Synthesis of positive logic programs for checking a class of definitions with infinite quantification

Francisco J. Galán *, José M. Cañete-Valdeón

Dept. of Languages and Computer Systems, Faculty of Computer Science, Av. Reina Mercedes s/n, 41012, Seville, Spain

A B S T R A C T

We describe a method based on unfold/fold transformations that synthesizes positive logic programs $P(r)$ with the purpose of checking mechanically definitions of the form $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X,Y))$ where r is the relation defined by the formula $Q\,Y\,R(X,Y)$, X is a set of variables to be instantiated at runtime by ground terms, $Q\,Y$ is a set of quantified variables on infinite domains (Q is the quantifier) and $R(X,Y)$ a quantifier-free formula in the language of a first-order logic theory. This work constitutes a first step towards the construction of a new type of assertion checkers with the ability of handling restricted forms of infinite quantification.

Keywords:
Program assertion
Logic program
Program synthesis
Unfold/fold transformation

1. Introduction

An assertion is a logic formula representing a condition the program-being-tested must satisfy in each of its executions. Current technology is not able to check assertions containing some kind of infinite quantification [36,40,5,29,33,30,61,50]. However, infinite quantification has shown to be a useful resource for expressing program states in a declarative way [20,17,26,18]. For instance, the following assertion formalizes the subset relation between a program variable L and a program variable S where $member(X,Y)$ is true if a natural number X is included in a sequence of natural numbers Y and false otherwise:

$$\forall E(member(E,L) \Rightarrow member(E,S))$$

Despite its simplicity, the sub-expression $\forall E$ formalizes a quantification over the infinite set of natural numbers. This kind of quantification is not recognized by current assertion checkers.

To palliate this lack of expressivity in current assertion languages, we propose the use of a class of assertion definitions of the form $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X,Y))$ where r is the relation defined by the assertion $Q\,Y\,R(X,Y)$, X is a set of variables to be instantiated at runtime by ground terms, $Q\,Y$ is a set of quantified variables on infinite domains (Q is the quantifier) and $R(X,Y)$ a quantifier-free formula in the language of a typed first-order logic theory. For instance,

$$D(subset) = \forall L, S(subset(L,S) \Leftrightarrow \forall E(member(E,L) \Rightarrow member(E,S)))$$

is an assertion definition written in the language of the following typed first-order logic theory:

* Corresponding author.
E-mail address: galanm@us.es (F.J. Galán).

Theory $\mathcal{T}_0$

Types

Nat	generated by	$zero :\rightarrow Nat \quad succ : Nat \rightarrow Nat$
Seq	generated by	$empty :\rightarrow Seq \quad seq : Nat \times Seq \rightarrow Seq$

Predicates

Signature: $id : Nat \times Nat$
Axioms:
1. $id(zero, zero) \Leftrightarrow true$
2. $\forall(id(succ(X), zero) \Leftrightarrow false)$
3. $\forall(id(zero, succ(Y)) \Leftrightarrow false)$
4. $\forall(id(succ(X), succ(Y)) \Leftrightarrow id(X, Y))$

Signature: $member : Nat \times Seq$
Axioms:
5. $\forall(member(E, empty) \Leftrightarrow false)$
6. $\forall(member(E, seq(X, Y)) \Leftrightarrow id(E, X) \vee member(E, Y))$

The *rationale* of our proposal is the following. Given an assertion definition $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X, Y))$, we propose to synthesize a positive logic program $P(r)$ for checking (runtime) assertions of the form $r(X)\theta$, being θ a ground substitution for X. The high-level design of $P(r)$ will depend on the quantifier Q in $D(r)$. If $Q = \forall$ then $P(r)$ will be a program which searches for refutations and if $Q = \exists$ then $P(r)$ will be a program which searches for proofs. In concrete terms, $P(r)$ is implemented by a clause $\forall(r(X) \Leftarrow r_1(X, Y))$ which defines r in terms of a new relation symbol r_1. This new relation is defined by a positive logic program $P(r_1)$ which is synthesized from an auxiliary specification $S(r_1) = \forall X, Y(r_1(X, Y) \Leftrightarrow \neg R(X, Y))$ if $Q = \forall$ or from an auxiliary specification $S(r_1) = \forall X, Y(r_1(X, Y) \Leftrightarrow R(X, Y))$ if $Q = \exists$. For instance, $P(subset_1)$ is synthesized from the following auxiliary specification:

$$S(subset_1) = \forall E, L, S(subset_1(E, L, S) \Leftrightarrow \neg(member(E, L) \Rightarrow member(E, S)))$$

For synthesizing positive logic programs, we follow the so-called transformational approach [19,20]. Most of the program synthesis methods proposed in the literature [3,14,16,19,20,28,43,47,49] are of theoretical nature or require human intervention. By contrast, our proposal is similar in spirit to the ones described in [13,54] where programs are derived from a restricted class of specifications in a completely automatic manner.

Our synthesis method must satisfy two main requirements: (1) the synthesis process must be terminating, that is, $P(r_1)$ must be synthesized in a finite amount of transformation steps and (2) the synthesized programs (i.e. $P(r_1)$) must preserve total correctness wrt the class of goals $\Leftarrow r_1(X, Y)\theta$, that is, $\exists Y r_1(X, Y)\theta$ is true if and only if $P(r_1) \cup \{\Leftarrow r_1(X, Y)\theta\}$ has an SLD-refutation [37].

In our example, $P(assert_1)$ has been synthesized after five transformation steps resulting the following program:

Synthesized program $P(subset_1)$

$\forall(subset_1(E, seq(X, Y), S) \Leftarrow subset_2(E, S) \wedge subset_3(E, X))$
$\forall(subset_1(E, seq(X, Y), S) \Leftarrow subset_1(E, Y, S) \wedge subset_4(E, X))$
$\forall(subset_1(E, empty, S) \Leftarrow subset_5(E, S))$
$\forall(subset_2(E, seq(X, Y)) \Leftarrow subset_2(E, Y) \wedge subset_4(E, X))$
$\forall(subset_2(E, empty) \Leftarrow subset_7)$
$subset_3(zero, zero) \Leftarrow subset_9$
$\forall(subset_3(succ(X), succ(Y)) \Leftarrow subset_3(X, Y))$
$\forall(subset_4(zero, succ(Y)) \Leftarrow subset_9)$
$\forall(subset_4(succ(X), succ(Y)) \Leftarrow subset_4(X, Y))$
$\forall(subset_4(succ(X), zero) \Leftarrow subset_9)$
$\forall(subset_5(E, seq(X, Y)) \Leftarrow subset_5(E, Y) \wedge subset_4(E, X))$
$subset_7 \Leftarrow$
$subset_9 \Leftarrow$

Finally, $P(subset)$ results from the union of $P(subset_1)$ and the clause which defines $subset$ in terms of the new relation symbol $subset_1$: $\forall(subset(L, S) \Leftarrow subset_1(E, L, S))$.

As we will see in detail in Sect. 7, for checking a (runtime) assertion $r(X)\theta$, we propose to compute $P(r) \cup \{G\}$, being G the goal $\Leftarrow r(X)\theta$ and $P(r)$ the synthesized program for checking $D(r)$. Again, depending on the quantifier Q in $D(r)$, we can distinguish two cases: (a) for $Q = \forall$, we have that $r(X)\theta$ is false if the empty answer is computed for $P(r) \cup \{G\}$ (*refutation*) and $r(X)\theta$ is true if no answer is computed for $P(r) \cup \{G\}$ (*impossibility of refutation*) and (b) for $Q = \exists$, we have that $r(X)\theta$ is true if the empty answer is computed for $P(r) \cup \{G\}$ (*proof*) and $r(X)\theta$ is false if no answer is computed for $P(r) \cup \{G\}$ (*impossibility of proof*).

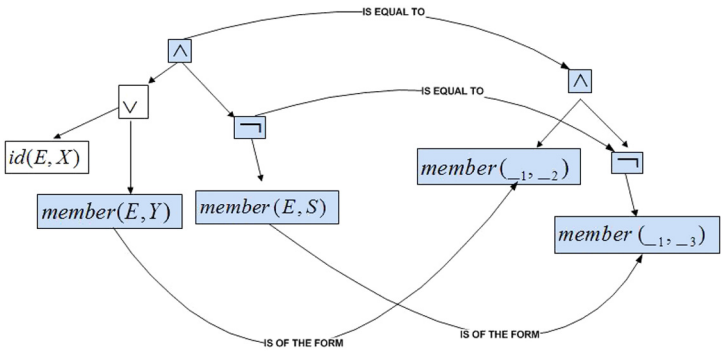

Fig. 1. Similarity between formula and pattern.

For instance, for checking a runtime assertion $subset(seq(zero, empty), empty)$, we propose to compute $P(subset) \cup \{\Leftarrow subset(seq(zero, empty), empty)\}$. As we can verify, the empty answer results from this computation what means that $subset(seq(zero, empty), empty)$ is false (refutation).

At this point, we illustrate the transformational nature of the synthesis method with a partial synthesis of $P(subset_1)$.

The synthesis of $P(subset_1)$ starts from an auxiliary specification $S(subset_1)$. The first transformation we apply to $S(subset_1)$ is to *normalize* its right-hand side into negated normal form (i.e. every negation occurring in the formula must be part of a literal) [44]:

$$(0)\ \forall E, L, S(subset_1(E, L, S) \Leftrightarrow (member(E, L) \wedge \neg member(E, S)))$$

The next transformation is to *unfold* the resulting formula with respect to some of its atoms. In this example, the selected atom is $member(E, L)$. The replacement of this atom by its definition (see the axioms of *member* in our example theory $\mathcal{T}$) gives the following result:

$$(1a)\ \forall E, X, Y, S(subset_1(E, seq(X, Y), S) \Leftrightarrow (id(E, X) \vee member(E, Y)) \wedge \neg member(E, S))$$
$$(1b)\ \forall E, S(subset_1(E, empty, S) \Leftrightarrow (false \wedge \neg member(E, S)))$$

In what follows, we show the folding of $(1a)$. For $(1b)$, the folding is done in a similar way.

The *folding* of $(1a)$ begins with the inference of a pattern similar in form to $(1a)$. This pattern is $subset_1(\_1, \_2, \_3) \Leftrightarrow (member(\_1, \_2) \wedge \neg member(\_1, \_3))$, being $\_i$ a placeholder for a variable. Fig. 1 shows graphically the similarity between the right-hand side of $(1a)$ and the right-hand side of the inferred pattern.

As we will detail in Sect. 4, the similarity with respect to the pattern will allow to *rewrite* $(1a)$ as follows:

$$(2a)\ \forall E, X, Y, S(subset_1(E, seq(X, Y), S) \Leftrightarrow (true \wedge \neg member(E, S)) \wedge id(E, X) \vee \underbrace{(member(E, Y) \wedge \neg member(E, S))}_{F} \wedge \neg id(E, X))$$

We can now instantiate the inferred pattern with the variables of the sub-formula F resulting the following formula:

$$subset_1(E, Y, S) \Leftrightarrow (member(E, Y) \wedge \neg member(E, S))$$

and then to *fold* F in $(2a)$:

$$(3a)\ \forall E, X, Y, S(subset_1(E, seq(X, Y), S) \Leftrightarrow (true \wedge \neg member(E, S)) \wedge id(E, X) \vee subset_1(E, Y, S) \wedge \neg id(E, X))$$

The synthesis continues by folding the remaining sub-formulae in $(3a)$. This task requires the inference of additional patterns:

$$subset_2(\_1, \_2) \Leftrightarrow true \wedge \neg member(\_1, \_2)$$
$$subset_3(\_1, \_2) \Leftrightarrow id(\_1, \_2)$$
$$subset_4(\_1, \_2) \Leftrightarrow \neg id(\_1, \_2)$$

The result of the whole folding is:

Table 1
NNF rewriting rules.

$\neg(\phi \Leftrightarrow \psi)$	$\rightarrow$	$(\neg\phi \wedge \psi) \vee (\phi \wedge \neg\psi)$	$\neg(\phi \vee \psi)$	$\rightarrow$	$\neg\phi \wedge \neg\psi$
$\neg(\phi \Rightarrow \psi)$	$\rightarrow$	$(\phi \wedge \neg\psi)$	$\neg\neg\phi$	$\rightarrow$	ϕ
$\neg(\phi \wedge \psi)$	$\rightarrow$	$\neg\phi \vee \neg\psi$			

$$(4a)\ \forall E, X, Y, S(subset_1(E, seq(X, Y), S) \Leftrightarrow \\ subset_2(E, S) \wedge subset_3(E, X) \vee \\ subset_1(E, Y, S) \wedge subset_4(E, X))$$

From $(4a)$, we can finally derive the following clauses of $P(subset_1)$:

$$\forall(subset_1(E, seq(X, Y), S) \Leftarrow subset_2(E, S) \wedge subset_3(E, X))$$
$$\forall(subset_1(E, seq(X, Y), S) \Leftarrow subset_1(E, Y, S) \wedge subset_4(E, X))$$

In a similar way, the synthesis of $(1b)$ generates the clause:

$$\forall(subset_1(E, empty, S) \Leftarrow subset_5(E, S))$$

As we can see, the synthesis of $(1a)$ (and of $(1b)$) generates new relation symbols (i.e. $subset_2$, $subset_3$ and $subset_4$) that have to be synthesized in subsequent iterations until completing all clauses of $P(subset_1)$.

The remainder of this paper has been organized in the following manner. Sect. 2 introduces basic notations and definitions. In Sect. 3, we formalize a class of assertion definitions containing infinite quantification. In Sect. 4, we define an unfold/fold transformation step based on the concept of similarity between a formula and a pattern. In Sect. 5, we define the synthesis of an assertion definition as a finite sequence of unfold/fold transformation steps. To validate the feasibility of our proposal, we have constructed an experimental synthesizer. In Sect. 6, we show the results of multiple synthesis experiments with assertions taken from the literature. Once defined the manner of synthesizing positive logic programs, we define in Sect. 7 the manner of checking assertions with such programs. In Sect. 8, we focus on related works and finally, in Sect. 9, we establish the conclusions. We have added an appendix which includes the proofs of the main results, the set of predicate specifications used in our experiments and the logic program which results from the synthesis of a non-trivial assertion definition.

2. Notation and preliminary definitions

In this section, we introduce the notation followed in this paper and a set of basic definitions.

For writing logic formulae, we will use the following symbols: $\neg$ (negation), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implication), $\Leftarrow$ (implication), $\Leftrightarrow$ (equivalence), $\forall$ (universal quantifier) and $\exists$ (existential quantifier). Variables are denoted by letters in upper case (E, L, etc). Sets of variables are also denoted by letters in upper case. The context will aid to decide between the reference to a variable and the reference to a set of variables. Constant and function symbols are denoted by words in lower case ($zero$, $succ$, $empty$, seq, etc). Relation symbols (predicates) are denoted by words in lower case ($member$, $nocc$, etc).

Variables and constants are *terms*. If f is a function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a *term*. For instance, $succ(X)$ is a term. A *ground term* is a term without any variable. For instance, $succ(zero)$ is a ground term. A *ground substitution* is a substitution whose terms are all ground.

An *atom* (or atomic formula) is a predicate symbol defined on a sequence of terms. For instance, $member(E, empty)$ is an atom. Every atom is a *formula*. If F and G are formulae, then $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$, $F \Leftarrow G$, $F \Leftrightarrow G$, $\forall X(F)$ and $\exists X(F)$ are *formulae*. If $p(t_1, ..., t_n)$ is an atom, then $p(t_1, ..., t_n)$ is its positive literal, $\neg p(t_1, ..., t_n)$ is its negative literal, and these two literals are said to be complements of each other. A *ground atom* is an atom without any variable. We can define existential quantification in function of universal quantification or universal quantification in function of existential quantification:

$$\exists X(F) \text{ stands for } \neg\forall X(\neg F)$$
$$\forall X(F) \text{ stands for } \neg\exists X(\neg F)$$

In $Q\,X(F)$, F is called the *scope* of the quantifier Q. The occurrence of a variable out of the scope of a quantifier is known as *free variable*. For instance, in $\forall E(member(E, L) \Rightarrow member(E, S))$, L and S are free variables. A *quantifier-free formula* is a formula without quantification. We will denote by $F(X)$ a formula F whose set of free variables is X. The quantification of all free variables in a formula is known as *closure*. We denote by $Q(F)$ the Q-closure of a quantifier-free formula F, being Q the quantifier. For instance, $\forall(member(E, L) \Rightarrow member(E, S))$ is the $\forall$-closure of $member(E, L) \Rightarrow member(E, S)$.

A formula is said to be in *negated normal form* (NNF) if and only if each of its negation symbols occurs as part of a literal [44]. This normal form can be achieved by applying recurrently the rules shown in Table 1.

For instance, the formula $\forall E, L, S(subset_1(E, L, S) \Leftrightarrow \neg(member(E, L) \Rightarrow member(E, S))$ is not in NNF. By applying the rules in Table 1 we can transform it into an equivalent one in NNF: $\forall E, L, S(subset_1(E, L, S) \Leftrightarrow (member(E, L) \wedge \neg member(E, S))$.

A *clause* is formula of the form $\forall(A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_n)$ where $A_1, ..., A_k, B_1, ..., B_n$ are atoms. Because clauses are so common in logic programming, it will be convenient to denote it by $A_1, ..., A_k \Leftarrow B_1, ..., B_n$.

A *definite clause* is a clause of the form $A \Leftarrow B_1, ..., B_n$. A is called the head and $B_1, ..., B_n$ is called the body of the clause. A *unit clause* is a definite clause of the form $A \Leftarrow$.

A *definite (or positive) logic program* is a finite set of definite clauses. A *definite goal* is a clause of the form $\Leftarrow B_1, ..., B_n$ where $B_1, ..., B_n$ are atoms.

3. The specification formalism

This section is devoted to the formalization of a class of assertions with infinite quantification. For the purpose of checking assertions mechanically, we restrict the assertion language to a class of theories adapted to the synthesis of logic programs by unfold/fold transformations.

In a first approximation, a *theory* is a specification of a (finite) set of types and predicates. Every *type* defines a domain of ground terms recursively generated from a set of constants and function symbols. A *predicate specification* $S(r)$ consists of a signature for a predicate (or relation symbol) r and a set of if-and-only-if axioms of the form $\forall(r(X) \Leftrightarrow R(Y))$ where $r(X)$ is an atom called the left-hand side of the axiom and $R(Y)$ is a quantifier-free first-order logic formula called the right-hand side of the axiom. In the language of a theory, we also count with two propositional constants, $true$ and $false$, corresponding to the truth values, true and false, respectively.

In the rest of this paper, we denote by $lhs(Ax)(rhs(Ax))$ the left-hand side (right-hand side) of an if-and-only-if axiom Ax.

Example 1. Theory with two types (Nat and Seq) and three predicate symbols (id, $nocc$ and $member$).

Theory $\mathcal{T}_1$

Types

Nat	generated by	$zero :\rightarrow Nat \quad succ : Nat \rightarrow Nat$
Seq	generated by	$empty :\rightarrow Seq \quad seq : Nat \times Seq \rightarrow Seq$

Predicates

Signature: $id : Nat \times Nat$

Axioms:

1. $id(zero, zero) \Leftrightarrow true$
2. $\forall(id(succ(X), zero) \Leftrightarrow false)$
3. $\forall(id(zero, succ(Y)) \Leftrightarrow false)$
4. $\forall(id(succ(X), succ(Y)) \Leftrightarrow id(X, Y))$

Signature: $nocc : Nat \times Seq \times Nat$

Axioms:

5. $\forall(nocc(E, empty, N) \Leftrightarrow id(N, zero))$
6. $\forall(nocc(E, seq(X, Y), zero) \Leftrightarrow \neg id(X, E) \wedge nocc(E, Y, zero))$
7. $\forall(nocc(E, seq(X, Y), succ(N)) \Leftrightarrow id(X, E) \wedge nocc(E, Y, N) \vee \neg id(X, E) \wedge nocc(E, Y, succ(N)))$

Signature: $member : Nat \times Seq$

Axioms:

8. $\forall(member(E, empty) \Leftrightarrow false)$
9. $\forall(member(E, seq(X, Y)) \Leftrightarrow id(E, X) \vee member(E, Y))$

A *term of type* τ is defined inductively as follows: (a) a variable of type τ is a term of type τ, (b) a constant of type τ is a term of type τ, (c) if f is a function symbol of type $\tau_1 \times ... \times \tau_n \rightarrow \tau$ and t_i is a term of type τ_i, with $i = 1..n$, then $f(t_1, ..., t_n)$ is a term of type τ. A *typed (well-formed) formula* is defined inductively as follows: (a) if p is a predicate symbol of type $\tau_1 \times ... \times \tau_n$ and t_i is a term of type τ_i, with $i = 1..n$, then $p(t_1, ..., t_n)$ is a typed atomic formula, (b) if F and G are typed formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$, $F \Leftrightarrow G$, $F \Leftarrow G$ are typed formulas and (c) if F is a typed formula then $\forall(F)$ and $\exists(F)$ are typed formulas.

The *Herbrand universe* of a theory $\mathcal{T}$, $\mathcal{U}_\mathcal{T}$, is the set of all possible (well-typed) ground terms we can construct by taking the set of constants and function symbols declared in $\mathcal{T}$. For instance, $\mathcal{U}_{\mathcal{T}_1} = \{zero, succ(zero), ...\}$. The *Herbrand base* of a theory $\mathcal{T}$, $\mathcal{B}_\mathcal{T}$, is the set of all possible (well-typed) ground atoms we can construct by taking all predicate symbols in $\mathcal{T}$ with arguments in $\mathcal{U}_\mathcal{T}$.

A *partial interpretation* I is a consistent set of literals whose atoms are in the Herbrand base of $\mathcal{T}$. A *total interpretation* is a partial interpretation that contains every atom of the Herbrand base or its negation. A ground literal is true in I when this is in I and it is false in I when its complement is in I.

The *instantiation of an axiom* Ax in a theory $\mathcal{T}$, denoted by $Ax_\mathcal{H}$, is the set of all formulae which results from substituting terms in the Herbrand universe of $\mathcal{T}$ for variables in Ax in every possible way. A *Herbrand instantiation* of a theory $\mathcal{T}$, $\mathcal{T}_\mathcal{H}$, is the set of formulae obtained by substituting each axiom in $\mathcal{T}$ by its instantiation. An instantiated theory may well be infinite.

A *total model* $\mathcal{M}$ of a theory $\mathcal{T}$ is a total interpretation which satisfies every axiom in $\mathcal{T}_{\mathcal{H}}$. Given a theory $\mathcal{T}$ with a total model $\mathcal{M}$, we have:

$\mathcal{T} \models \exists(G)$ iff $\mathcal{T} \models G\theta$, for some θ
$\mathcal{T} \models \forall(G)$ iff $\mathcal{T} \models G\theta$, for every θ
$\mathcal{T} \models L$ iff $L \in \mathcal{M}$
$\mathcal{T} \models \neg F$ iff $\mathcal{T} \not\models F$
$\mathcal{T} \models F \wedge G$ iff $\mathcal{T} \models F$ and $\mathcal{T} \models G$
$\mathcal{T} \models F \vee G$ iff $\mathcal{T} \models F$ or $\mathcal{T} \models G$
$\mathcal{T} \models F \Rightarrow G$ iff $\mathcal{T} \models \neg F \vee G$
$\mathcal{T} \models F \Leftrightarrow G$ iff $\mathcal{T} \models F \Rightarrow G$ and $\mathcal{T} \models G \Rightarrow F$

with L as a ground literal, F and G as two formulae in the language of $\mathcal{T}$ and θ as a ground substitution with terms taken from $\mathcal{U}_{\mathcal{T}}$.

3.1. The consistency of a theory

In relation to the consistency problem, we recall at this point the concepts of call-consistency [34,52] and first-order program [53].

The concept of *call-consistency* is primarily defined for general logic programs. A general logic program is *call-consistent* if no predicate calls itself through an odd number of negative goals [52]. It is proved that call-consistency guarantees the consistency in two-valued logic [34,52]. Sato extends this result to first-order logic theories $\Delta \cup \Gamma$, being Δ a base theory which defines a set of primitive predicates in a consistent way and Γ a call-consistent first-order program [53]. A *first-order program* is a finite set of predicate definitions, one for each predicate. A *predicate definition* is a formula of the form $\forall(p(x_1, ..., x_n) \Leftrightarrow F)$ where $x_1, ..., x_n$ are distinct variables and F a first-order formula whose free variables are among $x_1, ..., x_n$. The purpose of first-order programs is to allow a user to define new predicates on top of old ones. For a first-order program Γ, we can define the *signed dependency* among its predicate symbols [34,52], denoted by $p >_+ q$ (p depends on q positively) and $p >_- q$ (p depends on q negatively), respectively, as the least relation satisfying:

$p >_+ q$ iff $p \gg_+ q$, or for some r, $p \gg_+ q$ and $r >_+ q$, or $p \gg_- q$ and $r >_- q$
$p >_- q$ iff $p \gg_- q$, or for some r, $p \gg_+ q$ and $r >_- q$, or $p \gg_- q$ and $r >_+ q$

where $p \gg_+ q (p \gg_- q)$ iff there is a predicate definition $p(X) \Leftrightarrow F$ in Γ such that q occurs positively (negatively) in F.

A first-order program Γ is *call-consistent* if we never have $p >_- p$ for any p in Γ. According to [53], a first-order logic theories $\Delta \cup \Gamma$ is *consistent* if Δ is consistent and Γ is call-consistent.

It is not difficult to see that the concept of predicate definition given in [53] is similar to our concept of predicate specification. At this point, we focus on *total and non-overlapping* predicate specifications. We say that a predicate specification $S(r)$ in a theory $\mathcal{T}$ is *total* iff, for every atom $A \in \mathcal{B}_{\mathcal{T}}$ defined on r, there exists one axiom in $\mathcal{T}_{\mathcal{H}}$ whose left-hand side is A. We say that a predicate specification $S(r)$ is *non-overlapping* iff, for any pair of axioms $Ax_1, Ax_2 \in S(r)$, $lhs(Ax_1)_{\mathcal{H}} \cap lhs(Ax_2)_{\mathcal{H}} = \emptyset$. For instance, every predicate specification in $\mathcal{T}_1$ (Example 1) is total and non-overlapping.

A total and non-overlapping predicate specification can be translated to an equivalent predicate definition by following the method given by Clark for the completion of a predicate in logic programming [11]. For instance, $S(member)$ in $\mathcal{T}_1$ (Example 1) can be translated to the following predicate definition, being $idSeq$ the identity for sequences of natural numbers.

Signature: $member : Nat \times Seq$
Axioms:
$\forall(member(E, L) \Leftrightarrow$
$\exists X, Y(idSeq(L, empty) \wedge false \vee$
$idSeq(L, seq(X, Y)) \wedge (id(E, X) \vee member(E, Y))))$

Any theory in our formalism can now be translated to a first-order theory $\Delta \cup \Gamma$ as in [53]. In order to do so, we design Δ as a base theory which includes the types of the original theory and, for each of these types, a total and non-overlapping specification of an identity relation. Then, for every predicate symbol in the original theory distinct from an identity, we translate its (total and non-overlapping) specification into an equivalent predicate definition in Γ. For instance, theory $\mathcal{T}_1$ in Example 1 can be translated to a theory $\Delta \cup \Gamma$, being Δ the following consistent base theory:

Base Theory Δ

Types

Nat	generated by	$zero : \rightarrow Nat$ $succ : Nat \rightarrow Nat$
Seq	generated by	$empty : \rightarrow Seq$ $seq : Nat \times Seq \rightarrow Seq$

Predicates

Signature: $id : Nat \times Nat$

Axioms:

1. $id(zero, zero) \Leftrightarrow true$
2. $\forall(id(succ(X), zero) \Leftrightarrow false)$
3. $\forall(id(zero, succ(Y)) \Leftrightarrow false)$
4. $\forall(id(succ(X), succ(Y)) \Leftrightarrow id(X, Y))$

Signature: $idSeq : Seq \times Seq$

Axioms:

5. $idSeq(empty, empty) \Leftrightarrow true$
6. $\forall(idSeq(seq(X, Y), empty) \Leftrightarrow false)$
7. $\forall(idSeq(empty, seq(V, W)) \Leftrightarrow false)$
8. $\forall(idSeq(seq(X, Y), seq(V, W)) \Leftrightarrow id(X, V) \wedge idSeq(Y, W))$

---

and Γ the following call-consistent first-order program:

First-order Program Γ

Signature: $nocc : Nat \times Seq \times Nat$

Axioms:

9. $\forall(nocc(E, L, K) \Leftrightarrow$
   $\exists X, Y, N(idSeq(L, empty) \wedge id(K, N) \wedge id(N, zero) \vee$
   $idSeq(L, seq(X, Y)) \wedge id(K, zero) \wedge \neg id(X, E) \wedge nocc(E, Y, zero) \vee$
   $idSeq(L, seq(X, Y)) \wedge id(K, succ(N)) \wedge ((id(X, E) \wedge nocc(E, Y, N)) \vee$
   $(\neg id(X, E) \wedge nocc(E, Y, succ(N))))))$

Signature: $member : Nat \times Seq$

Axioms:

10. $\forall(member(E, L) \Leftrightarrow$
    $\exists X, Y(idSeq(L, empty) \wedge false \vee$
    $idSeq(L, seq(X, Y)) \wedge (id(E, X) \vee member(E, Y))))$

When checking an assertion, we want to know if this is either true or false (undefined situations are not of our interest). This is the reason why we are interested in theories having total models.

Let $\mathcal{T}$ be a theory where every predicate specification is total and non-overlapping. Then, $\mathcal{T}$ has a total model if it can be translated to a theory $\Delta \cup \Gamma$, being Δ a consistent base theory and Γ a call-consistent first-order program (see Theorem 1 in the appendix).

In what follows, we define two key concepts of our formalism: the *form of a formula* and the *construction of evaluations*.

3.2. The form of a formula

The *form of a formula* is a key concept for developing unfold/fold transformations in a completely automatic manner. In our proposal, the form of a formula is encoded by an f-formula. An *f-formula* F is the expression which results from replacing each variable occurrence in a quantifier-free formula F by the symbol _. The symbol _ is called *f-variable*. For instance, $id(\_, zero)$ and $\neg id(succ(\_), \_) \wedge nocc(\_, \_, zero)$ are f-formulae. Colloquially, we will say that a formula F "*is of the form*" F. Every atom A in a formula F will produce an expression A in F called *f-atom*. In a similar way, every term t in F will produce an expression t in F called *f-term*. We will call *f-axiom* Ax to the f-formula resulting from an axiom Ax after removing quantification.

Let t and s be two f-terms. We say that s *subsumes* t (equivalently, t is subsumed by s), denoted by $\mathsf{t} \prec \mathsf{s}$, iff (a) $\mathsf{t} = f(\mathsf{t}_1, ..., \mathsf{t}_n)$ ($n = 0$ means that f is a constant) and s is an f-variable or (b) $\mathsf{t} = f(\mathsf{t}_1, ..., \mathsf{t}_n)$ and $\mathsf{s} = f(\mathsf{s}_1, ..., \mathsf{s}_n)$ and there exists a non-empty subset $S \subseteq \{1..n\}$ such that (b.1) $\mathsf{t}_i \prec \mathsf{s}_i$ for every $i \in S$ and (b.2) t_j is identical to s_j for every $j \in (\{1..n\} - S)$. For instance,

$$seq(zero, \_) \prec seq(\_, \_) \qquad seq(zero, \_) \prec \_$$

The subsumption relation $\prec$ can be extended to f-atoms in a similar way. For instance, $id(\_, succ(zero)) \prec id(\_, \_)$ and $id(zero, succ(zero)) \prec id(zero, succ(\_))$. Given a chain of f-atoms $\alpha = \mathsf{A}_1 \prec \mathsf{A}_2 \prec ... \prec \mathsf{A}_k$, we say that A_1 is the *lower f-atom* in α, A_k is the *upper f-atom* in α and any A_j, with $j = 2..k$, is a *non-lower f-atom* in α.

We say that an f-atom A *occurs explicitly* in a theory $\mathcal{T}$ iff there is some axiom in $\mathcal{T}$ containing an atom whose form is equal to A.

An f-atom A_j is said to be *induced* from two f-atoms A_i and A_k which occur explicitly in a theory $\mathcal{T}$ iff A_j does not occur explicitly in $\mathcal{T}$ and $\mathsf{A}_i \prec \mathsf{A}_j \prec \mathsf{A}_k$. We say that a chain $\alpha = \mathsf{A}_1 \prec \mathsf{A}_2 \prec ... \prec \mathsf{A}_k$ is *complete* in a theory $\mathcal{T}$ iff (a) A_1 occurs explicitly in $\mathcal{T}$ and there is not any f-atom occurring explicitly in $\mathcal{T}$ lesser than A_1, (b) A_k occurs explicitly in $\mathcal{T}$ and there

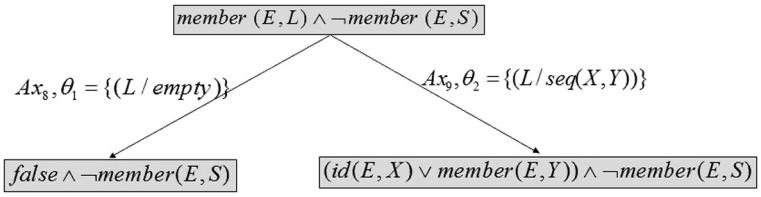

Fig. 2. Derivation tree.

is not any f-atom occurring explicitly in $\mathcal{T}$ greater than A_k and (c) every f-atom occurring between A_1 and A_k is either an f-atom occurring explicitly in $\mathcal{T}$ or an induced f-atom from A_1 and A_k.

For instance, the following chain is complete in $\mathcal{T}_1$ (see Example 1) (the induced f-atoms have been marked with the symbol *):

$$\alpha \quad = \quad id(zero, zero) \prec id(zero, \_)^* \prec id(\_, \_)$$

We say that an f-atom A *occurs* in a theory $\mathcal{T}$ iff A is member of some complete chain of $\mathcal{T}$. For instance, the f-atoms $id(zero, zero)$, $id(zero, \_)$ and $id(\_, \_)$ occur in $\mathcal{T}_1$ (Example 1).

In what follows, we will denote by lhs(Ax) the f-atom which results from $lhs(Ax)$, the left-hand side of Ax and by rhs(Ax) the f-formula which results from $rhs(Ax)$, the right-hand side of Ax.

Let Ax_i and Ax_j be two any axioms in a specification $S(r)$. Let $t_{i,1}..t_{i,n}$ and $t_{j,1}..t_{j,n}$ be the parameters in $lhs(Ax_i)$ and $lhs(Ax_j)$ respectively. We say that $S(r)$ is *regular* iff, for every $k = 1..n$, either $\mathrm{t}_{i,k} = \mathrm{t}_{j,k}$ or $\mathrm{t}_{i,k} \not\approx \mathrm{t}_{j,k}$ and $\mathrm{t}_{j,k} \not\approx \mathrm{t}_{i,k}$, being $\mathrm{t}_{i,k}$ and $\mathrm{t}_{j,k}$ the respective term patterns of $t_{i,k}$ and $t_{j,k}$. For instance, $S(nocc)$ is not regular in $\mathcal{T}_1$ (Example 1). Considering $lhs(Ax_5)$ and $lhs(Ax_6)$, we can see that neither $t_{5,3} = t_{6,3}$ nor $t_{5,3} \not\approx t_{6,3}$ and $t_{6,3} \not\approx t_{5,3}$ (in fact, $t_{5,3} = \_$ and $t_{6,3} = zero$). This also applies to $lhs(Ax_5)$ and $lhs(Ax_7)$. We can always transform a non-regular specification into a regular one by specializing its axioms. In what follows, we show the regular version of $S(nocc)$. As we can see, axioms 5 and 6 in the regular version result from the specialization of axiom 5 in the non-regular version (Example 1).

5. $\forall(nocc(E, empty, zero) \Leftrightarrow true)$
6. $\forall(nocc(E, empty, succ(N)) \Leftrightarrow false)$
7. $\forall(nocc(E, seq(X, Y), zero) \Leftrightarrow \neg id(X, E) \wedge nocc(E, Y, zero))$
8. $\forall(nocc(E, seq(X, Y), succ(N)) \Leftrightarrow id(X, E) \wedge nocc(E, Y, N) \vee \neg id(X, E) \wedge nocc(E, Y, succ(N)))$

3.3. The construction of evaluations

The *construction of evaluations* is another key concept in our proposal.

We will use the notation F^P_Q to denote the *replacement* of a sub-formula P in F by a formula Q. Let $Ax = \forall(r(X) \Leftrightarrow R(Y))$ be an axiom in a theory, A an atom in a formula $\forall(G)$ and θ the most general unifier (mgu) of A and $lhs(Ax)$, the left-hand side of Ax. Then, the formula $\forall(G\theta^{A\theta}_{R(Y)\theta})$ is called *derivation* from A in $\forall(G)$ using Ax and θ.

A *derivation tree* for a formula $\forall(G)$ in the context of a theory $\mathcal{T}$, denoted by $\Delta_{\forall(G),\mathcal{T}}$, is a tree satisfying the following conditions: (a) the root is G and (b) each branch β is a sequence $G_0 = G$, G_1, ... of formulae. The construction of β is based on the use of a sequence Ax_1, Ax_2, ... of (variants of) axioms in $\mathcal{T}$ and a sequence θ_1, θ_2, ... of mgu's such that each G_{i+1} is derived from G_i using Ax_{i+1} and θ_{i+1}.

Every branch β in $\Delta_{\forall(G),\mathcal{T}}$ derives an *instance* of $\forall(G)$, $\forall(G\theta)$, being θ the sequential composition of the mgu's used in β. A derivation tree is *finite* if it has a finite number of branches and each branch is finite. The last formula in a finite branch is called leaf formula. Let $\Delta_{\forall(G),\mathcal{T}}$ be a finite derivation tree and let $\forall(G\theta_1)$,...,$\forall(G\theta_k)$ be the set of instances of $\forall(G)$ derived from the respective branches β_1,...,β_k in $\Delta_{\forall(G),\mathcal{T}}$. We say that $\Delta_{\forall(G),\mathcal{T}}$ is *complete* iff $\forall(G)_{\mathcal{H}} = \forall(G\theta_1)_{\mathcal{H}} \cup ... \cup \forall(G\theta_k)_{\mathcal{H}}$. For instance, the derivation tree shown in Fig. 2 is complete. Each derivation step has been annotated with the identification of the axiom in its corresponding theory (i.e. $\mathcal{T}_1$) and the mgu.

We say that a complete derivation tree $\Delta_{\forall(G),\mathcal{T}}$ is a *positive evaluation* of $\forall(G)$ in $\mathcal{T}$, denoted by $\mathcal{T} \vdash \forall(G)$, if every branch in $\Delta_{\forall(G),\mathcal{T}}$ ends with the leaf formula *true* (success branch). We say that $\Delta_{\forall(G),\mathcal{T}}$ is a *negative evaluation* of $\forall(G)$, denoted by $\mathcal{T} \vdash \neg\forall(G)$, if, at least, one branch in $\Delta_{\forall(G),\mathcal{T}}$ ends with the leaf formula *false* (failure branch). The existence in $\Delta_{\forall(G),\mathcal{T}}$ of, at least, one success branch means that $\exists(G)$ follows from $\mathcal{T}$, denoted by $\mathcal{T} \vdash \exists(G)$. When every branch in $\Delta_{\forall(G),\mathcal{T}}$ is a failure branch then $\neg\exists(G)$ follows from $\mathcal{T}$, denoted by $\mathcal{T} \vdash \neg\exists(G)$.

We extend predicate specifications with evaluation signatures. An *evaluation signature* for a predicate symbol r, $ev(r)$, is a signature for r extended with annotations $\downarrow$ in its parameters. For instance, $member : Nat \times Seq\downarrow$ is an evaluation signature for the predicate *member*. We say that an atom A defined upon r *instantiates an evaluation signature* $ev(r)$ if every parameter annotated with $\downarrow$ in $ev(r)$ corresponds with a *ground term* in A. For instance, the atom $member(E, empty)$ instantiates $member : Nat \times Seq\downarrow$ but the atom $member(E, L)$ does not instantiate $member : Nat \times Seq\downarrow$.

We say that a predicate specification $S(r)$ is *evaluable* in a theory $\mathcal{T}$ wrt an evaluation signature $ev(r)$ iff, for every atom A that instantiates $ev(r)$, there exists an evaluation of $\forall(A)$ in $\mathcal{T}$.

The writing of evaluable specifications is a responsibility of the specifier. In the theory shown in Example 2 (page 9), the existence of evaluations is due to the use of well-founded recursive definitions: the parameters annotated with $\downarrow$ in each evaluation signature have been considered as recursive parameters in the corresponding specification. Also, we have used well-founded recursive definitions but in an indirect manner in the writing of $S(odd)$ and $S(even)$ in theory $\mathcal{T}_3$ (Example 3, page 10). In any way, *we do not impose any specific manner of ensuring the existence of evaluations. We simply assume the existence of these*. In the literature, we can find multiple references which may assist in this sense [39,1,16].

It is useful to have some syntactic criterion for checking whether a theory can preserve the construction of evaluations. The *propagation of an evaluation signature* $ev(r)$ to an axiom Ax is the expression which results from the following sequence of actions:

1. Every parameter t in the left-hand side of Ax corresponding with a parameter $\downarrow$ in $ev(r)$ is replaced by $t\downarrow$.
2. Every parameter t in the right-hand side of Ax is replaced by $t\downarrow$ if $t\downarrow$ occurs in the left-hand side of Ax.

We will assume that a term $t\downarrow$ propagates the annotation $\downarrow$ to each of its sub-terms. For instance, $seq(X, Y)\downarrow$ implies $X\downarrow$ and $Y\downarrow$. For instance, the propagation of the evaluation signature $member : Nat \times Seq\downarrow$ (Example 2) to the axioms in $S(member)$ produces the following expressions:

$$\forall(member(E, empty\downarrow) \Leftrightarrow false)$$
$$\forall(member(E, seq(X, Y)\downarrow) \Leftrightarrow id(E, X\downarrow) \vee member(E, Y\downarrow))$$

We say that an atom A defined upon r with annotations $\downarrow$ *matches an evaluation signature* $ev(r)$ if every parameter annotated with $\downarrow$ in $ev(r)$ corresponds with a parameter annotated with $\downarrow$ in A. For instance, the atom $member(E, Y\downarrow)$ matches $ev(member)$.

We say that a theory $\mathcal{T}$ *preserves the construction of evaluations* if, after propagating its evaluation signatures to the axioms, all resulting atoms match evaluation signatures in $\mathcal{T}$. For instance, the theory shown in Example 2 preserves the construction of evaluations.

3.4. The formalization of the theory

Once introduced the main auxiliary concepts, we focus on the formalization of a theory.

Definition 1 *(Theory).* A theory $\mathcal{T}$ is an axiomatization of a set of types and a set of predicate specifications satisfying the following properties:

(1) Every predicate specification $S(r)$ in $\mathcal{T}$ is total and non-overlapping. Additionally, $\mathcal{T}$ can be translated to a theory $\Delta \cup \Gamma$, being Δ a consistent base theory and Γ call-consistent first-order program.

(2) Every axiom Ax in any predicate specification is of the form $\forall(r(X) \Leftrightarrow R(Y))$ where $Y \subseteq X$.

(3) Every predicate specification $S(r)$ is regular.

(4) For every axiom Ax, lhs(Ax), the form of the left-hand side of Ax, is a lower f-atom occurring in $\mathcal{T}$ and every f-atom in the rhs(Ax), the form of the right-hand side of Ax, is a non-lower f-atom occurring in $\mathcal{T}$.

(5) $\mathcal{T}$ preserves the construction of evaluations.

(6) Every predicate specification $S(r)$ is evaluable for each of its evaluation signatures.

Property (1) is needed to ensure the consistency of $\mathcal{T}$ *(see Theorem 1 in the appendix). Properties (2), (3) and (4) are intended to satisfy two additional requirements: the method for synthesizing logic programs from* $\mathcal{T}$ *must be terminating (see Theorem 3 in the appendix) and must preserve semantics (see Theorem 4 in the appendix). Properties (5) and (6) are intended to satisfy a third requirement: the execution of a synthesized program must be terminating (see Theorem 5 in the appendix). It is important to remark that both property (1) (in particular, its second part) and property (6) require human ingenuity. All other properties are syntactical and therefore verifiable by machine.*

Example 2 shows a version of $\mathcal{T}_1$ (Example 1) which satisfies all properties in Definition 1.

Example 2 *(Theory).*

Theory $\mathcal{T}_2$

Types

Nat	generated by	$zero :\rightarrow Nat \quad succ : Nat \rightarrow Nat$
Seq	generated by	$empty :\rightarrow Seq \quad seq : Nat \times Seq \rightarrow Seq$

Predicates

Predicate signature: $id : Nat \times Nat$

Evaluation signature: 1. $id : Nat \times Nat\downarrow$, 2. $id : Nat\downarrow \times Nat$

Axioms:
1. $id(zero, zero) \Leftrightarrow true$ 2. $\forall(id(succ(X), zero) \Leftrightarrow false)$
3. $\forall(id(zero, succ(Y)) \Leftrightarrow false)$ 4. $\forall(id(succ(X), succ(Y)) \Leftrightarrow id(X, