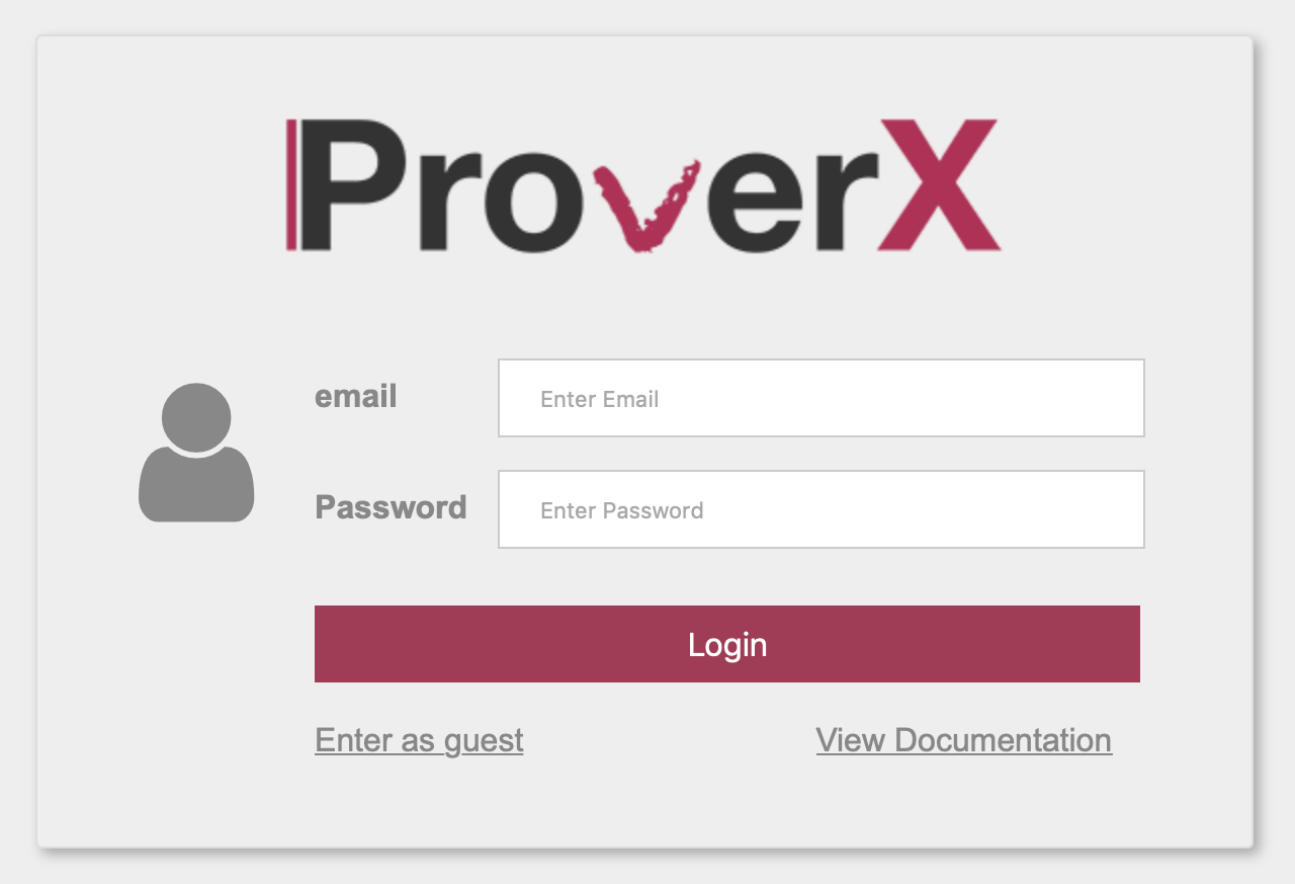
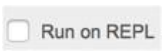


Figure 11- ProverX Login

If you don't have an account, just choose the option "Enter as guest".

Now, here are a few things you can try to get a quick overview of ProverX possibilities:

- Click the file "abelian.in" on the file tree; the file will appear in a pane (on the left). Now, press the "Run with Prover9" 

button on the toolbar (make sure that "Run with REPL is unchecked) 

. A new pane on the right should appear with the proof found by Prover9.

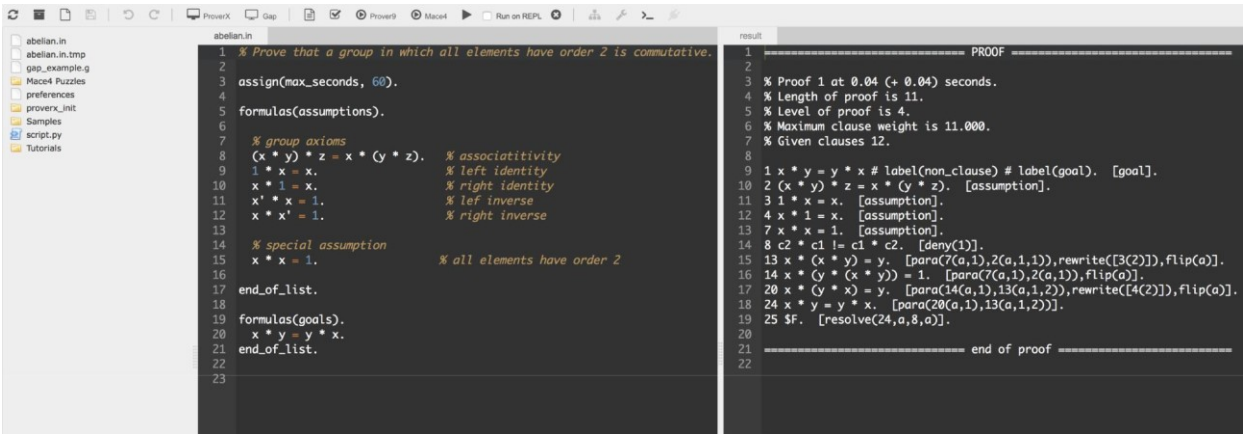


Figure 12- Prover9 example

- Now, comment line 15 (the line with: x * x = 1) by writing a % at the beginning. Press the "Run with Prover9" button again. You should now see the text: "No proof found!" (This is because we are trying to prove that all groups are commutative which is obviously false). Notice that the active file (in this case the file with the axioms) is automatically saved when you press the "Run with Prover9" button. This is true for the other buttons as well (Mace4, run a script, etc.)

- So, if there is no proof, let's try to find a counter example (a model that doesn't fit the theory). Press the "Run with Mace4" button 

and you should get a model! (This is the smallest non-commutative group).

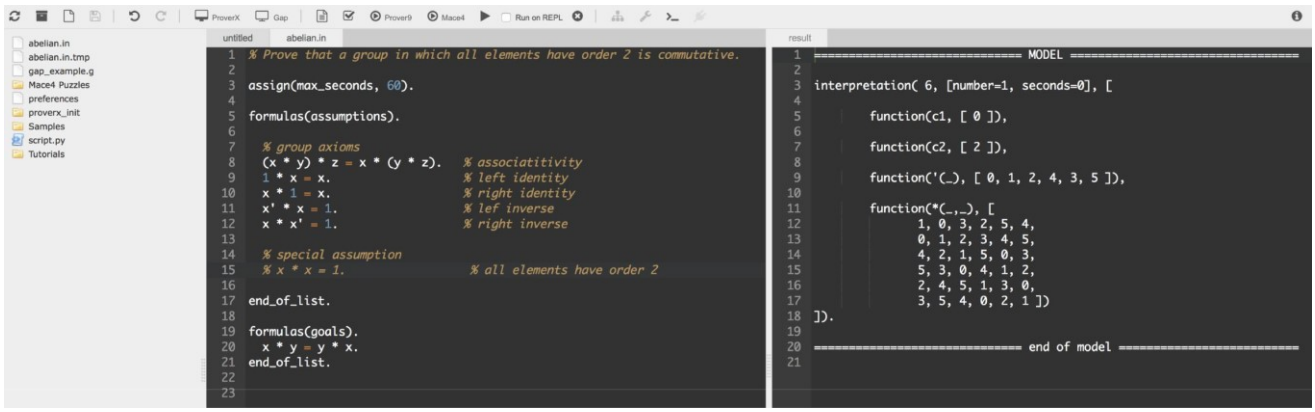


Figure 13 - Mace4 example

- Uncomment line 15 on file "abelian.in" and save it (you have to save it because next the active file will be "script.py" and not "abelian.in"). Now open thefile "script.py" (and maybe try to understand what it does). Run the script by pressing this button:

- The script has created new files! You should see on the file tree two new files: "abelian.gv" and "abelian.gv.png" Click the file "abelian.gv.png". This a graph representation of the proof.

