Automated Theorem Proving for Prolog Verification

Fred Mesnard, Thierry Marianne, and Étienne Payet

LIM, université de La Réunion, France
{frederic.mesnard,thierry.marianne,etienne.payet}@univ-reunion.fr

Abstract

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 90'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 explicit 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 90's, Robert F. Stärk defined a framework for Prolog verification [23, 25]. He considered a subset of ISO-Prolog [11]: 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 wrt. the clauses defining predicates, considered as inductive definitions. 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[1] 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.

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 skills. On the other hand, during the last three decades,

[1]available at https://github.com/FredMesnard/lptp

the increase in computing power and the advances in automated theorem proving have been notable. For instance, the TPTP (Thousands of Problems for Theorem Provers) [26] 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 [22] and Vampire [14] are two powerful freely available automated theorem provers, performing very well in many international competitions over the years. Interactive theorem prover implementers starting with [20, 3] were able to take advantage of these progress by implementing so-called *hammers* for their tools.

This evolution raises the following questions: can we also use the TPTP *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 [27]) 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[2].

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. Section 4 explains how to compile function and predicate definitions which may appear in a proof 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:

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 [17]). For sake of conciseness, we do not consider built-in predicates (see [25] for a full treatment) other than the equality `=/2`. We start with $\mathcal{L}$, the first-order language associated to P. The *goals* of $\mathcal{L}$ are:

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

where s and t are two terms, x is a variable and A is an atomic goal. The goals of $\mathcal{L}$ have the operational semantics specified by ISO-Prolog [11] assuming the occurs check.

[2]https://github.com/atp-lptp/automated-theorem-proving-for-prolog-verification

$\hat{\mathcal{L}}$ is the specification language of LPTP. For each user-defined predicate symbol R, $\hat{\mathcal{L}}$ 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. $\hat{\mathcal{L}}$ also contains a unary constraint for groundness gr, expressing that its argument is ground. The *formulas* of $\hat{\mathcal{L}}$ are:

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

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

For any of the user-defined logic procedure R in a logic program P, $D_R^P(\overrightarrow{x})$ denotes its Clark's *if-and-only-if* completed definition [4, 17].

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

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

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

Finally, we add the definition of gr, which is a constraint of the specification language and is needed for defining $\mathbf{T}\backslash\texttt{+}G$:

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

3 Compiling LPTP axioms to FOF

With LPTP, we prove properties of a logic program P wrt. its *inductive extension* IND(P) which includes Clark's completion [4] and induction along the definition of the predicates. Stärk shows that the inductive extension is always consistent and proves various correctness and completeness results wrt. the operational semantics of Prolog in [25]. The first-order theory IND(P) (cf. [25], pp. 253–254) is defined by nine kinds of axioms which we describe now, along with their translation in FOF. We omit the fixed point axioms for builtins.

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. $f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \rightarrow x_i = y_i$ [if f is n-ary and $1 \leq i \leq n$]
2. $f(x_1, \dots, x_n) \neq g(y_1, \dots, y_m)$ [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 rational trees. Note that it is an axiom schema, i.e., an infinite set of first order axioms. We will omit it and it can be a source of imprecision, but we stay sound. Here is the FOF version:

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

Axioms for gr/1

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

Actually, LPTP deals with *non-ground* terms, as any ISO-Prolog processor does. LPTP 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 [4] of the user-defined procedure $R(\vec{x})$ in the logic program P. In the previous section, we saw how to apply the operator $\mathbf{S}$, $\mathbf{F}$, and $\mathbf{T}$ 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 $\forall\vec{x}[R^s(\vec{x}) \to \phi(\vec{x})]$, where $R(\vec{x})$ is a user-defined procedure and $\phi(\vec{x})$ an $\hat{\mathcal{L}}$ -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 [25]. 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 [23] 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 $\hat{\mathcal{L}}$-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 $\forall\vec{x}(R^s(\vec{x}) \to \phi(\vec{x}))$.
Let $closed(\phi(\vec{x})/R)$ be the formula obtained from $\forall\vec{x}(\mathbf{S}D_R^P(\vec{x}) \to R^s(\vec{x}))$ by replacing

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

Then the induction axiom is the following formula:

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

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 $sub(\phi(\overrightarrow{x})/R)$ of axiom 9. So $R \equiv$ `nat`, $R^s \equiv$ `nat_succeeds` and $\phi(\overrightarrow{x}) \equiv \texttt{add\_succeeds}(x, 0, x)$.

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

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

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

Summarizing, in FOF, associated to 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[3] to the E prover or to Vampire. Both systems will find a refutation in a fraction of a second[4] 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 $\texttt{add}(s^n(0), 0, s^n(0))$, 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 `:-` $\texttt{add}(s^n(0), 0, s^n(0))$. 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)))])).
```

[3]E.g., in the terminal `eprover --auto test.fof` and `vampire test.fof`

[4]0.05s for Vampire, 0.017s for E on a MacBook Air, Apple M2, 16Go, macOS Sonoma.

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 predicate. 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 $\mathtt{add}(x, y, z)$ then $\mathtt{add}(y, x, z)$. 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 to 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*] $\forall x, y, z\,(\mathbf{S}\,\texttt{nat}(x) \to \mathbf{T}\,\texttt{add}(x, y, z))$. **Proof.**

Induction$_0$: $\forall x\,(\mathbf{S}\,\texttt{nat}(x) \to \forall y, z\ \mathbf{T}\,\texttt{add}(x, y, z))$.
  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*] $\forall x, y, z\,(\mathbf{S}\,\texttt{nat}(z) \to \mathbf{T}\,\texttt{add}(x, y, z))$. **Proof.** Similar. □

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

Assumption$_0$: $\mathbf{S}\,\texttt{nat}(x) \vee \mathbf{S}\,\texttt{nat}(z)$.
  Case$_1$: $\mathbf{S}\,\texttt{nat}(x)$. $\mathbf{T}\,\texttt{add}(x, y, z)$ by Lemma 1 [*add:term:1*].
  Case$_1$: $\mathbf{S}\,\texttt{nat}(z)$. $\mathbf{T}\,\texttt{add}(x, y, z)$ by Lemma 2 [*add:term:3*].
  Hence$_1$, in all cases: $\mathbf{T}\,\texttt{add}(x, y, z)$.
Thus$_0$: $\mathbf{S}\,\texttt{nat}(x) \vee \mathbf{S}\,\texttt{nat}(z) \to \mathbf{T}\,\texttt{add}(x, y, z)$. □

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 Dealing with definitions in LPTP proof files

The specification language $\hat{\mathcal{L}}$ of LPTP can be extended by new function and predicate symbols, which can be handy while formalizing properties. Although Stärk often uses this feature in his proofs (see for instance the LPTP library `nat`), one may safely skip this section on first reading. Function definitions have the form

```
:- definition_fun(f, n,
  all [x_1,...,x_n,y]: δ(x_1,...,x_n) => (f(x_1,...,x_n) = y <=> γ(x_1,...,x_n,y)),
  existence  by fact(reference_1),
  uniqueness by fact(reference_2)
).
```

where f is the defined symbol, n is its number of arguments and δ (resp. γ) is the domain (resp. graph) of the function. Moreover, *reference*$_1$ refers to a fact (i.e., LPTP lemma, corollary or theorem) from which the formula

$$\texttt{all [}x_1,\ldots,x_n\texttt{]: } \delta(x_1,\ldots,x_n) \texttt{ => (ex } y\texttt{: } \gamma(x_1,\ldots,x_n,y)\texttt{)} \quad (1)$$

is immediately derivable and *reference*$_2$ to a fact from which the formula

$$\texttt{all [}x_1,\ldots,x_n,y,z\texttt{]: (}\delta(x_1,\ldots,x_n)\texttt{ \& }\gamma(x_1,\ldots,x_n,y)\texttt{ \& }\gamma(x_1,\ldots,x_n,z)\texttt{) => } y \texttt{ = } z \quad (2)$$

is immediately derivable. On the other hand, predicate definitions have simply the form

```
:- definition_pred(R, n, all [x_1,...,x_n]: R(x_1,...,x_n) <=> φ).
```

where the free variables of φ are contained in $x_1,\ldots,x_n$.

4.1 An example of a function definition

Here is a manual LPTP proof in its TEX version of a fact that exactly matches formula (1), where $\delta \equiv$ `nat_succeeds` and $\gamma \equiv$ `add_succeeds`:

Lemma [*add:existence*] $\forall x, y\, (\mathsf{S}\, \mathtt{nat}(x) \to \exists z\ \mathsf{S}\, \mathtt{add}(x, y, z))$.

Proof.
Induction$_0$: $\forall x\, (\mathsf{S}\, \mathtt{nat}(x) \to \forall y\, \exists z\ \mathsf{S}\, \mathtt{add}(x, y, z))$.
  Hypothesis$_1$: none. $\mathsf{S}\, \mathtt{add}(\mathtt{0}, y, y)$. $\exists z\ \mathsf{S}\, \mathtt{add}(\mathtt{0}, y, z)$.
  Conclusion$_1$: $\forall y\, \exists z\ \mathsf{S}\, \mathtt{add}(\mathtt{0}, y, z)$.
  Hypothesis$_1$: $\forall y\, \exists z\ \mathsf{S}\, \mathtt{add}(x, y, z)$ and $\mathsf{S}\, \mathtt{nat}(x)$. $\exists z\ \mathsf{S}\, \mathtt{add}(x, y, z)$.
    Let$_2$ z with $\mathsf{S}\, \mathtt{add}(x, y, z)$. $\mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, \mathtt{s}(z))$.
    Thus$_2$: $\exists z\ \mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z)$. $\exists z\ \mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z)$.
  Conclusion$_1$: $\forall y\, \exists z\ \mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z)$.
Assumption$_0$: $\mathsf{S}\, \mathtt{nat}(x)$. $\forall y\, \exists z\ \mathsf{S}\, \mathtt{add}(x, y, z)$. $\exists z\ \mathsf{S}\, \mathtt{add}(x, y, z)$.
Thus$_0$: $\mathsf{S}\, \mathtt{nat}(x) \to \exists z\ \mathsf{S}\, \mathtt{add}(x, y, z)$. □

The corresponding FOF conjecture and its associated induction axiom are:

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

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

The conjecture is proved in less than a second by E and Vampire.
Here is also an LPTP proof of a fact of the form

$$\mathtt{all}\ [x_1, \ldots, x_n, y, z]\mathtt{:}\ (\gamma(x_1, \ldots, x_n, y)\ \mathtt{\&}\ \gamma(x_1, \ldots, x_n, z))\ \mathtt{=>}\ y\ \mathtt{=}\ z$$

(for the same γ as above) from which formula (2) is immediately derivable:

Lemma [*add:uniqueness*] $\forall x, y, z_1, z_2\, (\mathsf{S}\, \mathtt{add}(x, y, z_1) \wedge \mathsf{S}\, \mathtt{add}(x, y, z_2) \to z_1 = z_2)$. **Proof.**
Induction$_0$: $\forall x, y, z_1\, (\mathsf{S}\, \mathtt{add}(x, y, z_1) \to \forall z_2\, (\mathsf{S}\, \mathtt{add}(x, y, z_2) \to z_1 = z_2))$.
  Hypothesis$_1$: none.
    Assumption$_2$: $\mathsf{S}\, \mathtt{add}(\mathtt{0}, y, z_2)$. $\mathsf{D}\,\mathsf{S}\, \mathtt{add}(\mathtt{0}, y, z_2)$ by completion. $y = z_2$.
    Thus$_2$: $\mathsf{S}\, \mathtt{add}(\mathtt{0}, y, z_2) \to y = z_2$.
  Conclusion$_1$: $\forall z_2\, (\mathsf{S}\, \mathtt{add}(\mathtt{0}, y, z_2) \to y = z_2)$.
  Hypothesis$_1$: $\forall z_2\, (\mathsf{S}\, \mathtt{add}(x, y, z_2) \to z_1 = z_2)$ and $\mathsf{S}\, \mathtt{add}(x, y, z_1)$.
    Assumption$_2$: $\mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z_2)$. $\mathsf{D}\,\mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z_2)$ by completion.
      $\exists z_3\, (z_2 = \mathtt{s}(z_3) \wedge \mathsf{S}\, \mathtt{add}(x, y, z_3))$.
      Let$_3$ z_3 with $z_2 = \mathtt{s}(z_3) \wedge \mathsf{S}\, \mathtt{add}(x, y, z_3)$. $\mathsf{S}\, \mathtt{add}(x, y, z_3) \to z_1 = z_3$. $z_1 = z_3$.
      Thus$_3$: $\mathtt{s}(z_1) = z_2$.
    Thus$_2$: $\mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z_2) \to \mathtt{s}(z_1) = z_2$.
  Conclusion$_1$: $\forall z_2\, (\mathsf{S}\, \mathtt{add}(\mathtt{s}(x), y, z_2) \to \mathtt{s}(z_1) = z_2)$.

Assumption$_0$: $\mathsf{S}\,\mathtt{add}(x,y,z_1) \land \mathsf{S}\,\mathtt{add}(x,y,z_2)$. $\forall z_2\,(\mathsf{S}\,\mathtt{add}(x,y,z_2) \to z_1 = z_2)$.
$\mathsf{S}\,\mathtt{add}(x,y,z_2) \to z_1 = z_2$. $z_1 = z_2$.
Thus$_0$: $\mathsf{S}\,\mathtt{add}(x,y,z_1) \land \mathsf{S}\,\mathtt{add}(x,y,z_2) \to z_1 = z_2$. □

The corresponding FOF conjecture and its associated induction axiom are:

```
fof('lemma-(add:uniqueness)',conjecture,
  ! [Xx,Xy,Xz1,Xz2] : ((add_succeeds(Xx,Xy,Xz1) & add_succeeds(Xx,Xy,Xz2))
                       => Xz1 = Xz2)).

fof(induction,axiom,
(! [Xx,Xy,Xz1] :
  ((? [Xx4] : ? [Xx5] :
    (Xx = s(Xx4) & (Xz1 = s(Xx5)
     & (add_succeeds(Xx4,Xy,Xx5)
        & ! [Xz2] : (add_succeeds(Xx4,Xy,Xz2) => Xx5 = Xz2))))
  | (Xx = '0' & Xz1 = Xy))
  => ! [Xz2] : (add_succeeds(Xx,Xy,Xz2) => Xz1 = Xz2))
=>
  ! [Xx,Xy,Xz1] : (add_succeeds(Xx,Xy,Xz1) =>
    ! [Xz2] : (add_succeeds(Xx,Xy,Xz2) => Xz1 = Xz2)))).
```

Neither Vampire neither E is able to prove the conjecture in reasonnable time. Nonetheless, we have the LPTP proof, so we can define the function `@+` that takes two arguments:

```
:- definition_fun(@+, 2,
 all [x,y,z]: succeeds nat(?x) => (?x @+ ?y = ?z <=> succeeds add(?x,?y,?z)),
 existence  by lemma(add:existence),
 uniqueness by lemma(add:uniqueness)
).
```

The TEX version of this definition produced by LPTP is:

Definition [*@+/2*] $\forall x,y,z\,(\mathsf{S}\,\mathtt{nat}(x) \to (x + y = z \leftrightarrow \mathsf{S}\,\mathtt{add}(x,y,z)))$.

The domain of `@+` is $\delta \equiv$ `nat_succeeds` and its graph is $\gamma \equiv$ `add_succeeds`. We note that for all terms y, `add(0,`y`,`y`)` is true (see the definition of `add`) so, by the uniqueness property, we have `0 @+` y `=` y. Hence the following manual LPTP proof:

Corollary [*add:zero*] $\forall y\, 0 + y = y$. **Proof.** $0 + y = y$ by Definition @+/2 [*@+/2*]. □

In the FOF version, we handle the definition of `@+` as an axiom:

```
fof('@+',axiom,
  ! [Xx,Xy,Xz] : (nat_succeeds(Xx) =>
                 ('@+'(Xx,Xy) = Xz <=> add_succeeds(Xx,Xy,Xz)))).
```

We also add the existence property and the uniqueness property as axioms. Our conjecture is:

```
fof('corollary-(add:zero)', conjecture, ! [Xy] : '@+'('0',Xy) = Xy).
```

We gather all the axioms and the conjecture in a file that we submit to E and Vampire. Both systems find a refutation in a fraction of a second.

4.2 An example of a predicate definition

We define the predicate `even` using the function `@+` introduced above:

```
:- definition_pred(even, 1, all [x]: even(?x) <=> (ex y:  ?y @+ ?y = ?x)).
```

The TEX version produced by LPTP is:

Definition [*even/1*] $\forall x\,(\mathit{even}(x) \leftrightarrow \exists y\, y + y = x)$.

We can show that `even` is not empty. The manual LPTP proof is:

Lemma [*even:non_empty*] $\exists x\ \mathit{even}(x)$. **Proof.** $\mathsf{S}\,\mathtt{add}(\mathtt{0},\mathtt{0},\mathtt{0})$.
$\mathtt{0} + \mathtt{0} = \mathtt{0}$ by Definition @+/2 [*@+/2*]. $\mathit{even}(\mathtt{0})$ by Definition even/1 [*even/1*]. □

In the FOF version, we handle the predicate definition as an axiom and we get:

```
fof(even,axiom,
  ! [Xx] : (even_succeeds(Xx) <=> ? [Xy] : ('@+'(Xy,Xy) = Xx))).

