Formal verification of logic programs: foundations and implementation

Robert F. Stärk

Institute of Informatics, University of Fribourg
Rue Faucigny 2, CH–1700 Fribourg, Switzerland
Email: ⟨robert.staerk@unifr.ch⟩

Abstract. We present the theoretical foundations of LPTP, a logic program theorem prover implemented in Prolog by the author. LPTP is an interactive theorem prover in which one can prove termination and correctness properties of pure Prolog programs that contain negation and built-in predicates like `is`/2 and `call`/n. The largest program that has been verified using LPTP is 635 lines long including its specification. The full formal correctness proof is 13128 lines long (133 pages). The formal theory underlying LPTP is the inductive extension of pure Prolog programs. This is a first-order theory that contains induction principles corresponding to the definition of the predicates in the program plus appropriate axioms for built-in predicates.

1 Introduction

There are several reasons that we have implemented an interactive theorem prover for the verification of pure Prolog programs. First of all, we wanted to show that results of [8,10,11] about the foundations of logic programming are not only of theoretical interest. In the spirit of Apt [1] we wanted to show that the results can be extended to a rather large subset of Prolog. Secondly, we believe that if computer programs become bigger and more complex, it will be inevitable that parts of it have to be formally verified. This is one possible way to ensure that they work as they are supposed to do.

Why Prolog programs and not imperative programs? When we start to reason about imperative programs, then soon we are on the low level where the state of the system is given by the contents of the variables together with a pointer to the program code that shows where the execution is at the moment. Formulas describing relations between such states can be very complex.

The advantage of Prolog programs is their high level of abstraction. For Prolog programs the state of a computation is given by the atom that is called, i.e. it is of the form $R(t_1, \ldots, t_n)$. The predicate R corresponds to a pointer into the code of the program and the terms $t_1, \ldots, t_n$ corresponds to the contents

Appeared in: S. Adian and A. Nerode, editors, *Logical Foundations of Computer Science LFCS '97 — Logic at Yaroslavl, Russia*. Springer Lecture Notes in Computer Science 1234, pages 354–368, 1997.

of the registers. If we ask, for example, what can be reached from the state $R(t_1, \dots, t_n)$, then we just have to match this atom against the clauses of the program. This shows, that for Prolog programs, formulas describing relations between states of the computation are very close to the syntax of the program.

There are other reasons in favor of Prolog. For example, Prolog is both, a specification language and a programming language. This means that one can write specifications in Prolog as well as efficient algorithms. As a consequence, correctness of Prolog programs can be reduced to equivalence of programs. One just has to show that the specification program computes the same relation as the implementation program.

Finally, there is a pragmatic reason to use Prolog. It is desirable that a theorem prover for a certain programming language is implemented in the programming language itself. For example, a theorem prover for Java programs should be implemented in Java. Since proof-checking and proof-search of our theorem prover is based on backtracking, it is natural to use a programming language that provides backtracking for free. And Prolog is such a language.

The plan of this paper is as follows. In Section 2 we define a subset of Prolog and describe a simple operational model for it. In Section 3 we introduce a first-order theory that is computationally adequate with respect to the operational model of Section 2. In Section 4 we give a short overview of LPTP. This is the logic program theorem prover based on the theoretical results of Sections 2 and 3. Section 5 finally illustrates how LPTP can be used to prove the correctness of an algorithm for inserting elements in AVL trees.

There are essential differences between this article and the first-order theory introduced in [12]. We are now working with general goals and not only with sequences of literals. This makes it possible to treat built-in predicates in a uniform and simple way. Mode assignments are no longer needed. Instead of it we have a unary predicate gr in the formal language and are now able to treat higher-order programs that use the `call`/n predicate.

2 Pure Prolog with negation and built-in predicates

Pure Prolog is a subset of Prolog. Which subset, however, is not always so clear. Apt, for example, uses in [1] the term “pure Prolog” for Horn clause programs when they are viewed as sequences of clauses. We use the term “pure Prolog” for a larger subset which we define below. We include negation and built-in predicates like `integer`/1, `is`/2, `<`/2, and `call`/n. Even the term decomposition predicates `functor`/3 and `arg`/3 are allowed. Predicates like `var`/1 which tests during run-time whether a variable is bound are not included in pure Prolog. Also the predicates `assert`/1 and `rectract`/1 which modify a program during run-time are forbidden in pure Prolog. The cut operator (`!`) does not belong to pure Prolog, since it destroys the lifting lemma. We assume that pure Prolog performs the occurs check during unification.

Let $\mathfrak{L}$ be a first-order language. The terms r, s, t of $\mathfrak{L}$ are built up as usual from variables x, y, z and constants c, d using function symbols f, g. The predicate symbols of $\mathfrak{L}$ are divided into user-defined and built-in predicates. If R is an n-ary predicate symbol of $\mathfrak{L}$ then the expression $R(t_1, \ldots, t_n)$ is an atomic goal of $\mathfrak{L}$. The atomic goal is called user-defined or built-in according to whether the predicates symbol R is user-defined or built-in. Atomic goals are denoted by A, B. The goals of $\mathfrak{L}$ are

$$E, F, G ::= \texttt{true} \mid \texttt{fail} \mid s = t \mid A \mid F \;\texttt{\&}\; G \mid F \;\texttt{or}\; G \mid \texttt{not}\, G \mid \texttt{some}\, x\, G.$$

The goal `true` is the goal that always succeeds; `fail` is the goal that always fails. Equations $s = t$ are solved by unification. Conjunction (`&`), disjunction (`or`), and negation (`not`) are written in Prolog as (F, G), $(F; G)$, and $\backslash\!+ G$. The meaning of these connectives will be explained below in terms of an operational semantics and later by a transformation of goals into formulas that contain the logical connectives $\land$, $\lor$ and $\neg$. Conjunction and disjunction are both associated to the right. The goal $E \;\texttt{\&}\; F \;\texttt{\&}\; G$, for example, stands for $E \;\texttt{\&}\; (F \;\texttt{\&}\; G)$. The empty conjunction is identified with `true`; the empty disjunction corresponds to the goal `fail`. A goal of the form $G_1 \;\texttt{\&}\; \ldots \;\texttt{\&}\; G_n \;\texttt{\&}\; \texttt{true}$ is called a query. It can be considered as a finite list of goals. We use $[G_1, \ldots, G_n]$ as an abbreviation for the query $G_1 \;\texttt{\&}\; \ldots \;\texttt{\&}\; G_n \;\texttt{\&}\; \texttt{true}$. The existential quantifier $\texttt{some}\, x\, G$ binds the variable x in the goal G. Existential quantification is implicit in Prolog. It is explicit in extensions of Prolog like Gödel [7] and Mercury [9].

Free and bound variables in goals are defined as usual. We use the vector notation $\boldsymbol{x}$ for a finite list $x_1, \ldots, x_n$. We write $G[\boldsymbol{x}]$ to express that all free variables of G are among the list $\boldsymbol{x}$; $G(\boldsymbol{x})$ may contain other free variables than $\boldsymbol{x}$. A goal or a term is called ground, if it does not contain free variables.

If A is a user-defined atomic goal and G is a goal then the expression $A \;\texttt{:-}\; G$ is called a clause with head A and body G. Let C be the clause

$$R(t_1[\boldsymbol{y}], \ldots, t_n[\boldsymbol{y}]) \;\texttt{:-}\; G[\boldsymbol{y}].$$

Then the *definition form* of C is defined to be the goal

$$D_C[x_1, \ldots, x_n] :\equiv \texttt{some}\, \boldsymbol{y}\, (x_1 = t_1[\boldsymbol{y}] \;\texttt{\&}\; \ldots \;\texttt{\&}\; x_n = t_n[\boldsymbol{y}] \;\texttt{\&}\; G[\boldsymbol{y}]),$$

where $\boldsymbol{y}$ are fresh variables. The *normal form* of C is the clause

$$R(x_1, \ldots, x_n) \;\texttt{:-}\; D_C[x_1, \ldots, x_n].$$

A program is a finite sequence of clauses. Let P be a program and R be a user-defined predicate symbol such that the clauses for R in P are $C_1, \ldots, C_m$ (in this order). Then the *definition form* of R with respect to P is defined to be the goal

$$D_R^P[\boldsymbol{x}] :\equiv D_{C_1}[\boldsymbol{x}] \;\texttt{or}\; \ldots \;\texttt{or}\; D_{C_m}[\boldsymbol{x}].$$

The *normalized definition* of R in P is the clause $R(\boldsymbol{x}) \;\texttt{:-}\; D_R^P[\boldsymbol{x}]$.

Both, the definition form of a clause and the definition form of a user-defined predicate are goals. Thus, from a theoretical point of view, one could as well define a logic program to be a function that assigns to every user-defined predicate symbol R a goal $D_R^P[\boldsymbol{x}]$ for some distinguished variables $\boldsymbol{x}$.

Without general goals, a theory of built-in predicates would be rather ad-hoc, since then every built-in predicate has to be treated in a different way. Using the concept of goals, built-in predicates can be treated in a uniform way. Built-in predicates can be modeled by a set $\mathbb{D}$ of built-in atomic goals and a function $\mathbb{B}$ from $\mathbb{D}$ into the set of goals such that the following two conditions are satisfied:

(D) If $A \in \mathbb{D}$ then $A\sigma \in \mathbb{D}$ for each substitution σ.
(B) $\mathbb{B}(A\sigma) = \mathbb{B}(A)\sigma$ for each $A \in \mathbb{D}$ and each substitution σ.

The idea is that $\mathbb{D}$ contains exactly those built-in atomic goals that can be evaluated and do not report an error message because of type violations or insufficient instantiation of arguments. The goal $\mathbb{B}(A)$ is then the result of the evaluation of A. In most cases the goal $\mathbb{B}(A)$ is either the goal `true` or the goal `fail`. In other cases $\mathbb{B}(A)$ can be an equation or a conjunction of equations.

$\mathbb{D}$ and $\mathbb{B}$ can also be understood as a foreign language interface. Given an atom A from the set $\mathbb{D}$ some code in a foreign language, like for example C, is called. $\mathbb{B}(A)$ is the result of the call. In order that Prolog can use the result, it must be converted into a goal.

There is another possibility is to think of a built-in predicate R. It is given by the (possibly infinite) collection of clauses $R(\boldsymbol{t})$ `:-` $\mathbb{B}(R(\boldsymbol{t}))$ for $R(\boldsymbol{t}) \in \mathbb{D}$.

Example 1. The predicates `integer`/1, `is`/2, `<`/2 and `call`/n satisfy conditions (D) and (B):

$$\texttt{integer}(t) \in \mathbb{D} :\Leftrightarrow t \text{ is ground.}$$

$$\mathbb{B}(\texttt{integer}(t)) := \begin{cases} \texttt{true}, & \text{if } t \text{ is an integer constant;} \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

$$(t_1 \ \texttt{is}\ t_2) \in \mathbb{D} :\Leftrightarrow t_2 \text{ is a ground arithmetic expression.}$$

$$\mathbb{B}(t_1 \ \texttt{is}\ t_2) := (t_1 = n), \text{ where } n \text{ is the value of } t_2 \text{ (as an integer).}$$

$$(t_1 \ \texttt{<}\ t_2) \in \mathbb{D} :\Leftrightarrow t_1 \text{ and } t_2 \text{ are ground arithmetic expressions.}$$

$$\mathbb{B}(t_1 \ \texttt{<}\ t_2) := \begin{cases} \texttt{true}, & \text{if the value of } t_1 \text{ is less than the value of } t_2; \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

$$\texttt{call}(s, \boldsymbol{t}) \in \mathbb{D} :\Leftrightarrow s \text{ is a constant.}$$

$$\mathbb{B}(\texttt{call}(s, \boldsymbol{t})) := s(\boldsymbol{t}).$$

Not all of the commonly used built-in predicates can be modeled this way. The `var`/1 predicate, for example, violates condition (B).

Example 2. The `var`/1 predicate violates (B):

$$\texttt{var}(t) \in \mathbb{D} :\Leftrightarrow t \text{ is a term.}$$

$$\mathbb{B}(\texttt{var}(t)) := \begin{cases} \texttt{true}, & \text{if } t \text{ is a variable;} \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

Some multi-purpose, built-in predicates like `functor`/3 have to be decomposed into their single components.