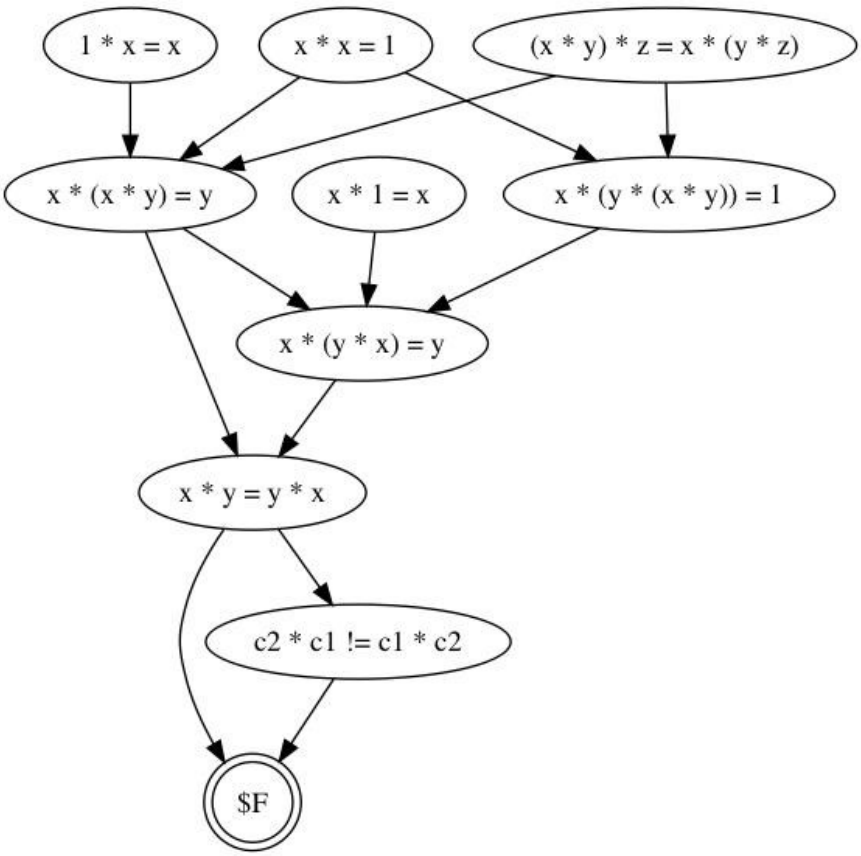


Figure 14- Graphic representation of the proof

- Try some of the above, but this time with the "Run with REPL" option checked.

- Now open the file "gap_example.g" and press the "Run Script" button. Wait a few seconds for GAP to start in the background and you will have the results of this script on the right.

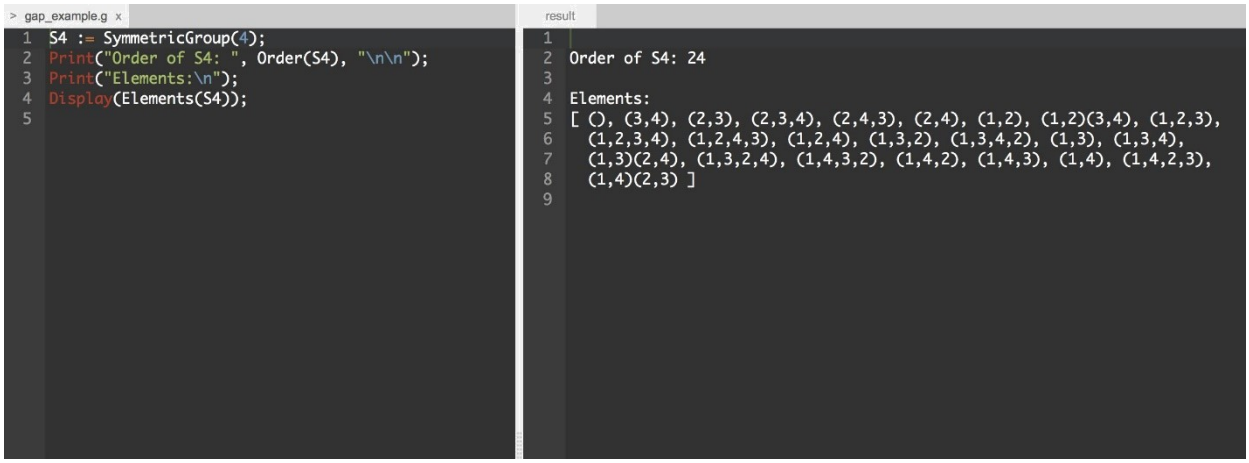


Figure 15- GAP example

Appendix C - ProverX Scripting Library

The ProverX Scripting Library is a collection of modules and classes written in Python that are automatically imported in ProverX. They allow the user to interact easily with Prover9 and Mace4, launch simultaneous jobs, work with proofs, models, hints, etc.

Class Proverx

This is the main class of ProverX. It opens files with Prover9 syntax or the new advanced syntax (see Preprocessor Class) and tries to find proofs or models.

Example of usage:

```
p = Proverx('abelian.in')

print p.axioms

print p.goals

print "Proofs found:"
print p.find_proofs()

print p.proofs

p.axioms.pop()

print p.find_proofs()  #should print 0

p.find_models()
# This is the smallest non commutative group
# Just change max_models option to find more (ex. 20)
p.set_option('Mace4','assign','max_models, 20')

p.find_models()
print p.models

# put the last axiom back (notice that the dot is optional)
p.axioms.add('x * x = 1')
p.find_proofs()

# Show the hints
print p.hints
```

Attributes:

proofs

Proofs instance containing all proofs found.

models

Models instance containing all models found.

axioms

Lines instance with all assumptions.

goals

Lines instance with the goals.

hints

Lines instance with hints found in all proofs.

givens

Lines instance with all givens found in all proofs.

weights

Lines instance with weights list from input.

kbo_weights

Lines instance with kbo_weights list from input.

actions

Lines instance with actions list from input.

interps

Lines instance with interpretations list from input.

filename

Name of the input file.

input

Lines instance with all the lines from input.

output

Lines instance with all the lines from Prover9/Mace4 output.

Methods:

__init__(filename)

Constructor, filename is a string with the name/path of the input file. It can also receive a string (with embedded newlines).

find_proofs()

Tries to find a proof and returns the number of proofs found.

find_models(isofilter = False, ignore_constants = False)

Tries to find models and returns the number of interpretations found.

