Automated Theorem Proving for Prolog Verification*

Fred Mesnard Thierry Marianne

Étienne Payet

LIM, université de La Réunion, France

{frederic.mesnard,thierry.marianne,etienne.payet}@univ-reunion.fr

LPTP (Logic Program Theorem Prover) is an interactive natural-deduction-based theorem prover for pure Prolog programs with negation as failure, unification with the occurs check, and a restricted but extensible set of built-in predicates. With LPTP, one can formally prove termination and partial correctness of such Prolog programs. LPTP was designed in the mid-1990's by Robert F. Stärk. It is written in ISO-Prolog and comes with an Emacs user-interface.

From a theoretical point of view, in his publications about LPTP, Stärk associates a set of first-order axioms IND(P) to the considered Prolog program P. IND(P) contains the Clark's equality theory for P, definitions of success, failure and termination for each user-defined logic procedure in P, axioms relating these three points of view, and an axiom schema for proving inductive properties. LPTP is thus a dedicated proof editor where these axioms are hard-wired.

We propose to translate these axioms as first-order formulas (FOFs), and apply automated theorem provers to check the property of interest. Using FOF as an intermediary language, we experiment the use of automated theorem provers for Prolog program verification. We evaluate the approach over a benchmark of about 400 properties of Prolog programs from the library available with LPTP. Both the compiler which generates a set of FOF files from a given input Prolog program together with its properties and the benchmark are publicly available.

1 Introduction

In the mid-1990's, Robert F. Stärk defined a framework for Prolog verification [26, 29]. He considered a subset of ISO-Prolog [13]: pure Prolog programs with negation as failure, unification with the occurs check, and allowed a restricted but extensible set of built-in predicates. He presented a first-order formalisation with axiom schemas of the usual operational semantics of Prolog. A safeness condition included in termination condition imposes groundness before evaluation of negated goals. He showed soundness and completeness for termination, success, and failure. The framework also allows partial correctness properties to be proved by induction w.r.t. the clauses defining predicates, considered as inductive definitions. Some examples will be discussed in Section 3 and Section 4. The logical theory was hard-wired in an interactive dedicated first-order natural-deduction-based theorem prover called LPTP (Logic Program Theorem Prover). Stärk implemented LPTP in ISO-Prolog, together with an Emacs user-interface, an HTML and TEX manager, a detailed user-manual, and a library of predicates for Peano numbers, integers, lists, sorting algorithms, etc. with numerous proven properties. A copy of LPTP is vailable at `https://github.com/FredMesnard/lptp`.

Thirty years later, LPTP is still running on any ISO-Prolog processor, with its initial interface. Today, formal verification of computer programs is an established discipline within computer science. Nonetheless, program verification by interactive theorem proving is still a slow process and requires non-trivial

*This paper is an updated version of [21] that appeared in the 2024 LPAR Complementary Volume.

EPTCS ??, 20??, pp. 446–458, doi:10.4204/EPTCS.??.33

skills. On the other hand, during the last three decades, the increase in computing power and the advances in automated theorem proving have been notable. For instance, TPTP (Thousands of Problems for Theorem Provers, [30]) is a library of test problems for automated theorem proving. It provides online tools to check the syntax of input problems and apply a bunch of user selected automated theorem provers. Among them, E [25] and Vampire [16] are two powerful freely available automated theorem provers, performing very well in many international competitions over the years. Interactive theorem prover implementers were able to take advantage of these progress by implementing so-called *hammers* for their tools, see e.g. [23, 3].

This evolution raises the following questions: can we also use the TPTP FOF *Esperanto* to formulate the logic theory Stärk associates to a logic program? Can we use *off-the-shelf* TPTP provers and obtain automatic proofs in reasonable time? Can we get an acceptable success rate with such an approach?

The main contribution of this paper is the following. Using FOF (*First-Order Form*, one of the logic languages proposed by TPTP, see [31]) as an intermediary language, we describe the first – to the best of our knowledge – experiment of the use of automated theorem provers, namely E and Vampire, for Prolog program verification, including termination and partial correctness. We evaluate the approach over about 400 properties of Prolog programs. Both the compiler applying Stärk's theory to a given input Prolog program and its properties to a set of FOF files and the benchmark are publicly available at `https://github.com/atp-lptp/automated-theorem-proving-for-prolog-verification`.

We organize the paper as follows. The next section presents a brief summary of the LPTP system. The third section describes step by step how to compile a Prolog program, its associated LPTP axioms and a property of interest into a FOF file. Then we present an experimental evaluation, related work and we conclude.

2 Notation

FOF (*First Order Form*) is a well-known logic language from TPTP for expressing first-order logic (FOL) axioms and conjectures. A formula is written `fof(`*name*`,`*role*`,`*formula*`)`, where *name* is the name of the formula, *role* is either `axiom` or `conjecture` and *formula* is informally defined as follows:

FOL	FOF	FOL	FOF
$A \wedge B$	`A & B`	$\neg p(x)$	`~ p(X)`
$A \vee B$	`A | B`	$\exists x.A$	`?[X] : A`
$A \rightarrow B$	`A => B`	$\forall x.A$	`![X] : A`

Numerous examples will appear in the next sections.

Let P be a pure logic program where negative literals may appear in the body of clauses (also called *normal program* in [19]). For sake of conciseness, we do not consider built-in predicates (see [29] for a full treatment) other than the equality `=/2`. We start with $\mathscr{L}$, the first-order language associated with P. The *goals* of $\mathscr{L}$ are:

$$G,H ::= \texttt{true} \mid \texttt{fail} \mid s = t \mid A \mid \texttt{\textbackslash+}\ G \mid (G,H) \mid (G;H) \mid \texttt{some}\ x\ G$$

where s and t are two terms, x is a variable and A is an atomic goal. The goals of $\mathscr{L}$ have the operational semantics specified by ISO-Prolog [13] assuming the occurs check.

$\hat{\mathscr{L}}$ is the specification language of LPTP. For each user-defined predicate symbol R, $\hat{\mathscr{L}}$ does not include R, but instead it contains three predicate symbols R^s, R^f, R^t of the same arity as R, which respectively express success, failure and termination of R. $\hat{\mathscr{L}}$ also contains a unary constraint for groundness

gr, expressing that its argument is ground. The *formulas* of $\hat{\mathscr{L}}$ are:

$$\phi, \psi ::= \top \mid \bot \mid s = t \mid R(\vec{t}) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall x\phi \mid \exists x\phi$$

where $\vec{t}$ is a sequence of n terms and R denotes a n-ary predicate symbol of $\hat{\mathscr{L}}$. The semantics of $\hat{\mathscr{L}}$ is the first-order predicate calculus of classical logic.

For any of the user-defined logic procedure R in a logic program P, $D_R^P(\vec{x})$ denotes its Clark's *if-and-only-if* completed definition, cf. [5, 19].

For defining the declarative semantics of logic programs, Stärk uses three syntactic operators **S**, **F** and **T** which map goals of $\mathscr{L}$ into $\hat{\mathscr{L}}$-formulas. Intuitively, **S***G* means *G* succeeds (any breadth-first evaluation of *G* succeeds), **F***G* means *G* fails (the ISO-Prolog evaluation stops without any answer), and **T***G* means *G* terminates (the ISO-Prolog evaluation produces a finite number of answers then stops). The definition of the operators follows:

$\mathbf{S}R(\vec{t}) := R^s(\vec{t})$	$\mathbf{S}\ \texttt{true} := \top$	$\mathbf{S}\ \texttt{fail} := \bot$	$\mathbf{S}(s=t) := (s=t)$
$\mathbf{S}\backslash\texttt{+}G := \mathbf{F}G$	$\mathbf{S}(G,H) := \mathbf{S}G \wedge \mathbf{S}H$	$\mathbf{S}(G;H) := \mathbf{S}G \vee \mathbf{S}H$	$\mathbf{S}(\texttt{some}\ x\ G) := \exists x \mathbf{S}G$
$\mathbf{F}R(\vec{t}) := R^f(\vec{t})$	$\mathbf{F}\ \texttt{true} := \bot$	$\mathbf{F}\ \texttt{fail} := \top$	$\mathbf{F}(s=t) := \neg(s=t)$
$\mathbf{F}\backslash\texttt{+}G := \mathbf{S}G$	$\mathbf{F}(G,H) := \mathbf{F}G \vee \mathbf{F}H$	$\mathbf{F}(G;H) := \mathbf{F}G \wedge \mathbf{F}H$	$\mathbf{F}(\texttt{some}\ x\ G) := \forall x \mathbf{F}G$

$\mathbf{T}R(\vec{t}) := R^t(\vec{t})$	$\mathbf{T}\ \texttt{true} := \top$
$\mathbf{T}\ \texttt{fail} := \top$	$\mathbf{T}(s=t) := \top$
$\mathbf{T}\backslash\texttt{+}G := \mathbf{T}G \wedge gr(G)$	$\mathbf{T}(G,H) := \mathbf{T}G \wedge (\mathbf{F}G \vee \mathbf{T}H)$
$\mathbf{T}(G;H) := \mathbf{T}G \wedge \mathbf{T}H$	$\mathbf{T}(\texttt{some}\ x\ G) := \forall x \mathbf{T}G$

Note that termination requires a safe use of negation, see the definition of $\mathbf{T}\backslash\texttt{+}G$ where the goal *G* has to be proved terminating and ground at proof time. Finally, we add the definition of *gr*, which belongs to the specification language and is needed for defining $\mathbf{T}\backslash\texttt{+}G$:

$gr(\texttt{true}) := \top$	$gr((G,H)) := gr(G) \wedge gr(H)$
$gr(\texttt{fail}) := \top$	$gr((G;H)) := gr(G) \wedge gr(H)$
$gr(s=t) := gr(s) \wedge gr(t)$	$gr(\texttt{some}\ x\ G) := \exists x\ gr(G)$
$gr(R(t_1,\ldots,t_n)) := gr(t_1) \wedge \ldots \wedge gr(t_n)$	$gr(\backslash\texttt{+}G) := gr(G)$

We refer the reader to the papers of Stärk [26, 27, 28, 29] for a complete presentation of LPTP.

3 Compiling LPTP axioms to FOF

With LPTP, we prove properties of a logic program *P* w.r.t. its *inductive extension* IND(*P*) which includes Clark's completion [5] and induction along the definition of the predicates. Stärk shows that the inductive extension is always consistent and proves various correctness and completeness results w.r.t. the operational semantics of Prolog [29]. The first-order theory IND(*P*) (cf. [29], pp. 253–254) is defined by nine axiom schemas which we describe now, along with their translation in FOF. We omit the fixed point axioms for builtins. Let us also point out that the specification language $\hat{\mathscr{L}}$ of LPTP can be extended by new function and predicate symbols at a logical level (there is no associated Prolog code). As shown in [21], such function and predicate definitions can also be compiled into FOF.

3.1 First steps

Let us consider the following logic program ADD as our running example.

```
nat(0).                          add(0,Y,Y).
nat(s(X)) :- nat(X).             add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

We discuss the axioms proposed by Stärk and apply them to the ADD program.

The axioms of Clark's equality theory

1. $f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \rightarrow x_i = y_i$ [if f is n-ary and $1 \leq i \leq n$]
2. $f(x_1,\ldots,x_n) \neq g(y_1,\ldots,y_m)$ [if $n \neq m$ or $f \not\equiv g$]
3. $t \neq x$ [if x occurs in t and $t \not\equiv x$]

The first two axioms specify some properties of the trees built from the function symbols extracted from the program under consideration. The third axiom forbids infinite trees. Note that it is actually an axiom schema, i.e., an infinite set of first order axioms. We will omit it but we stay sound. Here is the FOF version (see Section 2 for the FOF syntax):

```
fof(id1,axiom,! [Xx4] : ! [Xx5] : (s(Xx4) = s(Xx5) => Xx4 = Xx5)).
fof(id2,axiom,! [Xx3] : ~ ('0' = s(Xx3))).
```

Axioms for gr/1

4. $\mathrm{gr}(c)$ [if c is a constant]
5. $\mathrm{gr}(x_1) \wedge \ldots \wedge \mathrm{gr}(x_m) \leftrightarrow \mathrm{gr}(f(x_1,\ldots,x_m))$ [f is m-ary]

Actually, LPTP deals with *non-ground* terms and offers a predefined predicate $gr/1$ that we can consider as a constraint. This relation is useful for instance for dealing with negation as failure as LPTP only allows negation by failure for *ground* goals (see the definition $\mathbf{T}\backslash{+}G$). Back to our example, here is the FOF version:

```
fof(id4,axiom,gr('0')).
fof(id5,axiom,! [Xx6] : (gr(Xx6) <=> gr(s(Xx6)))).
```

The ADD program contains two user-defined predicates, `add/3` and `nat/1`. LPTP considers each user-defined predicate through three points of view: failure, success and termination. So LPTP creates the following predicates: `add_fails/3`, `add_succeeds/3`, `add_terminates/3`, and similarly for `nat/1`. These three viewpoints are linked with the following axioms, where R^s (resp. R^f and R^t) denotes `R_succeeds/3` (resp. `R_fails/3` and `R_terminates/3`).

Uniqueness axioms and totality axioms

6. $\neg(R^s(\vec{x}) \wedge R^f(\vec{x}))$ [if R is a user-defined predicate]
7. $R^t(\vec{x}) \rightarrow (R^s(\vec{x}) \vee R^f(\vec{x}))$ [if R is a user-defined predicate]

Axiom 6 says that for any tuple of (possibly non-ground) terms, we cannot have at the same time success and failure of R. Axiom 7 states that given termination, we have success or failure. Altogether, it means that for any tuple of terms $\vec{x}$, assuming termination, either $R(\vec{x})$ succeeds or (exclusively) $R(\vec{x})$ fails. So for our example, we get:

```
fof(ida6,axiom,! [Xx7,Xx8,Xx9] :
```

```
    ~ ((add_succeeds(Xx7,Xx8,Xx9) & add_fails(Xx7,Xx8,Xx9)))).
fof(ida7,axiom,! [Xx7,Xx8,Xx9] :
    (add_terminates(Xx7,Xx8,Xx9) =>
    (add_succeeds(Xx7,Xx8,Xx9) | add_fails(Xx7,Xx8,Xx9)))).
fof(idn6,axiom,! [Xx10] :
    ~ ((nat_succeeds(Xx10) & nat_fails(Xx10)))).
fof(idn7,axiom,! [Xx10] :
    (nat_terminates(Xx10) =>
    (nat_succeeds(Xx10) | nat_fails(Xx10)))).
```

Fixed point axioms for user-defined predicates R

8. $R^s(\vec{x}) \leftrightarrow \mathbf{S}D_R^P(\vec{x}),\ R^f(\vec{x}) \leftrightarrow \mathbf{F}D_R^P(\vec{x}),\ R^t(\vec{x}) \leftrightarrow \mathbf{T}D_R^P(\vec{x})$

We recall that $D_R^P(\vec{x})$ denotes the definition of the completion [5] of the user-defined procedure $R(\vec{x})$ in the logic program P. In the previous section, we saw how to apply the operator **S**, **F** and **T** to formulas. So for instance, the first equivalence $R^s(\vec{x}) \leftrightarrow \mathbf{S}D_R^P(\vec{x})$ defines $R^s(\vec{x})$. Back to our running example, we get:

```
fof(idns8,axiom,! [Xx1] : (nat_succeeds(Xx1) <=>
    (? [Xx2] : (Xx1 = s(Xx2) & nat_succeeds(Xx2)) | Xx1 = '0'))).
fof(idnf8,axiom,! [Xx1] : (nat_fails(Xx1) <=>
    (! [Xx2] : (~ (Xx1 = s(Xx2)) |
                nat_fails(Xx2)) & ~ (Xx1 = '0')))).
fof(idnt8,axiom,! [Xx1] : (nat_terminates(Xx1) <=>
    (! [Xx2] : ((~ (Xx1 = s(Xx2)) | nat_terminates(Xx2)))))).
```

and similarly for `add/3`.

Finally, for any property of the form $\forall\vec{x}[R^s(\vec{x}) \rightarrow \phi(\vec{x})]$, where $R(\vec{x})$ is a user-defined procedure and $\phi(\vec{x})$ an $\hat{\mathscr{L}}$ -formula, we have an induction schema. The interactive prover LPTP is able to *dynamically* generate an induction axiom on demand while the user interacts with it. In our approach, we *statically* generate the induction axiom *once* from the conjecture to be proved, if the conjecture can be easily rewritten as required. This is a potential source of imprecision, but again we stay sound. Let us examine a simple case. It is exactly what happens using LPTP, which slightly generalizes [29]. By *directly recursive user-defined predicate* in the box below, we forbid mutual recursive definitions. Of course, LPTP is able to handle mutually recursive properties, see [26] for some examples.

A (simplified) induction schema for a user-defined predicate R

Let R be a directly recursive user-defined predicate and let $\phi(\vec{x})$ be an $\hat{\mathscr{L}}$-formula such that the length of $\vec{x}$ is equal to the arity of R.
Let $sub(\phi(\vec{x})/R)$ be the formula to be proven $\forall\vec{x}(R^s(\vec{x}) \rightarrow \phi(\vec{x}))$.
Let $closed(\phi(\vec{x})/R)$ be the formula obtained from $\forall\vec{x}(\mathbf{S}D_R^P(\vec{x}) \rightarrow R^s(\vec{x}))$ by replacing

- $R^s(\vec{x})$ by $\phi(\vec{x})$ on the right of $\rightarrow$,
- all occurrences of $R(\vec{t})$ appearing on the left of $\rightarrow$ by $\phi(\vec{t}) \wedge R(\vec{t})$.

Then the induction axiom is the following formula:

9. $closed(\phi(\vec{x})/R) \rightarrow sub(\phi(\vec{x})/R)$

Let us apply this axiom to the following property, informally stated as: for any term x, if `nat(`x`)` then `add(`x`,0,`x`)`. Expressed in LPTP, it gives: for any term x, if `nat_succeeds(`x`)` then `add_succeeds(`x`,0,`x`)`, which is exactly the formula $sub(\phi(\vec{x})/R)$ of axiom 9. So $R \equiv$ `nat`, $R^s \equiv$ `nat_succeeds` and $\phi(\vec{x}) \equiv$ `add_succeeds`$(x,0,x)$.

For the left-hand side of axiom 9, we start from

$$\forall x(\mathbf{S}D_{nat}^{ADD}(x) \rightarrow \texttt{nat\_succeeds}(x))$$

We have $D_{nat}^{ADD}(x) \equiv x = 0 \vee \exists y(x = s(y) \wedge \texttt{nat}(y))$. We replace `nat`$(y)$ by `nat`$(y) \wedge$ `add_succeeds`$(y,0,y)$. We replace `nat_succeeds`(x) by `add_succeeds`$(x,0,x)$. We get: $\forall x$ ($\mathbf{S}$ $[x = 0 \vee \exists y(x = s(y) \wedge \texttt{nat}(y) \wedge$ `add_succeeds`$(y,0,y))] \rightarrow$ `add_succeeds`$(x,0,x))$. We apply $\mathbf{S}$ and obtain: $\forall x([x = 0 \vee \exists y(x = s(y) \wedge$ `nat_succeeds`$(y) \wedge$ `add_succeeds`$(y,0,y))] \rightarrow$ `add_succeeds`$(x,0,x))$.

Summarizing, in FOF, associated with the property to be proved:

```
fof(lemma,conjecture,
! [Xx] : (nat_succeeds(Xx) => add_succeeds(Xx,'0',Xx))).
```

we obtain the following induction axiom:

```
fof(induction,axiom,(
    ! [Xx] :
        ((? [Xx2] : (Xx = s(Xx2) & (nat_succeeds(Xx2)
                                    & add_succeeds(Xx2,'0',Xx2)))
        | Xx = '0') => add_succeeds(Xx,'0',Xx))
=>
    ! [Xx] : (nat_succeeds(Xx) => add_succeeds(Xx,'0',Xx)))).
```

We can gather all the 15 axioms, including the axioms defining `add_success/3`, `add_fails/3`, and `add_terminates/3` and the conjecture plus its induction axiom in a file, say `test.fof` and submit it to the E prover or to Vampire. Both systems will find a refutation in a fraction of a second on a standard laptop.

It allows us to conclude for any term x, if `nat(`x`)` then `add(`x`,0,`x`)` is true. Operationally, for any natural number n, in the Prolog search tree corresponding to the goal `add`$(s^n(0),0,s^n(0))$, the empty clause appears. Assuming termination, which will be shown later, it means that the user will get (at least) one positive answer for the query `:-` `add`$(s^n(0),0,s^n(0))$. when executed with any ISO-Prolog system.

Here's the manual proof of the same property in its LPTP version (a Prolog file), followed by its TEX version produced by LPTP. Using the interactive LPTP Emacs mode, we began this proof by invoking the

ind tactic, asking for an inductive proof. Both the base case and the inductive case were automatically generated and completed by LPTP.

```
:- lemma(add:x_0_x, all [x]: succeeds nat(?x) => succeeds add(?x,0,?x),
induction([all x: succeeds nat(?x) => succeeds add(?x,0,?x)],
 [step([],[],[],succeeds add(0,0,0)),
  step([x], [succeeds add(?x,0,?x), succeeds nat(?x)], [],
   succeeds add(s(?x),0,s(?x)))])).
```

Lemma [*add:x_0_x*] $\forall x\,(\mathbf{S}\,\mathtt{nat}(x) \rightarrow \mathbf{S}\,\mathtt{add}(x,0,x))$.

Proof.

Induction$_0$: $\forall x\,(\mathbf{S}\,\mathtt{nat}(x) \rightarrow \mathbf{S}\,\mathtt{add}(x,0,x))$.

Hypothesis$_1$: none.

Conclusion$_1$: $\mathbf{S}\,\mathtt{add}(0,0,0)$.

Hypothesis$_1$: $\mathbf{S}\,\mathtt{add}(x,0,x)$ and $\mathbf{S}\,\mathtt{nat}(x)$.

Conclusion$_1$: $\mathbf{S}\,\mathtt{add}(\mathtt{s}(x),0,\mathtt{s}(x))$. □

3.2 A second property

Now let us consider the following property: for any x, y and z such that $\mathtt{nat}(x)$, $\mathtt{nat}(y)$ and $\mathtt{add}(s(x),y,z)$, we have $\mathtt{add}(x,s(y),z)$. Let us first assert the previous property as an axiom, which can now be freely used by the automated prover, then we have our new conjecture:

```
fof('lemma-(add:x_0_x)',axiom,
  ! [Xx] : (nat_succeeds(Xx) => add_succeeds(Xx,'0',Xx))).

fof('lemma-(add:succ)',conjecture,
  ! [Xx,Xy,Xz] : (((nat_succeeds(Xx) & nat_succeeds(Xy))
                   & add_succeeds(s(Xx),Xy,Xz))
                   => add_succeeds(Xx,s(Xy),Xz))).
```

In order to generate an induction axiom for this property, we first rewrite it in the form $\forall\vec{x}[R^s(\vec{x}) \rightarrow \phi(\vec{x})]$ and we apply the simplified induction schema for user-defined predicates. It gives:

```
fof(induction,axiom,(
! [Xx] :
    ((? [Xy25] :
      (Xx = s(Xy25) & (nat_succeeds(Xy25)
       & ! [Xy,Xz] : ((add_succeeds(s(Xy25),Xy,Xz)
                       & nat_succeeds(Xy))
      => add_succeeds(Xy25,s(Xy),Xz))))
    | Xx = '0') =>
      ! [Xy,Xz] : ((add_succeeds(s(Xx),Xy,Xz) & nat_succeeds(Xy))
         => add_succeeds(Xx,s(Xy),Xz)))
 => ! [Xx] : (nat_succeeds(Xx)
    => ! [Xy,Xz] : ((add_succeeds(s(Xx),Xy,Xz) & nat_succeeds(Xy))
                    => add_succeeds(Xx,s(Xy),Xz))))).
```

Again, we can gather all axioms, the conjecture and its induction axiom in a file and submit it to Vampire, which will find a refutation in about one minute.

3.3 Commutativity of Peano addition

We are now equipped to consider commutativity of Peano addition: for any x,y,z, if $\texttt{add}(x,y,z)$ then $\texttt{add}(y,x,z)$. Of course, stated this way, the property is false. We need to enforce that x and y are Peano numbers. So first we add our two previous properties as axioms. Here is our new conjecture, associated with its induction axiom:

```
fof('theorem-(add:commutative)',conjecture,
  ! [Xx,Xy,Xz] : (((nat_succeeds(Xx) & nat_succeeds(Xy))
                   & add_succeeds(Xx,Xy,Xz))
                  => add_succeeds(Xy,Xx,Xz))).

fof(induction,axiom,
(! [Xx] :
  ((? [Xy26] : (Xx = s(Xy26) & (nat_succeeds(Xy26)
    & ! [Xy,Xz] : ((add_succeeds(Xy26,Xy,Xz) & nat_succeeds(Xy))
    => add_succeeds(Xy,Xy26,Xz))))
    | Xx = '0') =>
      ! [Xy,Xz] : ((add_succeeds(Xx,Xy,Xz) & nat_succeeds(Xy))
    => add_succeeds(Xy,Xx,Xz)))
=>
! [Xx] : (nat_succeeds(Xx) =>
  ! [Xy,Xz] : ((add_succeeds(Xx,Xy,Xz) & nat_succeeds(Xy))
  => add_succeeds(Xy,Xx,Xz))))).
```

The conjecture is proved in a fraction of a second by Vampire.

3.4 Some termination proofs

Finally, let us prove some termination properties about `add/3`. It is immediate to see that the Prolog proof of $\texttt{add}(x,y,z)$ terminates if $\texttt{nat}(x)$ or $\texttt{nat}(z)$. We prove this by stating two lemmas which we will gather in a theorem. Here are the LPTP properties and their proofs (we omit the second one).

Lemma [*add:term:1*] $\forall x,y,z\,(\mathbf{S}\,\texttt{nat}(x) \rightarrow \mathbf{T}\,\texttt{add}(x,y,z))$. **Proof.**

Induction$_0$: $\forall x\,(\mathbf{S}\,\texttt{nat}(x) \rightarrow \forall y,z\,\mathbf{T}\,\texttt{add}(x,y,z))$.

Hypothesis$_1$: none.

Conclusion$_1$: $\forall y,z\,\mathbf{T}\,\texttt{add}(0,y,z)$.

Hypothesis$_1$: $\forall y,z\,\mathbf{T}\,\texttt{add}(x,y,z)$ and $\mathbf{S}\,\texttt{nat}(x)$.

Conclusion$_1$: $\forall y,z\,\mathbf{T}\,\texttt{add}(\texttt{s}(x),y,z)$. □

Lemma [*add:term:3*] $\forall x,y,z\,(\mathbf{S}\,\texttt{nat}(z) \rightarrow \mathbf{T}\,\texttt{add}(x,y,z))$. **Proof.** Similar. □

Theorem [*add:term*] $\forall x,y,z\,(\mathbf{S}\,\mathtt{nat}(x) \vee \mathbf{S}\,\mathtt{nat}(z) \rightarrow \mathbf{T}\,\mathtt{add}(x,y,z))$. **Proof.**

Assumption$_0$: $\mathbf{S}\,\mathtt{nat}(x) \vee \mathbf{S}\,\mathtt{nat}(z)$.

Case$_1$: $\mathbf{S}\,\mathtt{nat}(x)$. $\mathbf{T}\,\mathtt{add}(x,y,z)$ by Lemma 1 [*add:term:1*].

Case$_1$: $\mathbf{S}\,\mathtt{nat}(z)$. $\mathbf{T}\,\mathtt{add}(x,y,z)$ by Lemma 2 [*add:term:3*].

Hence$_1$, in all cases: $\mathbf{T}\,\mathtt{add}(x,y,z)$.

Thus$_0$: $\mathbf{S}\,\mathtt{nat}(x) \vee \mathbf{S}\,\mathtt{nat}(z) \rightarrow \mathbf{T}\,\mathtt{add}(x,y,z)$. □

Each of the three statements is proved in a fraction of a second by Vampire. Our compiler generates an instance of the induction axiom for each lemma and not for the theorem. For instance, here is the first conjecture and its induction axiom:

```
fof('lemma-(add:term:1)',conjecture,
  ! [Xx,Xy,Xz] : (nat_succeeds(Xx) => add_terminates(Xx,Xy,Xz))).

fof(induction,axiom,(
! [Xx] :
  ((? [Xx2] : (Xx = s(Xx2) & (nat_succeeds(Xx2)
                     & ! [Xy,Xz] : add_terminates(Xx2,Xy,Xz)))
    | Xx = '0')
  => ! [Xy,Xz] : add_terminates(Xx,Xy,Xz))
=>
! [Xx] : (nat_succeeds(Xx) => ! [Xy,Xz] : add_terminates(Xx,Xy,Xz)))).
```

4 Experimental Results

We applied the schema explained in the previous sections to various libraries available with LPTP which we summarize now. The library `nat` defines some basic Peano relations with the expected properties. The library `ack` defines the relational version of the Ackermann function with three properties (see below). The library `gcd` defines a version of the greatest common divisor relation, with its full correctness proof. The library `int` defines integers. The library `list` proposes some elementary relations about lists with their properties. The library `suffix` defines two versions of the sublist relation, one as the prefix of a suffix, the other as the suffix of a prefix, and shows that the two versions are equivalent w.r.t. termination, success and failure. Similarly, the library `reverse` defines the two classical versions of the reverse relation, one with the append relation, the other with an accumulator and shows their full equivalence. The library `permutation` defines the permutation relation with some useful properties for the correctness proofs of the sorting algorithms defined in the libraries `sort` and `mergesort`. The library `taut` defines a tautology checker for propositional formulas, together with its correctness proof (see [27] for a detailed description).

How do we process such files? Given a program from the LPTP library, we first enumerate the requirements for trying to prove the properties listed in its associated LPTP proof file. Requirements are the logic program P and the associated LPTP proof file. If P depends on other logic programs, we must include them. If the associated LPTP proof file uses other proof files, we must include them as well. We assume there is no circularity such as assuming a lemma before trying to prove it. We use these requirements to build a target logic program P' and a target LPTP proof file. Then P' is compiled into the FOF version of IND(P'). Each fact (i.e., lemma, corollary or theorem) is compiled as a FOF conjecture (possibly with its induction axiom) and stored in a single file. Such file also contains the logic theory

IND(P') compiled as FOF axioms. Previously processed FOF conjectures are converted as FOF axioms as well. As a result, we produce as many FOF files as there are facts in the initial LPTP proof file. At last, both the E Theorem Prover and Vampire are applied to each FOF file with the commands `vampire -mode casc -m 16384 -cores 7 -t $TO $FILE` and `eprover -delete-bad-limit=2000000000 -definitional-cnf -s -auto-schedule=8 -proof-object -cpu-limit=$TO $FILE`.

We gather the results in Table 1. The first column gives the library names. The second column gives the number of (lemmas/corollaries/theorems) of the associated proof file. The remaining nine columns can be divided in three groups. On a MacBook Pro, 8 cores, M2, 24 GB, macOS Sonoma, the first group gives the success rate for a 1 second timeout for the E prover (column E-1s), Vampire (column V-1s) and for the combination of the two provers (column EV-1s). The second group (resp. third group) gives the success rate for a timeout of 10 seconds (resp. 60 seconds).

lib	#	E-1s	V-1s	EV-1s	E-10s	V-10s	EV-10s	E-60s	V-60s	EV-60s
nat	91	70%	88%	88%	76%	95%	95%	78%	97%	97%
gcd	11	45%	45%	45%	45%	45%	45%	45%	45%	45%
ack	3	33%	33%	33%	33%	33%	33%	33%	33%	33%
int	67	76%	82%	87%	79%	88%	90%	79%	91%	91%
list	84	75%	94%	94%	80%	96%	96%	81%	99%	99%
suffix	31	94%	100%	100%	94%	100%	100%	97%	100%	100%
reverse	25	72%	88%	88%	84%	100%	100%	84%	100%	100%
permut.	42	48%	71%	71%	60%	79%	81%	62%	86%	86%
sort	42	45%	62%	62%	50%	74%	74%	55%	76%	76%
merges.	24	79%	88%	88%	79%	92%	92%	79%	100%	100%
taut	43	65%	81%	81%	70%	84%	84%	74%	84%	84%

Table 1: Experimental Evaluation

Let us comment these results. The `gcd` Prolog file contains a mutually recursive definition for the predicates `gcd/3` and `gcd_leq/3`. Proving properties of such definitions is currently out of scope of our translation schema.

The `ack` proof file contains the following three properties. The first one is successfully checked. The last two ones cannot be proved with our simplified induction schema. Indeed, the LPTP proofs use an induction inside the top level induction, which is out of scope of our translation schema.

Lemma [*ackermann:types*] $\forall m,n,k\,(\mathbf{S}\,\mathtt{ackermann}(m,n,k) \wedge \mathbf{S}\,\mathtt{nat}(n) \rightarrow \mathbf{S}\,\mathtt{nat}(k))$.

Lemma [*ack:existence*] $\forall m,n\,(\mathbf{S}\,\mathtt{nat}(m) \wedge \mathbf{S}\,\mathtt{nat}(n) \rightarrow \exists k\,\mathbf{S}\,\mathtt{ackermann}(m,n,k))$.

Lemma [*ack:termination*] $\forall m,n,k\,(\mathbf{S}\,\mathtt{nat}(m) \wedge \mathbf{S}\,\mathtt{nat}(n) \rightarrow \mathbf{T}\,\mathtt{ackermann}(m,n,k))$.

5 Related Work

There is quite a few Prolog verification frameworks, see e.g. [7, 10, 2, 24] and more recently [8]. Most of them aim at *paper and pencil* proofs. Although they may offer interesting and elegant methods, the validity of the proofs relies on the usual mathematical writing in natural language, and proofs are not

automatically checked. In our opinion, writing and verifying such hand-written proofs can be a time consuming and error-prone process compared to a push-button approach as the one we present here.

Recently, some quite interesting works have been reported on including datatypes, taking into account the acyclicity of their values, and induction in modern first-oder theorem provers, see, e.g., [4, 12]. We have not yet tested these extensions within our framework.

For Answer Set Programming (a declarative specification language with a Prolog syntax, oriented towards knowledge representation and search problems), [9] describes an approach toward verification in which Vampire checks the equivalence of Answer Set programs.

Some programming languages include automated verification tools *by design*. For example, Dafny [17] makes heavy use of SMT solving. The Why3 system [11] allows to export verification conditions to many automatic and interactive theorem provers.

An earlier account of the integration of automated and interactive theorem proving is described in [1]. As already announced in the introduction, most interactive theorem provers now include the possibility to run some automated theorem provers. Starting with Isabelle, [20, 23, 3, 22], *hammers* can be found in e.g., ACL2, [14], Coq, [6] and Lean, [18].

6 Conclusion

Let us recall the questions of the introduction and propose our answers after this experiment:

- Can we also use the TPTP FOF *Esperanto* to formulate the logic theory Stärk associates to a logic program? Yes. One axiom schema was not implemented: Axiom 3 which forbids rational terms. Another one was partially implemented: Axiom 9 for induction. Actually an inductive argument inside an inductive proof is not possible with our approach. We lose precision but in both cases we stay sound.
- Can we use *off-the-shelf* TPTP provers and obtain automatic proofs in reasonable time? Yes. We use Vampire and the E prover with their most basic options, essentially a timeout. Although Vampire seems to find refutations faster, the E prover can sometimes find proofs while Vampire cannot conclude within the time limit. Hence the two provers are complementary. For the moment, we did not try advanced features offered by the provers like the one proposed in [15].
- Can we get an acceptable success rate with such an approach? Yes. With the E prover and Vampire running in parallel, the average success rate we get from our benchmark is about 83% for a one minute timeout on a standard laptop.

Compared to the efforts one spends while manually, laboriously elaborating certain proofs with an interactive theorem prover, the use of state of the art automated theorem provers is clearly a time-saver. We did not expect such a good success rate for this first experiment. We think there are various reasons that can explain it. Clearly, the computing power of our current laptops is huge and automated theorem provers have been largely improved. Also, thanks to Stärk's ideas, the clean and simple semantics of both the pure subset of Prolog targeted by LPTP and the LPTP specification language – essentially first-order logic – implies a straightforward translation to FOF. Last but not least, Stärk's art of proving, by slicing the proofs of the LPTP library properties into manageable lemmas, certainly has an impact on the success rate we obtain.

Finally, there is room for improvement of the presented work, which can be considered as a first approach towards a hammer for LPTP according to [3]. The first step of a hammer – the *premise selector*, which selects subparts of the LPTP library potentially useful for a proof – and the third step – the *proof*

reconstruction module, which rewrites the proof found by the automatic prover in the LPTP proof format – are yet to be investigated.

Acknowledgements. We thank Manuel Hermenegildo, Daniel Jurjo, Pedro López-Garcia, and Jose Morales for stimulating discussions about LPTP, Geoff Sutcliffe for his help with TPTP and the reviewers for their constructive comments.

References

[1] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reif, G. Schellhorn & P. Schmitt (1998): *Integrating Automated and Interactive Theorem Proving. Automated Deduction — A Basis for Applications: Volume II: Systems and Implementation Techniques*, pp. 97–116. Springer.

[2] K. R. Apt & E. Marchiori (1994): *Reasoning About Prolog Programs: From Modes Through Types to Assertions*. *Formal Aspects Comput.* 6(6A), pp. 743–765, doi:10.1007/BF01213601.

[3] J. C. Blanchette, C. Kaliszyk, L. C. Paulson & J. Urban (2016): *Hammering towards QED*. *J. Formaliz. Reason.* 9(1), pp. 101–148, doi:10.6092/ISSN.1972-5787/4593.

[4] J. C. Blanchette, N. Peltier & S. Robillard (2018): *Superposition with Datatypes and Codatatypes*. In D. Galmiche, S. Schulz & R. Sebastiani, editors: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, *Lecture Notes in Computer Science* 10900, Springer, pp. 370–387, doi:10.1007/978-3-319-94205-6_25.

[5] K. L. Clark (1977): *Negation as Failure*. In H. Gallaire & J. Minker, editors: *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, Plemum Press, New York, pp. 293–322, doi:10.1007/978-1-4684-3384-5_11.

[6] L. Czajka, B. Ekici & C. Kaliszyk (2018): *Concrete Semantics with Coq and CoqHammer*. In F. Rabe, W. M. Farmer, G. O. Passmore & A. Youssef, editors: *CICM*, *LNCS* 11006, Springer, pp. 53–59, doi:10.1007/978-3-319-96812-4_5.

[7] P. Deransart (1993): *Proof Methods of Declarative Properties of Definite Programs*. *Theor. Comput. Sci.* 118(2), pp. 99–166, doi:10.1016/0304-3975(93)90107-5.

[8] W. Drabent (2016): *Correctness and Completeness of Logic Programs*. *ACM Trans. Comput. Log.* 17(3), p. 18, doi:10.1145/2898434.

[9] J. Fandinno, V. Lifschitz, P. Lühne & T. Schaub (2020): *Verifying Tight Logic Programs with Anthem and Vampire*. *Theory Pract. Log. Program.* 20(5), pp. 735–750, doi:10.1017/S1471068420000344.

[10] G. Ferrand & P. Deransart (1993): *Proof Method of Partial Correctness and Weak Completeness for Normal Logic Programs*. *J. Log. Program.* 17(2/3&4), pp. 265–278, doi:10.1016/0743-1066(93)90033-D.

[11] J.-C. Filliâtre & A. Paskevich (2013): *Why3 - Where Programs Meet Provers*. In M. Felleisen & P. Gardner, editors: *ESOP*, *LNCS* 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[12] M. Hajdú, P. Hozzová, L. Kovács & A. Voronkov (2021): *Induction with Recursive Definitions in Superposition*. In: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, IEEE, pp. 1–10, doi:10.34727/2021/ISBN.978-3-85448-046-4_34.

[13] ISO/IEC 13211-1 (1995): *Information Technology – Programming Languages – Prolog – Part 1: General Core*.

[14] S. J. C. Joosten, C. Kaliszyk & J. Urban (2014): *Initial Experiments with TPTP-style Automated Theorem Provers on ACL2 Problems*. In F. Verbeek & J. Schmaltz, editors: *International Workshop on ACL2*, *EPTCS* 152, pp. 77–85, doi:10.4204/EPTCS.152.6.

[15] L. Kovács, S. Robillard & A. Voronkov (2017): *Coming to terms with quantified reasoning*. In G. Castagna & A. D. Gordon, editors: *POPL 2017*, ACM, pp. 260–270, doi:10.1145/3009837.3009887.

[16] L. Kovács & A. Voronkov (2013): *First-Order Theorem Proving and Vampire*. In N. Sharygina & H. Veith, editors: *CAV 2013*, *LNCS* 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[17] K. R. M. Leino (2012): *Developing Verified Programs with Dafny*. In R. Joshi, P. Müller & A. Podelski, editors: *VSTTE*, *LNCS* 7152, Springer, p. 82, doi:10.1007/978-3-642-27705-4_7.

[18] P. Lippe (2019): *Lean Hammer*. `https://github.com/phlippe/Lean_hammer`. Accessed: 2025-04.

[19] J. W. Lloyd (1987): *Foundations of Logic Programming, 2nd Edition*. Springer, doi:10.1007/978-3-642-83189-8.

[20] J. Meng & L. C. Paulson (2004): *Experiments on Supporting Interactive Proof Using Resolution*. In D. A. Basin & M. Rusinowitch, editors: *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, *Lecture Notes in Computer Science* 3097, Springer, pp. 372–384, doi:10.1007/978-3-540-25984-8_28.

[21] F. Mesnard, T. Marianne & É. Payet (2024): *Automated Theorem Proving for Prolog Verification*. In N. S. Bjørner, M. Heule & A. Voronkov, editors: *LPAR 2024 Complementary Volume*, *Kalpa Publications in Computing* 18, EasyChair, pp. 137–151, doi:10.29007/C25R.

[22] L. C. Paulson (2022): *Sledgehammer: some history, some tips*. `https://lawrencecpaulson.github.io/2022/04/13/Sledgehammer.html`. Accessed: 2025-04.

[23] L. C. Paulson & J. C. Blanchette (2010): *Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In G. Sutcliffe, S. Schulz & E. Ternovska, editors: *IWIL*, *EPiC Series in Computing* 2, EasyChair, pp. 1–11, doi:10.29007/36DT.

[24] D. Pedreschi & S. Ruggieri (1999): *Verification of Logic Programs*. *J. Log. Program.* 39(1-3), pp. 125–176, doi:10.1016/S0743-1066(98)10035-3.

[25] S. Schulz, S. Cruanes & P. Vukmirovic (2019): *Faster, Higher, Stronger: E 2.3*. In P. Fontaine, editor: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, *Lecture Notes in Computer Science* 11716, Springer, pp. 495–507, doi:10.1007/978-3-030-29436-6_29.

[26] R. F. Stärk (1995): *First-order theories for pure Prolog programs with negation*. *Arch. Math. Log.* 34(2), pp. 113–144, doi:10.1007/BF01270391.

[27] R. F. Stärk (1996): *Total Correctness of Logic Programs: A Formal Approach*. In R. Dyckhoff, H. Herre & P. Schroeder-Heister, editors: *ELP'96*, *LNCS* 1050, Springer, pp. 237–254, doi:10.1007/3-540-60983-0_17.

[28] R. F. Stärk (1997): *Formal Verification of Logic Programs: Foundations and Implementation*. In S. I. Adian & A. Nerode, editors: *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97, Yaroslavl, Russia, July 6-12, 1997, Proceedings*, *Lecture Notes in Computer Science* 1234, Springer, pp. 354–368, doi:10.1007/3-540-63045-7_36.

[29] R. F. Stärk (1998): *The Theoretical Foundations of LPTP (A Logic Program Theorem Prover)*. *J. Log. Program.* 36(3), pp. 241–269, doi:10.1016/S0743-1066(97)10013-9.

[30] G. Sutcliffe (2017): *The TPTP Problem Library and Associated Infrastructure - From CNF to TH0, TPTP v6.4.0*. *J. Autom. Reason.* 59(4), pp. 483–502, doi:10.1007/S10817-017-9407-7.

[31] G. Sutcliffe (2023): *The logic languages of the TPTP world*. *Log. J. IGPL* 31(6), pp. 1153–1169, doi:10.1093/JIGPAL/JZAC068.