Example 3. The components of `functor`/3 are `decompose`/3 and `construct`/3:

$$\begin{array}{ll}
\texttt{decompose}(t_1,t_2,t_3)\in\mathbb{D} & :\Leftrightarrow t_1 \text{ is not a variable.}\\
\mathbb{B}(\texttt{decompose}(f(r_1,\dots,r_n),s,t)) & := \ (s=f \ \texttt{\&}\ t=\bar{n}).\\
\texttt{construct}(t_1,t_2,t_3)\in\mathbb{D} & :\Leftrightarrow t_2 \text{ is a constant, } 0\le t_3\le 255.\\
\mathbb{B}(\texttt{construct}(t,f,\bar{n})) & := \ \texttt{some}\, x_1,\dots,x_n\,(t=f(x_1,\dots,x_n)).\\
\texttt{arg}(t_1,t_2,t_3)\in\mathbb{D} & :\Leftrightarrow t_1 \text{ is an integer, } t_2 \text{ is not a variable.}\\
\mathbb{B}(\texttt{arg}(\bar{\imath},f(s_1,\dots,s_n),t)) & := \begin{cases} t=s_i, & \text{if } 1\le i\le n;\\ \texttt{fail}, & \text{otherwise.}\end{cases}
\end{array}$$

For example, we have

1. $\mathbb{B}(\texttt{decompose}(f(c,d),x,y)) = (x=f \ \texttt{\&}\ y=2)$,
2. $\mathbb{B}(\texttt{construct}(x,f,2)) = \texttt{some}\, y,z\,(x=f(y,z))$,
3. $\mathbb{B}(\texttt{arg}(f(c,d),2,x)) = (x=d)$.

Given a program P, the set $\mathbb{D}$ and the function $\mathbb{B}$, we can describe the evaluation of goals as a transition relation between states of a computation. States are defined in the following way:

An *environment* is a finite set of bindings $\{t_1/x_1,\dots,t_n/x_n\}$ such that the x_i's are pairwise different variables. It is not required that $t_i \not\equiv x_i$ (cf. [4]).

A *frame* consists of a query G and an idempotent environment η. Idempotent means that if $t_i \not\equiv x_i$ then x_i does not occur in $t_1,\dots,t_n$. Remember that a query is a list of goals.

A *frame stack* consists of a (possibly empty) sequence $\langle G_1,\eta_1;\dots;G_n,\eta_n\rangle$ of frames. The frames G_i,η_i are alternatives, also called choice points. The query G_n together with the environment η_n is called the *topmost frame* of the stack. Capital greek letters Φ, Ψ and Θ denote finite, possibly empty, sequences of the form $G_1,\eta_1;\dots;G_n,\eta_n$. Thus $\langle\Phi;G,\eta\rangle$ denotes a stack with topmost frame G,η.

A *state* of a computation is a finite sequence $\langle\Phi_1\rangle\ \dots\ \langle\Phi_n\rangle$ of frame stacks. $\langle\Phi_n\rangle$ is called the topmost stack of the state. States are denoted by the capital greek letter Σ. For a query G with free variables $x_1,\dots,x_n$ let *init*(G) be the state $\langle G,\{x_1/x_1,\dots,x_n/x_n\}\rangle$. There are three kinds of final states: *yes*(η), *no* and *error*.

Definition 1. The transition rules of the query evaluation procedure are:

1. $\Sigma\ \langle\Phi;\texttt{true}\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;G,\eta\rangle$
2. $\Sigma\ \langle\Phi;\texttt{fail}\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi\rangle$
3. $\Sigma\ \langle\Phi;s=t\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;G,\eta\tau\rangle$ [if $\tau = mgu(s\eta,t\eta)$]
4. $\Sigma\ \langle\Phi;s=t\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi\rangle$ [if $s\eta$ and $t\eta$ are not unifiable]
5. $\Sigma\ \langle\Phi;R(\boldsymbol{t})\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;D^P_R[\boldsymbol{t}]\ \texttt{\&}\ G,\eta\rangle$ [if R is user-defined]
6. $\Sigma\ \langle\Phi;A\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;\mathbb{B}(A\eta)\ \texttt{\&}\ G,\eta\rangle$ [if A is built-in and $A\eta\in\mathbb{D}$]
7. $\Sigma\ \langle\Phi;A\ \texttt{\&}\ G,\eta\rangle \longrightarrow error$ [if A is built-in and $A\eta\notin\mathbb{D}$]
8. $\Sigma\ \langle\Phi;(E\ \texttt{\&}\ F)\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;E\ \texttt{\&}\ (F\ \texttt{\&}\ G),\eta\rangle$
9. $\Sigma\ \langle\Phi;(E\ \texttt{or}\ F)\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;F\ \texttt{\&}\ G,\eta;E\ \texttt{\&}\ G,\eta\rangle$
10. $\Sigma\ \langle\Phi;(E\ \texttt{or}\ F)\ \texttt{\&}\ G,\eta\rangle \longrightarrow \Sigma\ \langle\Phi;E\ \texttt{\&}\ G,\eta;F\ \texttt{\&}\ G,\eta\rangle$

11. $\Sigma\ \langle \Phi; (\texttt{some}\, x\, F)\ \texttt{\&}\ G, \eta\rangle \longrightarrow \Sigma\ \langle \Phi; F\{y/x\}\ \texttt{\&}\ G, \eta \cup \{y/y\}\rangle$ [where y is new]
12. $\Sigma\ \langle \Phi; (\texttt{not}\, F)\ \texttt{\&}\ G, \eta\rangle \longrightarrow \Sigma\ \langle \Phi; (\texttt{not}\, F)\ \texttt{\&}\ G, \eta\rangle\ \langle [F], \eta\rangle$ [if $F\eta$ is ground]
13. $\Sigma\ \langle \Phi; (\texttt{not}\, F)\ \texttt{\&}\ G, \eta\rangle \longrightarrow \mathit{error}$ [if $F\eta$ is not ground]
14. $\Sigma\ \langle \Phi; (\texttt{not}\, F)\ \texttt{\&}\ G, \eta\rangle\ \langle \Psi; \texttt{true}, \tau\rangle \longrightarrow \Sigma\ \langle \Phi\rangle$
15. $\Sigma\ \langle \Phi; (\texttt{not}\, F)\ \texttt{\&}\ G, \eta\rangle\ \langle\rangle \longrightarrow \Sigma\ \langle \Phi; G, \eta\rangle$
16. $\langle \Phi; \texttt{true}, \eta\rangle \longrightarrow \mathit{yes}(\eta)$
17. $\langle\rangle \longrightarrow \mathit{no}$