It takes two boolean parameters:

- **isofilter:** will eliminate models that are isomorphic to one already found.
- **ignore_constants:** will ignore all constants during the isomorphism tests.

find_both()

This function will launch two Proverx instances simultaneously (one with Prover9 and one with Mace4) and will return the first one that gives an answer (a proof or a model).

Note that the result is a Proverx object.

set_option(*args)

Sets Prover9/Mace4 options.

The parameter args is a list of srings.

The first argument should be 'Mace4' or 'Prover' (if omitted, the option will affect both cases).

Example:

```
p.set_option('Mace4','assign','max_models, 20')
```

set_option(*args)

Returns a Prover9/Mace4 option.

The parameter args is a list of srings.

The first argument should be 'Mace4' or 'Prover' (if omitted , the option will affect both cases).

Example:

```
p.get_option('Prover9','assign')
```

prooftrans(fname, args)

This function will use the utility Prooftrans to create a new file fname with a modified view of the proof.

The parameter args is a list of Prooftrans options:

renumber: renumber the steps of each proof consecutively, starting with step 1.

parents_only: list only the parents in the justifications, not the details about inference rules or positions.

- **striplabels:** tells Prooftrans to remove all label attributes on clauses.
- **Expand:** tells Prooftrans to produce more detailed proofs.
- **xml or XML**: tell Prooftrans to produce proofs in XML.
- **ivy or IVY**: tell Prooftrans to produce very detailed proofs that can be checked with the Ivy proof checker.
- **Hints:** tells Prooftrans to take all of the proofs in the file and produce one list of hints that can be given to Prover9 to guide subsequent searches on related conjectures.

For instance, this instruction will create a file called 'proof.txt' with all the clauses renumbered.

```
p.prooftrans('proof.txt','renumber')
```

Class Preprocessor

This class is automatically called by Proverx on initialisation. it will preprocess the input file and will create a new file with the same name and extension '.tmp' with pure Prover9 syntax. Proverx will then parse this file.

New directives:

The preprocessor allows to add some new directives to Prover9:

Include

This directive allows to include another file.

For instance:

```
include(preferences).
```

will include an external file called 'preferences' that can contain some common options. If the file is not found in your local folder, it will search it on your local include folder, lastly, if still not found, it will search the include folder on the server (for example, in this particular case there is a file called preferences that contains some common Prover9 and Mace4 options).

Another use is to include known theories like so:

```
include(group)
```

This will include a file called ‘group’ that contains the base axioms of a group.

Exponentiation

The ** operator can be used for exponentiation (this will also works for +,\,/,v,^ and @ symbols).

For instance:

```
x ** 3 = 1.
```

Translates to:

```
(x * x) * x = 1.
```

For loops

You can can create long lists of axioms using for loops.

For example:

```
for(n,2,5).
  x ** n = n.
  (y + z) ** n = 0.
end_for.
```

will be translated to:

```
x * x = 2.
(y + z) * (y + z) = 0.
(x * x) * x = 3.
((y + z) * (y + z)) * (y + z) = 0.
((x * x) * x) * x = 4.
(((y + z) * (y + z)) * (y + z)) * (y + z) = 0.
(((x * x) * x) * x) * x = 5.
((((y + z) * (y + z)) * (y + z)) * (y + z)) * (y + z) = 0.
```

Define functions by induction

For example, if you define two functions:

```
induction(f).
    f(0) = (x * y).
    f(n) = f(n-1) * (x**n).
end_induction.


induction(fn, k)

  % main func
  fn(k) = fn(k-1) + fn(k-2).
  base cases fn(1) = y. fn(0) = x.

end_induction.
```

then, the line:

```
f(2) = fn(3).
```

will be translated to:

```
(((x * y) * (x * x)) * (x * x)) = ((y + x) + y).
```

Notice that induction can take an optional second parameter with the name of the variable (n by default).

Class Lines

Lines is a utility class for handling text lines. It is used by axioms, goals, givens, etc.

It subclasses list and adds methods for saving and opening text files, set methods (union, intersection etc.) and a very useful method to generate all subsets. It can also backup itself (create an internal copy) and restore.

usage:

```
l = Lines()
l.add('Line two')
l.add('Line two')
l.save('example.txt')
```

Methods:

Lines.__init__(lines)

Constructor: lines can be a list of text lines or any other type (it's str representation will be stored).

Lines.add_first(line)

Inserts line at the beginning.

Lines.add_last(line)

Appends line to the end.

Lines.add(lines)

Adds one or more lines (lines can be a list).

Lines.remove_first()

Removes the first line.

Lines.remove_last()

Removes the last line.

Lines.clear()

Clears the list.

Lines.replace(lines)

This method is used (and preferred) instead of the usual assignment. It clears the list and add the new lines

Lines.backup()

Creates a copy of its internal list.

Lines.restore()

Restores the internal list to the one stored by backup.

Lines.save(fname)

Saves the list to file fname.

Lines.open(fname)

Opens a file of text lines and stores them in the internal list.

Lines.save_as_list(fname, list_name)

Saves the list to file fname but inserts "formulas(list_name)." at the beginning and appends "end_of_list." at the end. (This is useful, for example, to export hints).