fof('lemma-(even:non_empty)', conjecture, ? [Xx] : even_succeeds(Xx)).
```

We gather all the axioms (including those for `@+`) and the conjecture in a file that we submit to E and Vampire. Both systems find a refutation in a fraction of a second.

5 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 `gcd` defines a version of the greatest common divisor relation, with its full correctness proof. 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 wrt. 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 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 [24] 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 predefined time limits.

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 Air, Apple M2, 16Go, 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	54%	72%	78%	58%	77%	80%	61%	81%	85%
gcd	11	45%	45%	45%	45%	45%	45%	45%	45%	45%
list	84	56%	73%	83%	67%	87%	90%	68%	89%	92%
suffix	31	74%	81%	93%	81%	94%	97%	81%	97%	100%
reverse	25	52%	64%	64%	64%	80%	84%	68%	84%	88%
permut.	42	45%	50%	55%	52%	59%	64%	52%	64%	67%
sort	42	33%	33%	40%	43%	57%	57%	48%	59%	62%
merges.	24	50%	71%	71%	62%	79%	79%	67%	79%	79%
taut	43	0%	67%	67%	65%	70%	70%	65%	74%	74%

Table 1: Success rate

6 Related Work

There is quite a few Prolog verification frameworks, see e.g. [6, 9, 2, 21] and more recently [7]. 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 process compared to a push-button approach as the one we present here. For Answer Set Programming (a declarative specification language with a Prolog syntax, oriented towards knowledge representation and search problems), [8] 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 [15] makes heavy use of SMT solving and Why3 [10] 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[18, 20, 3, 19], *hammers* can be found in e.g., ACL2 [12], Coq [5] and Lean [16].

7 Conclusion

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

- Can we also use the TPTP *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 loose 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 regularly 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 [13].

- 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 77% for a one minute timeout on a standard laptop, which we find quite acceptable.

Compared to the efforts one spends while manually, laboriously elaborating certain proofs, such a tool 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. Stärk's art of proving, by slicing the proofs of the LPTP library properties into manageable lemmas, may also have an impact.

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]. In particular, the first step – the *premise selector*, which could select subparts of the LPTP library potentially useful for a proof – and the third step – the *proof reconstruction module*, which could rewrite 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 many discussions about the LPTP system.

References

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

[2] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[3] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016.

[4] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[5] L. Czajka, B. Ekici, and C. Kaliszyk. Concrete semantics with Coq and CoqHammer. In F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, editors, *CICM*, volume 11006 of *LNCS*, pages 53–59. Springer, 2018.

[6] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, pages 99–166, 1993.

[7] W. Drabent. Correctness and completeness of logic programs. *ACM Trans. Comput. Log.*, 17(3):18, 2016.

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

[9] G. Ferrand and P. Deransart. Proof method of partial correctness and weak completeness for normal logic programs. *J. Log. Program.*, 17(2/3&4):265–278, 1993.

[10] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

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

[12] S. J. C. Joosten, C. Kaliszyk, and J. Urban. Initial experiments with TPTP-style automated theorem provers on ACL2 problems. In F. Verbeek and J. Schmaltz, editors, *International Workshop on ACL2*, volume 152 of *EPTCS*, pages 77–85, 2014.

[13] L. Kovács, S. Robillard, and A. Voronkov. Coming to terms with quantified reasoning. In G. Castagna and A. D. Gordon, editors, *POPL 2017*, pages 260–270. ACM, 2017.

[14] L. Kovács and A. Voronkov. First-order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[15] K. R. M. Leino. Developing Verified Programs with Dafny. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE*, volume 7152 of *LNCS*, page 82. Springer, 2012.

[16] P. Lippe. Lean Hammer. `https://github.com/phlippe/Lean_hammer`, 2019. Accessed: 2024-01.

[17] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[18] J. Meng and L. C. Paulson. Experiments on supporting interactive proof using resolution. In D. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 372–384. Springer, 2004.

[19] L. C. Paulson. Sledgehammer: some history, some tips. `https://lawrencecpaulson.github.io/2022/04/13/Sledgehammer.html`, 2022. Accessed: 2024-02-01.

[20] L. C. Paulson and J. C. Blanchette. Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *IWIL*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[21] D. Pedreschi and S. Ruggieri. Verification of Logic Programs. *J. Log. Program.*, 39(1-3):125–176, 1999.

[22] S. Schulz, S. Cruanes, and P. Vukmirović. Faster, higher, stronger: E 2.3. In P. Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

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

[24] R. F. Stärk. Total correctness of logic programs: A formal approach. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *ELP'96*, volume 1050 of *LNCS*, pages 237–254. Springer, 1996.

[25] R. F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *Journal of Logic Programming*, 36(3):241–269, 1998.

[26] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[27] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.