Remark 1. Rule 1 says that the goal `true` can be deleted. In 2, the goal `fail` starts backtracking. This means that the topmost frame of the topmost stack is popped. In 3 and 4, equations are solved by unification. If the unification is successful, it changes the current environment; if the unification fails then backtracking starts. We assume that $\mathit{mgu}(s,t)$ returns an idempotent most general unifier if s and t are unifiable. Rule 5 and 6 deal with atomic goals. User-defined predicates are replaced by their definition forms. Built-in predicates are replaced by their built-in definitions provided that the necessary type conditions are satisfied. Otherwise, in 7, built-in predicates report an error message. Rule 8 says that the left goal is selected in a conjunction. This corresponds to a left-most goal selection rule in standard terminology or to so-called LDNF-resolution (see [2]). Rule 9 and 10 are nondeterministic. This is the only place where nondeterminism occurs. To solve a disjunction E `or` F means either to solve first E and then F or to solve first F and then E. In both cases, new frames are allocated. Without rule 10 one obtains the deterministic evaluation procedure of Prolog. In 11, existential quantified variables are standardized apart. The environment is enlarged. The variable y must be chosen in such a way that is does not appear free neither in the query $(\texttt{some}\, x\, F)\ \texttt{\&}\ G$ nor in the environment η. In 12, negated goals start subcomputations. In order to process the goal $\texttt{not}\, F\eta$, the query $[F\eta]$ is started in a subcomputation, provided that $F\eta$ is ground. Otherwise, in 13, an error message is raised. Rule 14 and 16 deal with the cases where the query of the topmost frame is the goal `true`; rule 15 and 17 deal with the cases where the topmost stack is empty. Rule 14 says that if F succeeds then $\texttt{not}\, F$ fails. Rule 15 says that if F fails then $\texttt{not}\, F$ succeeds. Rule 16 corresponds to a global success and rule 17 to a global failure.

Definition 2. We say that

1. a query *G succeeds with answer σ*, if there exists a computation with initial state $\mathit{init}(G)$ and final state $\mathit{yes}(\eta)$ such that σ is the restriction of η to the variables of G;
2. a query *G succeeds with answer including σ*, if there exist substitutions τ and θ such that G succeeds with answer τ and $G\tau\theta \equiv G\sigma$;
3. a query *G fails*, if there exists a computation with initial state $\mathit{init}(G)$ and final state no;
4. a query *G terminates*, if *all* computations with initial state $\mathit{init}(G)$ are finite and do not end in error;
5. a query *G is safe*, if there exists no computation with initial state $\mathit{init}(G)$ and final state error.

If a query is safe then during a computation all negative goals goals are ground at the time when they are processed and all built-in atoms belong to $\mathbb{D}$ when they are called. We have defined termination in such a way that it includes safeness. If a goal terminates then it is safe. Note, that termination means universal termination. For Prolog-like systems this means that one can hit the semicolon key a finite number of times until one finally obtains the message *no more solutions*.

Practice shows that most goals are terminating in this sense (cf. [1]). Also our experience with the LPTP theorem prover supports this fact. We loose nothing if we restrict our attention to terminating goals only. However, given a program P and a goal G we have to prove that the goal G is terminating. This can be done, for example, using the method of Apt and Pedreschi in [2] by guessing a level mapping for atoms and a model of the program and showing that the program is *acceptable* with respect to the level mapping and the model. Another method which we present here is to use an appropriate first-order theory with induction. This theory is called the inductive extension of pure Prolog programs and is implemented in the interactive theorem prover LPTP.

3 The inductive extension of pure Prolog programs

The inductive extension of a logic program P is, roughly speaking, Clark's completion of a logic program (cf. [3]) plus induction along the definition of the predicates. However, there are essential differences. For instance, the inductive extension is consistent for arbitrary programs. This is not the case for Clark's completion. We can prove termination of predicates in the inductive extension. This is not possible in Clark's completion.

The inductive extension is formulated in a language $\hat{\mathfrak{L}}$ which is obtained from $\mathfrak{L}$ in the following way. For each predicate symbol R of $\mathfrak{L}$ we take in $\hat{\mathfrak{L}}$ three predicates symbols R^{s}, R^{f} and R^{t} of the same arity as R. The intended meaning of these predicates is that they express success, failure and termination of R. $\hat{\mathfrak{L}}$ contains in addition a special unary predicate gr which expresses that an object is ground. The syntactic objects of $\hat{\mathfrak{L}}$ are called formulas. They are

$$\varphi, \chi, \psi ::= \top \mid \bot \mid s = t \mid S(\boldsymbol{t}) \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \varphi \rightarrow \psi \mid \forall x\, \varphi \mid \exists x\, \varphi,$$

where S denotes any predicate symbol of $\hat{\mathfrak{L}}$. We write $s \neq t$ for $\neg(s = t)$.

The meaning of formulas is given by the first-order predicate calculus of classical logic. By an $\hat{\mathfrak{L}}$-theory we mean a (possibly infinite) collection T of formulas of $\hat{\mathfrak{L}}$. We write $T \vdash \varphi$ to express that the formula φ can be derived from the $\hat{\mathfrak{L}}$-theory T by the usual rules of predicate logic with equality.

For the declarative semantics of logic programs we need three syntactic operators $\mathbf{S}$, $\mathbf{F}$ and $\mathbf{T}$ which transform goals of the language $\mathfrak{L}$ into positive formulas of $\hat{\mathfrak{L}}$. The operators $\mathbf{S}$, $\mathbf{F}$ and $\mathbf{T}$ are not part of the language. They are defined notions. $\mathbf{S}\,G$ is read: G succeeds; $\mathbf{F}\,G$ is read: G fails; $\mathbf{T}\,G$ is read: G terminates and is safe. The operator have the following definitions:

$$
\begin{array}{ll}
\mathbf{S}\,\texttt{true} :\equiv \top, & \mathbf{F}\,\texttt{true} :\equiv \bot,\\
\mathbf{S}\,\texttt{fail} :\equiv \bot, & \mathbf{F}\,\texttt{fail} :\equiv \top,\\
\mathbf{S}\,s = t :\equiv s = t, & \mathbf{F}\,s = t :\equiv s \neq t,\\
\mathbf{S}\,R(\boldsymbol{t}) :\equiv R^{\mathrm{s}}(\boldsymbol{t}), & \mathbf{F}\,R(\boldsymbol{t}) :\equiv R^{\mathrm{f}}(\boldsymbol{t}),\\
\mathbf{S}(G \mathbin{\texttt{\&}} H) :\equiv \mathbf{S}\,G \wedge \mathbf{S}\,H, & \mathbf{F}(G \mathbin{\texttt{\&}} H) :\equiv \mathbf{F}\,G \vee \mathbf{F}\,H,\\
\mathbf{S}(G \mathbin{\texttt{or}} H) :\equiv \mathbf{S}\,G \vee \mathbf{S}\,H, & \mathbf{F}(G \mathbin{\texttt{or}} H) :\equiv \mathbf{F}\,G \wedge \mathbf{F}\,H,\\
\mathbf{S}\,\texttt{some}\,x\,G :\equiv \exists x\,\mathbf{S}\,G, & \mathbf{F}\,\texttt{some}\,x\,G :\equiv \forall x\,\mathbf{F}\,G,\\
\mathbf{S}\,\texttt{not}\,G :\equiv \mathbf{F}\,G, & \mathbf{F}\,\texttt{not}\,G :\equiv \mathbf{S}\,G,
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{T}\,\texttt{true} :\equiv \top, & \mathbf{T}(G \mathbin{\texttt{\&}} H) :\equiv \mathbf{T}\,G \wedge (\mathbf{F}\,G \vee \mathbf{T}\,H),\\
\mathbf{T}\,\texttt{fail} :\equiv \top, & \mathbf{T}(G \mathbin{\texttt{or}} H) :\equiv \mathbf{T}\,G \wedge \mathbf{T}\,H,\\
\mathbf{T}\,s = t :\equiv \top, & \mathbf{T}\,\texttt{some}\,x\,G :\equiv \forall x\,\mathbf{T}\,G,\\
\mathbf{T}\,R(\boldsymbol{t}) :\equiv R^{\mathrm{t}}(\boldsymbol{t}), & \mathbf{T}\,\texttt{not}\,G :\equiv \mathbf{T}\,G \wedge \mathrm{gr}(G).
\end{array}
$$