Lines.open_from_list(fname)

Opens a text file contains a list definition (starting by "formulas" and ending by "end_of_list.") and removes the first and last line before storing the remaining lines.

Lines.union(lines)

Returns the union of lines with the internal list (identical lines are not duplicated).

Lines.intesection(lines)

Returns the intersection of lines with the internal list.

Lines.difference(lines)

Returns the difference (as a set) of lines with the internal list.

Lines.symmetric_difference(lines)

Returns the symmetric difference (as a set) of `lines` with the internal list.

Lines.update(lines)

Adds only the lines that are not already stored in the internal list.

Lines.issameset(lines)

Tests if lines and the internal list are the same set.

Lines.isdisjoint(lines)

Tests if lines and the internal list are disjoint.

Lines.issubset(lines)

Tests if lines is a subset of the internal list.

Lines.issuperset(lines)

Tests if lines is a superset of the internal list

Lines.superset_in(base)

Tests if the subset formed by the internal list is a superset of one of the sets of base.

Lines.subset_in(base)

Tests if the subset formed by the internal list is a subset of one of the sets of base.

Lines.is_in(base)

Tests if the subset formed by the internal list is equal to one of the sets of base.

Lines.subsets(order = 'asc')

Returns all subsets of the internal list. It can be ordered (`order` = 'asc' or 'desc') by the size of the subsets (by default it gives the smallest subsets first)

Module proofs

Class Proofs

This class acts as an array of proofs. It has an str representation similar to the one give by Prover9, so it is very easy to see all profs on screen (just use print).
Example:

```
p = Proverx('abelian.in')
p.find_proofs()
print p.proofs[0].stats.max_weight
```

This will give you the maximum weight of the first proof found (remember that indexes starts at 0 in Python).

Attributes:

count

int : the number of proofs.

found

boolean : indicates if at least one proof was found.

hints

Lines : a list of hints from all the proofs.

stats

ProofsStats : statistics about the proofs (see ProofsStats class).

Class Proof

This class encapsulates one proof. It acts as an ordered dict of clauses (see ProofClause class) indexed by the id of the clause.

Attributes:

hints

Lines : a list of hints extracted from the proof.

stats

ProofStats : statistics about the proofs (see ProofStats class).

Class ProofClause

This corresponds to a clause from the proof.

Attributes:

id (string)

literals (string)

attributes (string)

justifications (string)

Class GivenClause

This class encapsulates given clauses.

Attributes:

sequence (string)

pick (string)

weight (string)

id (string)

literals (string)

attributes (string)

justifications (string)

Class ProofStat

This class gives statistic informations about each proof.

Attributes:

number (int)

seconds (float)

label (string)

length (int)

level (int)

max_weight (float)

given (int)

Class ProofStats

This class gives statistic informations about all the proofs.

Attributes:

given (int)

generated (int)

kept (int)

proofs = (int)

usable = (int)

sos = (int)

demods = (int)

megs = (float)

time = (float)Module models

Class Models

This class acts as an array of interpretations. It has a str representation similar to the one give by Mace4, so it is very easy to see all models on screen (just use print).

Attributes:

count

int : the number of interpretations.

found

boolean : indicates if at least one interpretation was found.

Methods:

save(fname, format = 'standard', wrap = False)

Saves the interpretations found with one of the formats used by the utility interpformat:

- **standard:** This transformation simply extracts the structure from the file and reprints it in the same (standard) format, with one line for each operation. The result should be acceptable to any of the LADR programs that take standard structures.
- **standard2:** This is similar to standard, except that the binary operations are split across multiple lines to make them more human-readable. The result should be acceptable to any of the LADR programs that take standard structures. portable: This form is a lists of strings and natural numbers. It can be parsed by several scripting systems such as GAP, Python, and Javascript.
- **tabular:** This form is designed to be easily readable by humans. It is not meant for input to other programs.
- **raw:** This form is a sequence of natural numbers
- **cooked:** This form is a sequence of ground terms.

- **xml:** This is an XML form.
- **tex:** This generates LaTeX source for the interpretation.

If the optional **wrap** parameter is true, the items of the list will be wrapped in "list(interpretations)." and "end_of_list.".

Class Interp

This class implements each interpretation found in class models.

Attributes:

functions

An OrderedDict indexed by the name of the function (or constant) containing the matrix of the interpretation.

For example:

```
p = Proverx('abelian.in')
p.axioms.pop()
p.find_models()
print p.models[0].functions
```

will return:

```
OrderedDict([('c1', [0]), ('c2', [2]), ("'", [0, 1, 2, 4, 3, 5]), ('*', [1, 0, 3, 2,
5, 4, 0, 1, 2, 3, 4, 5, 4, 2, 1, 5, 0, 3, 5, 3, 0, 4, 1, 2, 2, 4, 5, 1, 3, 0, 3, 5, 4,
0, 2, 1])])
```

and

```
print p.models[0]['*']
```

will return:

```
[1, 0, 3, 2, 5, 4, 0, 1, 2, 3, 4, 5, 4, 2, 1, 5, 0, 3, 5, 3, 0, 4, 1, 2, 2, 4, 5, 1, 3,
0, 3, 5, 4, 0, 2, 1]
```

arity

An OrderedDict indexed by the name of the function (or constant) containing the arity of each function (or 0 for the constants).

For example:

```
p = Proverx('abelian.in')
p.axioms.pop()
p.find_models()
print p.models[0].arity
```

will return:

```
OrderedDict([('c1', 0), ('c2', 0), ("'", 1), ('*', 2)])
```

size

The size of the interpretation.

number

The number of the interpretation.

seconds

The seconds to find the interpretation.

Methods:

matrix(function, increment = 0)

Returns a matrix (a list of lists) from the desired function and optionally increments every element by some integer.

This can be useful to pass models to gap.

For instance:

```
print p.models[0].matrix('*', 1)
```

will return:

```
[[2, 1, 4, 3, 6, 5], [1, 2, 3, 4, 5, 6], [5, 3, 2, 6, 1, 4], [6, 4, 1, 5, 2, 3], [3,
5, 6, 2, 4, 1], [4, 6, 5, 1, 3, 2]]
```

Class Parallel

This class allows to run several Prover9/Mace4 jobs in parallel. This class is used internally by Proverx class for its find_both() method.

Attributes:

proc

This is a proverX object (the one that succeeded).

type

Type of answer ('Prover9' or 'Mace4').

proofs

Number of proofs found (in case type is 'Prover9').

models

Number of models found (in case type is 'Mace4').

Methods:

add(proc, proc_type, label = '')

Adds a Proverx object to it's internal list with the type of what we want to find (options are: 'Prover9' or 'Mace4') and an optional label to identify the answer given by run().

run(wait = .5)

Runs all Proverx objects in it's internal list (either find_proofs() or find_models() depending on the type).

Every wait seconds it checks if a model or a proof was found. If so, it will kill the others and return a string indicating what was found.

Example:

```
p1 = Proverx('abelian.in')
p2 = Proverx('abelian.in')
p3 = Proverx('abelian.in')
p4 = Proverx('abelian.in')
p5 = Proverx('abelian.in')

c = Parallel()

c.add(p1, 'Mace4', 'label mace 1 - 60 secs' )

p2.set_option('Mace4','assign','max_seconds, 5')
c.add(p2, 'Mace4', 'label mace 2 - 5 secs' )

p3.set_option('Mace4','assign','max_seconds, 10')
c.add(p3, 'Mace4', 'label mace 3 - 10 secs' )

p4.set_option('Mace4','assign','max_seconds, 15')
p4.axioms.pop()
c.add(p4, 'Mace4', 'label mace 4 - 15 secs' )

p5.set_option('Mace4','assign','max_seconds, 20')
c.add(p5, 'Mace4', 'label mace 5 - 20 secs' )

print c.run()
print "Proofs = {}".format(c.proofs)
print "Models = {}".format(c.models)
print c.type
if c.proc:
  print c.proc.label
```

Answer:

```
Found 1 interp(s).
Proofs = 0
Models = 1
None
label mace 4 - 15 secs
```

This code creates five Proverx objects initialised with file abelian.in (for which Mace4 cannot find a proof). Each one will have a different option for max_seconds (so it will run that amount of time).

They arre all added to Parallel (with a label to know who is who) but the last axiom is removed from p4, thus allowing Mace4 to find a model.

Module Strategies

This module is a collection of methods that implements various strategies for attacking complex proofs, finding shorter proofs, optimising parameters, etc.

Methods:

create_all_options(*options)

Given a list of options, it will create all possible combinations. There are two types of options: assign and set/clear. The assign options are a tuple with the first element designating the name of the option and the second is a list with all possible options. For set/clear, just enter a string with the name of the option.

Example:

```
all_options = create_all_options(('order', ['kbo', 'rpo', 'lpo']), 'factor')
for opt in all_options:
    print opt
```

Answer:

```
(['assign', 'order, kbo'], ['set', 'factor'])
(['assign', 'order, kbo'], ['clear', 'factor'])
(['assign', 'order, rpo'], ['set', 'factor'])
(['assign', 'order, rpo'], ['clear', 'factor'])
(['assign', 'order, lpo'], ['set', 'factor'])
(['assign', 'order, lpo'], ['clear', 'factor'])
```

find_best_options(fname , proof, stat, *options)

This function will open the file fname and will try to find the best set of options that optimizes stat (typically it will be 'length' to try to find the shortest proof). Proof will be the number of the proof to optimize (usually 0).

The parameter stat can be: length, seconds, max_weight, level or given. It can also be 'first' in which case all the options will run in parallel and the fastest one will be returned.

The function returns the minimum value found and the set of options.

Example:

```
print find_best_options('hard.in', 2, 'length', ('order', ['kbo', 'rpo', 'lpo']))
```

prove_with_goals(fname)

This function will open the file fname which should have several goals. It will then move all the goals except one to the assumptions and try to find a proof. If it does, it will add the hints from that proof and will remove one of the goals from the assumptions, and so on until it can prove that goal with the original axioms. It will then start all over again with another axiom.

The function returns a list of the proofs found.