In this definition special attention require only the cases $\mathbf{T}(G \mathbin{\texttt{\&}} H)$, $\mathbf{T}(G \mathbin{\texttt{or}} H)$ and $\mathbf{T}\,\texttt{not}\,G$. The other cases are as one would expect. The definition of $\mathbf{T}(G \mathbin{\texttt{\&}} H)$ reflects the fact that a goal $G \mathbin{\texttt{\&}} H$ terminates if, and only if,

(a) G terminates, and
(b) G fails or H terminates.

The definition of $\mathbf{T}(G \mathbin{\texttt{or}} H)$ shows that termination has to be understood as universal termination. The goal $G \mathbin{\texttt{or}} H$ terminates if, and only if, both branches G and H terminate.

The definition of $\mathbf{T}\,\texttt{not}\,G$ is the essential difference between the $\mathbf{T}$ operator here and the $\mathbf{T}$ (resp. $\mathbf{L}$) operator in [10] and [12]. There, $\mathbf{T}\,\texttt{not}\,G$ is simply defined as $\mathbf{T}\,G$. Here, we require in addition that G is ground using the operator gr which is defined as follows:

$$
\begin{array}{ll}
\mathrm{gr}(\texttt{true}) :\equiv \top, & \mathrm{gr}(G \mathbin{\texttt{\&}} H) :\equiv \mathrm{gr}(G) \wedge \mathrm{gr}(H),\\
\mathrm{gr}(\texttt{fail}) :\equiv \top, & \mathrm{gr}(G \mathbin{\texttt{or}} H) :\equiv \mathrm{gr}(G) \wedge \mathrm{gr}(H),\\
\mathrm{gr}(s = t) :\equiv \mathrm{gr}(s) \wedge \mathrm{gr}(t), & \mathrm{gr}(\texttt{some}\,x\,G) :\equiv \exists x\,\mathrm{gr}(G),\\
\mathrm{gr}(R(t_1, \dots, t_n)) :\equiv \mathrm{gr}(t_1) \wedge \dots \wedge \mathrm{gr}(t_n), & \mathrm{gr}(\texttt{not}\,G) :\equiv \mathrm{gr}(G).
\end{array}
$$

What we want is that for a goal G with free variables $x_1, \dots, x_n$ the following is true:

$$(*) \qquad \mathrm{gr}(G) \leftrightarrow \mathrm{gr}(x_1) \wedge \dots \wedge \mathrm{gr}(x_n).$$

It is not possible to take this as a definition of $\mathrm{gr}(G)$ directly, since then we would loose the substitution property that $(\mathbf{T}\,G)\sigma \equiv \mathbf{T}(G\sigma)$ for each substitution σ. We will see that in the inductive extension of a logic program $(*)$ will be provable.

Definition 3. The inductive extension of P, $\mathrm{IND}(P)$, comprises the following axioms:

I. The axioms of Clark's equality theory CET:

1. $f(x_1,\ldots,x_m) = f(y_1,\ldots,y_m) \to x_i = y_i$ [if f is m-ary and $1 \le i \le m$]
2. $f(x_1,\ldots,x_m) \ne g(y_1,\ldots,y_n)$ [if f is m-ary, g is n-ary and $f \not\equiv g$]
3. $t \ne x$ [if x occurs in t and $t \not\equiv x$]

II. Axioms for gr**:**

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

III. Uniqueness axioms (UNI):

6. $\neg(R^{\mathrm{s}}(\boldsymbol{x}) \wedge R^{\mathrm{f}}(\boldsymbol{x}))$

IV. Totality axioms (TOT):

7. $R^{\mathrm{t}}(\boldsymbol{x}) \to R^{\mathrm{s}}(\boldsymbol{x}) \vee R^{\mathrm{f}}(\boldsymbol{x})$

V. Fixed point axioms for user-defined predicates R**:**

8. $\mathbf{S}\, D^P_R[\boldsymbol{x}] \leftrightarrow R^{\mathrm{s}}(\boldsymbol{x}), \quad \mathbf{F}\, D^P_R[\boldsymbol{x}] \leftrightarrow R^{\mathrm{f}}(\boldsymbol{x}), \quad \mathbf{T}\, D^P_R[\boldsymbol{x}] \leftrightarrow R^{\mathrm{t}}(\boldsymbol{x})$

VI. Fixed point axioms for built-in, atomic goals $A \in \mathbb{D}$**:**

9. $\mathbf{S}\,\mathbb{B}(A) \leftrightarrow \mathbf{S}\,A, \quad \mathbf{F}\,\mathbb{B}(A) \leftrightarrow \mathbf{F}\,A, \quad \mathbf{T}\,\mathbb{B}(A) \leftrightarrow \mathbf{T}\,A.$