Example:

```
proofs = prove_with_goals('McKenzie.in')
```

prove_with_interps(fname, time_out = 5)

This function will try to find models in all subsets of the axioms in order to eliminate certain assumptions and accelerate the process of finding a proof. If it finds a model, it will add it to the interpretation list of the original theory and will try to prove it. Both Prover9 and Mace4 will be given a max_seconds equal to the parameter time_out (the default is 5 seconds).

It returns a ProverX object if it finds a proof or None if not.

Example:

```
p = prove_with_interps('LT-82-2.in')
```

find_independent_axioms(fname, time_out = 5)

This function takes a file with a set of assumptions and will try to find all sets of independent axioms.

Example:

```
for f in find_independent_axioms('D.in', 1):
   print f
```

find_short_proof(fname, proof, global_time_out = 300)

This function tries to find the shortest proof by fiddling several parameters ('order', 'back_demod', 'and max_weight').

If after a period of time_out seconds it has not exhausted all possibilities, it will stop and return the shortest proof found until then.

Example:

```
p = find_short_proof('hard.in', 2, time_out = 30)
```

find_enumerating_axioms(fname, fixed_head, fixed_tail = 0, time_out = 5, hints = True)

This function will open the file fname and will fix a certain number of axioms at the beginning and at the end (fixed_head and fixed_tail). It will then pick each of the remaining axioms one by one and will try to find a proof. If it does, it will add the hints from that proof before trying with the second axiom.

Example:

```
p = find_enumerating_axioms("epivar.in", 13, 1, 300, True)
```

find_with_axioms_subsets(fname, fixed_head, fixed_tail = 0, time_out = 5, hints = True)

This function works similarly to find_enumerating_axioms() but instead of trying the remaining axioms one by one, it will try to find a proof for all the subsets of those axioms.

Example:

```
p = find_with_axioms_subsets("epivar.in", 13, 1, 300, True)
```

Module extprog

This module allows to run external programs and interact with them through *stdin/stdout*.

Public Methods:

which(program)

Returns the path where program is installed (replicate the bash command which).

Class Exec

This class executes an external program and interacts with its stdin, stdout streams, check if it's still running, kills it, etc.

Public Methods:

__init__(prog_name, parameters = [], keep_alive = False):

Constructor, prog_name is the path of the program, parameters can be passed through the parameters list.

If keep|_alive is true, the program keeps running in the background.

get(timeout = 5)

Returns the last output from the running program. If after timeout seconds there is no response, the function returns (default 5 seconds).

command(cmd, timeout = .1)

Function that sends cmd to the running program and returns the output.

kill()

Sends a kill signal to the running program.

terminate()

Sends a terminate signal to the running program.

close()

Closes the running program.

poll()

Returns the return code of the running program if it has finished running or None otherwise.

retcode()

The same as poll.

running()

Returns true if the program is still running.

set_quit_cmd(quit_cmd)

Sets the command used for closing the running program. (for instance 'quit;' in GAP). If no command is provided, the program is just killed.

Class Gap

This class is a subclass of Exec. It creates an instance of GAP and interacts with it by sending commands.

```
>g = Gap()
>g('S4:=SymmetricGroup(4)')
'Sym( [ 1 .. 4 ] )'
```

Class Race

This class creates a list of programs and executes them in parallel. If one of the programs finishes first, all the others will be killed. The output (and eventual error message) from the first program is kept.

Instance variables:

progs

List of running programs.

output

String with the output from the first program that finished.

error

String with the error output from the first program that finished.

Public Methods:

__init__(progs)

Constructor, accepts a list of paths:

add(prog)

Adds a program path to the list of programs.

run(interval = 1)

Executes the list of programs concurrently. It checks if a program has finished every interval seconds (default is one second, this value can be a float).

Extending ProverX

ProverX can be easily extended by writing Python modules. These modules can be installed on the system folder (reserved for ProverX), the lib folder (for all third-party extensions) or the user folder.

Writing packages

A package can be a simple Python function or a class. It can also be an external program or programs (or even a web API) but, in this case, there must be a Python module acting as an interface. To access ProverX functionality, the module should always start by importing proverx (see example below).

An example

Here is an example of an extension that draws a directed graph of the proof (see Quick Start):

Create a file called proofgraph.py

```
import re
import graphviz
import proverx

class ProofGraph(object):

    def __init__(self, proof = None, filename='graph'):
        self.graph = graphviz.Digraph(format='png', filename = filename + '.gv')
        self.graph.node('$F',shape='doublecircle')
        for id, clause in proof.clauses.iteritems():
            depends =  self.depends_from(clause.justifications)
            if depends:
                for dep_id in depends:
                    self.graph.edge(proof.clauses[dep_id].literals,
proof.clauses[id].literals)

    def depends_from(self, clause):
        ids = []
        rewrites = re.findall(r"rewrite\(\[.*?\]\)", clause, re.I) #find
rewrite justifications
        for rewrite in rewrites:
            ids += re.findall(r"([0-9]+[A-Z]*)\(", rewrite, re.I) #get clauses ids from
rewrite
            clause = clause.replace(rewrite, '')
        paras = re.findall(r"para\(.*?\)\)", clause, re.I) #find para
        justifications for para in paras:
            ids += re.findall(r"([0-9]+[A-Z]*)\(", para, re.I) #get clauses ids from
para
            clause = clause.replace(para, '')
        ids += re.findall('[0-9]+[A-Z]*', clause, re.I) # get remaining ids
        return list(set(ids)) #remove duplicate ids

     def create_graph(self):
         self.graph.render()
```

Once installed this extension can be used like so:

```
from proofgraph import *
pg = ProofGraph(p.proofs[0], 'abelian')
pg.create_graph()
```

References

[1] G. Kolata, “With Major Math Proof, Brute Computers Show Flash of Reasoning Power,” 10 December 1996. [Online]. Available: https://www.nytimes.com/1996/12/10/science/with-major-math-proof-brute-computers-show-flash-of-reasoning-power.html. [Accessed 28 8 2019].

[2] W. McCcune, “Solution of the Robbins Problem,” vol. 3, Berlin, Heidelberg, Springer-Verlag, 1997, pp. 263-276, DOI:10.1023/A:1005843212881.

[3] .. Murawski and W. Marciszewski, Mechanization of Reasoning in a Historical Perspective, volun 3 of Poznam Studies in the Philosophy of the Sciences and the Humanities, Amsterdam: Rodopi, 995.

[4] G. Boole, “The Calculus of Logic,” *The Cambridge and Dublin Mathematical Journal,* no. 3, pp. 183-198, 1848.

[5] J. Harrison, “A Short Survey of Automated Reasoning,” in *Algebraic Biology*, 2007.

[6] B. Russell and A. N. Whitehead, Principia Mathematica, Cambridge University Press, 1910, 1912 and 1913.

[7] P. Gilmore, “A proof method for quantification theory: its justification and realization,” *IBM Journal of Research and Development,* vol. 4, pp. 28-35, 1960, DOI:10.1147/rd.41.0028.

[8] J. A. Robinson, “A machine oriented logic based on the resolution principle,” *Journal of ACM,* vol. 12, pp. 23-41, 1965, DOI:10.1145/321250.321253.

[9] L. Henkin, J. Monk and A. Tarski, “Cylindric algebras, Part I, Studies in Logic and the Foundations of Mathematics,” vol. 64, Amsterdam - London, North-Holland Publishing co, 1971.

[10] G. F. McNulty, “A field guide to equational logic,” *Journal of Symbolic Computation,* vol. 14, p. 371–397, 1992.

[11] D. Pigozzi, *Equational logic and equational theories of algebras,* Mimeographed at Iowa State University, 1970, and reissued at Purdue University, 1975..

[12] W. Taylor, “Equational logic,” *Houston J. Math,* vol. 37, pp. 1-83, 1979, DOI:10.1007/978-0-387-77487-9_13.

[13] L. Wos, R. Veroff and G. W. Pieper, “Logical Basis for the Automation of Reasoning: Case Studies,” 2002.

[14] E. Lusk and R. McFadden, “Using automated reasoning tools: a study of the semigroup F2B2. Semigroup,” vol. 1, no. Forum 36, pp. 75-87, 1987.

[15] R. Padmanabhan and W. McCune, Automated Deduction in Equational Logic and Cubic Curves, vol. 1095, Springer Verlag, 1995.

[16] A. Voronkov, “Automated Reasoning: Past Story and New Trends,” in *Eighteen IJCAI*, Acapulco, 2003.

[17] L. Bachmair, N. Dershowitz, D. Plaisted, H. Ait-Kaci and M. Nivat, “Completion Without Failure,” in *Resolution of Equations in Algebraic Structures*, Academic Press, 1989, pp. 1-30.

[18] A. Voronkov, “AVATAR: The Architecture for First-Order Theorem Provers,” in *Computer Aided Verification, Lecture Notes in Computer Science*, vol. 8559, Springer, Cham, 2014.

[19] J. Araújo, M. Kinyon and J. Konieczny, “Minimal paths in the commuting graphs of semigroups,” *European Journal of Combinatorics,* vol. 32, no. 2, pp. 178 - 197, 2011.

[20] J. Araújo and M. Kinyon, “Centralizers in the full transformation semigroup,” *Semigroup Forum 82,* no. Nº 2, pp. 319-323, 2011.

[21] J. Araújo, M. Kinyon and A. malheiro, “A new definition of conjugacy for semigroups,” *Proc. R. Soc. Edinb.,* vol. 143, no. Nº2, pp. 1115-1122, 2013.

[22] J. Araújo and M. Kinyon, “Orbits of primitive k-homogenous groups on (nk)-partitions with applications to semigroups,” *Semigroup Forum 89,* vol. Nº2, no. 469, pp. 469-474, 2014.

[23] J. Araújo, A. Malheiro and J. Konieczny, “Conjugation in Semigroups,” *Journal of Algebra,* vol. 403, pp. 93-134, 2014, DOI:10.1016/j.jalgebra.2013.12.025, arXiv:1301.1568 .

[24] M. Fowler, in *Domain Specific Languages*, 2010, pp. 46Addison-Wesley Professional.