VII. True axioms for built-in predicates: We will explain below what we mean by that.

VIII. The simultaneous induction scheme for user-defined predicates: Let $R_1,\ldots,R_n$ be user-defined predicates and let $\varphi_1(\boldsymbol{x}_1),\ldots,\varphi_n(\boldsymbol{x}_n)$ be $\hat{\mathfrak{L}}$ formulas such that the length of $\boldsymbol{x}_i$ is equal to the arity of R_i for $i = 1,\ldots,n$. Let

$$\mathit{closed}(\varphi_1(\boldsymbol{x}_1)/R^{\mathrm{s}}_1,\ldots,\varphi_n(\boldsymbol{x}_n)/R^{\mathrm{s}}_n)$$

be the formula obtained from

$$\forall \boldsymbol{x}_1(\mathbf{S}\, D^P_{R_1}[\boldsymbol{x}_1] \to R^{\mathrm{s}}_1(\boldsymbol{x}_1)) \wedge \ldots \wedge \forall \boldsymbol{x}_n(\mathbf{S}\, D^P_{R_n}[\boldsymbol{x}_n] \to R^{\mathrm{s}}_n(\boldsymbol{x}_n))$$

by replacing simultaneously all occurrences of $R^{\mathrm{s}}_i(\boldsymbol{t})$ by $\varphi_i(\boldsymbol{t})$ for $i = 1,\ldots,n$ and renaming the bound variables when necessary. Let

$$\mathit{sub}(\varphi_1(\boldsymbol{x}_1)/R^{\mathrm{s}}_1,\ldots,\varphi_n(\boldsymbol{x}_n)/R^{\mathrm{s}}_n)$$

be the formula

$$\forall \boldsymbol{x}_1(R^{\mathrm{s}}_1(\boldsymbol{x}_1) \to \varphi_1(\boldsymbol{x}_1)) \wedge \ldots \wedge \forall \boldsymbol{x}_n(R^{\mathrm{s}}_n(\boldsymbol{x}_n) \to \varphi_n(\boldsymbol{x}_n)).$$

Then the simultaneous induction axiom is the formula

$$\mathit{closed}(\varphi_1(\boldsymbol{x}_1)/R^{\mathrm{s}}_1,\ldots,\varphi_n(\boldsymbol{x}_n)/R^{\mathrm{s}}_n) \to \mathit{sub}(\varphi_1(\boldsymbol{x}_1)/R^{\mathrm{s}}_1,\ldots,\varphi_n(\boldsymbol{x}_n)/R^{\mathrm{s}}_n).$$

Remark 2. I. Clark's equality theory CET is needed for the formalization of unification.

II. The predicate gr is used to express that a term is ground. If $\mathrm{gr}(t)$ is provable from $\mathrm{IND}(P)$, then t is ground. We assume that the language contains at least one constant symbol.

III. From the uniqueness axioms (UNI) one can immediately derive the principle $\neg(\mathbf{S}\,G \land \mathbf{F}\,G)$ for arbitrary goals G.

IV. From the totality axioms (TOT) one can derive $\mathbf{T}\,G \to \mathbf{S}\,G \lor \mathbf{F}\,G$ for each goal G.

V. The fixed point axioms for user defined-predicates express that one can read a clause both, from body to head, but also from head to body.

VI. In the fixed point axioms for built-in predicates it is important that A belongs to $\mathbb{D}$. Otherwise, $\mathbb{B}(A)$ is not defined.

VII. For example, the following axioms for built-in predicates are true:

1. $\forall x_1, x_2, y\,(\mathbf{S}\,x_1 \mathtt{~is~} y \land \mathbf{S}\,x_2 \mathtt{~is~} y \to x_1 = x_2)$.
2. $\forall x\,(\mathrm{gr}(x) \leftrightarrow \mathbf{T}\,\mathtt{integer}(x))$.
3. $\forall x(\mathbf{S}\,\mathtt{integer}(x) \to \mathbf{F}\,x < x)$.
4. $\forall x_1, x_2, y_1, y_2\,(\mathbf{S}\,x_1 \mathtt{~is~} y_1 \land \mathbf{S}\,x_2 \mathtt{~is~} y_2 \to (\mathbf{S}\,x_1 < x_2 \leftrightarrow \mathbf{S}\,y_1 < y_2))$.
5. $\forall x, y, z(\mathbf{S}\,\mathtt{integer}(x) \land \mathbf{S}\,\mathtt{integer}(y) \land \mathbf{S}\,\mathtt{integer}(z) \land \mathbf{S}\,x < y \land \mathbf{S}\,y < z \to \mathbf{S}\,x < z)$.

Note, that axioms like $x = 7 \leftrightarrow \mathbf{S}(x \mathtt{~is~} 3 + 4)$ are included in the fixed point axioms VI. The full version of this article [14] contains an exact definition of what it means that an axiom is true.

VIII. The simultaneous induction scheme expresses the minimality of the R^{s} predicates. Note, that the formulas $\mathbf{S}\,D_R^P$ are positive. Informally, the induction scheme says that one can use induction along the definition of the predicates. For example, for the $\mathtt{append}/3$ and the $\mathtt{list}/1$ predicate we have the following rules:

$$\frac{\forall \ell\,\varphi([], \ell, \ell) \qquad \forall x, \ell_1, \ell_2, \ell_3\,(\mathbf{S}\,\mathtt{append}(\ell_1, \ell_2, \ell_3) \land \varphi(\ell_1, \ell_2, \ell_3) \to \varphi([x|\ell_1], \ell_2, [x|\ell_3]))}{\forall \ell_1, \ell_2, \ell_3\,(\mathbf{S}\,\mathtt{append}(\ell_1, \ell_2, \ell_3) \to \varphi(\ell_1, \ell_2, \ell_3))}$$

$$\frac{\varphi([]) \qquad \forall x, \ell\,(\mathbf{S}\,\mathtt{list}(\ell) \land \varphi(\ell) \to \varphi([x|\ell]))}{\forall \ell\,(\mathbf{S}\,\mathtt{list}(\ell) \to \varphi(\ell))}$$

The predicates $\mathtt{list}/1$ and $\mathtt{append}/3$ have their standard definitions:

```
list([]).                append([], ℓ, ℓ).
list([x|ℓ]) :- list(ℓ).  append([x|ℓ1], ℓ2, [x|ℓ3]) :- append(ℓ1, ℓ2, ℓ3).
```

The expression $[]$ denotes a constant for the empty list and $[\cdot|\cdot]$ is a binary function symbol for constructing list.

The inductive extension is related to Clark's completion and Kunen's three-valued completion in the following way. Let (FIX) be the collection of the fixed-point axioms for the R^{s} and R^{f} relations. Then (CET)+(UNI)+(FIX) is equivalent to Kunen's three-valued completion of [8]. Moreover, Clark's completion of [3] can be obtained from the three-valued completion be adding the stronger totality axiom $R^{\mathrm{s}}(\boldsymbol{t}) \vee R^{\mathrm{f}}(\boldsymbol{t})$.

Example 4. The simultaneous induction scheme is more natural for logic programs than structural induction on the Herbrand universe. Assume that the language $\mathfrak{L}$ has exactly one constant symbol c and one unary function symbol f. In this case, *induction on the universe* is the scheme

$$(**) \qquad \varphi(c) \wedge \forall x\,(\varphi(x) \to \varphi(f(x)) \to \forall x\,\varphi(x).$$

Let P be the program with the two clauses $q \mathrel{:\!-} r(x)$ and $r(f(x)) \mathrel{:\!-} r(x)$. Using induction on the universe $(**)$ for $\varphi(x) :\equiv \mathbf{T}\,r(x)$ and the fixed point axioms

$$\forall x\,\mathbf{T}\,r(x) \leftrightarrow \mathbf{T}\,q \quad \text{and} \quad \forall y(x = f(y) \to \mathbf{T}\,r(y)) \leftrightarrow \mathbf{T}\,r(x)$$

one can easily derive $\forall x\,\mathbf{T}\,r(x)$ and hence $\mathbf{T}\,q$. But the goal q does not terminate under query evaluation. Therefore, induction on the universe is not appropriate for our purposes. We want that $\mathbf{T}\,G$ is provable if, and only if, G terminates.

Proofs of the following two theorems can be found in the full version of this paper [14]. They use ideas from [8] and [11].

Theorem 1 (Soundness).

1. *If G terminates, then* $\mathrm{IND}(P) \vdash \mathbf{T}\,G$.
2. *If G succeeds with answer σ, then* $\mathrm{IND}(P) \vdash \mathbf{S}\,G\sigma$.
3. *If G fails, then* $\mathrm{IND}(P) \vdash \mathbf{F}\,G$.

In the proof of this theorem the full power of the inductive extension is not used. Only CET and the directions from left to right in the fixed point axioms are needed.

Theorem 2 (Adequacy).

1. *If* $\mathrm{IND}(P) \vdash \mathbf{T}\,G$, *then G terminates.*
2. *If* $\mathrm{IND}(P) \vdash \mathbf{T}\,G \wedge \mathbf{S}\,G\sigma$, *then G succeeds with answer including σ.*
3. *If* $\mathrm{IND}(P) \vdash \mathbf{T}\,G \wedge \mathbf{F}\,G$, *then G fails.*

This theorem is not trivial, since the term model in which $R^{\mathrm{t}}(\boldsymbol{t})$ is true iff $R(\boldsymbol{t})$ terminates is, in general, not a model of the inductive extension (cf. Example 4). Note, that the theorem implies, for example, the following existence property:

Corollary 1. *If* $\mathrm{IND}(P) \vdash \mathbf{S}(\mathtt{some}\,x\,G[x]) \wedge \mathbf{T}(\mathtt{some}\,x\,G[x])$ *then there exists a term t such that the goal $G[x]$ succeeds with answer $\{t/x\}$ and* $\mathrm{IND}(P) \vdash \mathbf{S}\,G[t]$.

It is important to note, that from the provability of $\mathbf{T}\,G$ if follows not only that all computations for G terminate but also that there are no errors in calls of built-in predicates during the computation. There is an interesting analogy between the $\mathbf{T}$ operator and the *logic of partial terms* (cf. eg. [5,6]). In the logic of partial terms the expression $t\!\downarrow$ means that the functional program t terminates and that during the evaluation there are no type conflicts, i.e. the program is dynamically well-typed. The meaning of $\mathbf{T}\,G$ is similar. It means that the evaluation of the goal G terminates and that there are no error messages caused by non-ground negative goals or wrongly typed built-in atomic goals.

The next theorem is proved in [13] using standard methods like partial cut-elimination for infinitary systems and asymmetric interpretations.

Theorem 3. *Without built-in predicates,* IND(P) *has the same proof-theoretic strength as Peano Arithmetic.*

From the proof of this theorem in [13] one could extract a program P with a distinguished predicate symbol R such that the true formula

$$\forall x(\mathbf{S}\,\texttt{list}(x) \to \mathbf{T}\,R(x))$$

is not provable in IND(P). The reason that the formula is not provable is that the computation tree for $R(\ell)$ grows too fast compared to the length of the list ℓ. In practice, however, such programs do not occur.

4 LPTP — a logic program theorem prover

In this section we give a short overview of LPTP, an interactive theorem prover which is based on the inductive extension of pure Prolog programs. LPTP is a proof refinement system that allows a user to construct formal proofs interactively. The user can generate proofs deductively from the assumptions forwards to the goal or goal directed backwards from the goal to the axioms. LPTP has the ability to search for proofs automatically. In the simplest case, LPTP just finds the name of a lemma that can be used at a certain point in a proof. In the best case, LPTP finds complete proofs. In general LPTP can complete automatically small gaps in proofs that require not more than, say, 10 steps. For the rest of the proof LPTP has to be guided by the user.

LPTP consists of 6500 lines of Prolog code. It runs in CProlog, Quintus Prolog, and SICStus Prolog under Unix. LPTP has a graphical user interface in the Gnu Emacs Editor. For example, the user can double-click on a quantifier and the whole scope of the quantifier is highlighted. LPTP generates TEX and HTML output.

The kernel of LPTP is written in exactly the fragment of Prolog that can be treated in LPTP. This means that LPTP uses no single cut. Moreover, it is possible to prove properties of LPTP within LPTP.

The largest program we have verified with LPTP is 635 lines long. It is a parser for standard ISO Prolog. The 635 lines comprise not only the implementation but also the specification of the parser. The correctness proof includes theorems like the following: if a parse tree is transformed into a token list (using `write`) and the token list is parsed back into a parse tree (using `read`), then this parse tree is identical to the original one.

The fully formalized correctness proof for the ISO Prolog parser is 13000 lines long. So we have a factor of 20 for the full verification of this example program. LPTP is able to check the whole proof (133 pages) in 99.2 seconds on a Sun SPARCstation. This speed, however, says not much about LPTP, since it is much more important how fast a user can create proofs using the system. A skilled user can generate more than 1000 lines of formal proofs in one day. Altogether we have generated 25000 lines of formal proofs with LPTP.

5 En example proof in LPTP

As an example, we now sketch a proof of an algorithm that inserts elements into AVL trees. AVL trees are ordered binary trees. They are subject to the Adelson-Velskii-Landis balance criterion: A tree is balanced iff for every node the heights of its two subtrees differ by at most 1. Our Prolog version of the algorithm is generic in two predicates `a`/1 and `r`/2. The idea is that `r`/2 is a total ordering on the set `a`/1. In fact, we only need that `a`/1 and `r`/2 satisfy the following axioms:

1. $\forall x, y, z(\mathbf{S}\,\mathtt{a}(x) \wedge \mathbf{S}\,\mathtt{a}(y) \wedge \mathbf{S}\,\mathtt{a}(z) \wedge \mathbf{S}\,\mathtt{r}(x,y) \wedge \mathbf{S}\,\mathtt{r}(y,z) \rightarrow \mathbf{S}\,\mathtt{r}(x,z))$
2. $\forall x, y(\mathbf{S}\,\mathtt{a}(x) \wedge \mathbf{S}\,\mathtt{a}(y) \rightarrow \mathbf{S}\,\mathtt{r}(x,y) \vee \mathbf{S}\,\mathtt{r}(y,x))$
3. $\forall x, y(\mathbf{S}\,\mathtt{a}(x) \wedge \mathbf{S}\,\mathtt{a}(y) \rightarrow \mathbf{T}\,\mathtt{r}(x,y)$
4. $\forall x(\mathbf{S}\,\mathtt{a}(x) \rightarrow \mathrm{gr}(x))$.

Axiom (1) says that `r`/2 is transitive on `a`/1; (2) says that `r`/2 is total on `a`/1; (3) says that `r`/2 terminates on `a`/1; (4) says that `a`/1 contains only ground terms.

The algorithm is coded as predicate `addavl`/3. If x is a value of `a`/1 and t_1 is an AVL tree then $\mathtt{addavl}(x, t_1, t_2)$ inserts x into t_1 and returns the result in t_2.

The empty tree is represented as `t`. A tree with value x, left subtree ℓ and right subtree r is represented as $\mathtt{t}(x, b, \ell, r)$; b is the difference of the height of r and the height of ℓ; b can be `-1`, `0` or `1`.

For the specification of the correctness of the algorithm we need the predicates `avl`/1 and `in`/2. The predicate $\mathtt{avl}(t)$ expresses that (i) t is a tree with values x belonging to `a`/1; (ii) t satisfies the Adelson-Velskii-Landis balance criterion; (iii) t is ordered, i.e. in a node $\mathtt{t}(x, b, \ell, r)$, x is an upper bound of the elements of ℓ and a lower bound of the elements of r with respect to the ordering `r`/2. The predicate $\mathtt{in}(x, t)$ expresses that the value x occurs in the tree t.

Correctness of the algorithm can be expressed by the following formulas:

1. $\forall x, t_1, t_2(\mathbf{S}\,\mathtt{a}(x) \wedge \mathbf{S}\,\mathtt{avl}(t_1) \wedge \mathbf{S}\,\mathtt{addavl}(x, t_1, t_2) \rightarrow \mathbf{S}\,\mathtt{avl}(t_2))$.
2. $\forall x, t_1, t_2(\mathbf{S}\,\mathtt{addavl}(x, t_1, t_2) \rightarrow \mathbf{S}\,\mathtt{in}(x, t_2))$.
3. $\forall x, y, t_1, t_2(\mathbf{S}\,\mathtt{addavl}(x, t_1, t_2) \wedge \mathbf{S}\,\mathtt{in}(y, t_1) \rightarrow \mathbf{S}\,\mathtt{in}(y, t_2))$.
4. $\forall x, y, t_1, t_2(\mathbf{S}\,\mathtt{addavl}(x, t_1, t_2) \wedge \mathbf{S}\,\mathtt{in}(y, t_2) \rightarrow y = x \vee \mathbf{S}\,\mathtt{in}(y, t_1))$.
5. $\forall x, t_1, t_2(\mathbf{S}\,\mathtt{a}(x) \wedge \mathbf{S}\,\mathtt{avl}(t_1) \rightarrow \mathbf{T}\,\mathtt{addavl}(x, t_1, t_2))$.

6. $\forall x, t_1(\mathbf{S}\,\mathtt{a}(x) \wedge \mathbf{S}\,\mathtt{avl}(t_1) \to \exists t_2(\mathbf{S}\,\mathtt{addavl}(x, t_1, t_2)))$.

The formulas mean the following:

1. If we insert an element into an AVL tree, then the new tree we get is also an AVL tree.
2. If we insert an element in an AVL tree, then the added element is an element of the new AVL tree.
3. If we add an element to an AVL tree containing y, then the new AVL tree also contains y.
4. If we add an element x to an AVL tree, and if the new AVL tree contains y, then y is the element x we just added, or y was already in the initial AVL tree.
5. The algorithm terminates for appropriate inputs.
6. The algorithm is complete. It can insert elements into arbitrary AVL trees.

The algorithm `addavl/3` together with the predicates used in the specification is 137 lines long. The formal correctness proof of formulas 1–6 is 2903 lines long. It has been created by Patrik Fuhrer and Rene Lehmann and is part of the distribution of LPTP.

References

1. K. R. Apt. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice Hall, 1996.
2. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.
3. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
4. S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *J. of Logic Programming*, 5(1):61–91, 1988.
5. S. Feferman. Logics for termination and correctness of functional programs. In Y. N. Moschovakis, editor, *Logic from Computer Science*, pages 95–127, New York, 1992. Springer-Verlag.
6. S. Feferman. Logics for termination and correctness of functional programs, II. Logics of strength PRA. In P. Aczel, H. Simmons, and S. S. Wainer, editors, *Proof Theory*, pages 195–225. Cambridge University Press, 1992.
7. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.
8. K. Kunen. Signed data dependencies in logic programs. *J. of Logic Programming*, 7(3):231–245, 1989.
9. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. of Logic Programming*, 29(1–3):17–64, 1996.
10. R. F. Stärk. The declarative semantics of the Prolog selection rule. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, LICS '94*, pages 252–261, Paris, France, July 1994. IEEE Computer Society Press.
11. R. F. Stärk. Input/output dependencies of normal logic programs. *J. of Logic and Computation*, 4(3):249–262, 1994.

12. R. F. Stärk. First-order theories for pure Prolog programs with negation. *Archive for Mathematical Logic*, 34(2):113–144, 1995.
13. R. F. Stärk. The finite stages of inductive definitions. In P. Hájek, editor, *GÖDEL'96. Logical Foundations of Mathematics, Computer Science and Physics — Kurt Gödel's Legacy*, pages 267–290, Brno, Czech Republic, 1996. Springer-Verlag, Lecture Notes in Logic 6.
14. R. F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998.

LPTP is available on the WWW from:
`http://www.inf.ethz.ch/~staerk/lptp/lptp-1.06.tar.gz`
Further information on LPTP can be found at:
`http://www.inf.ethz.ch/~staerk/lptp.html`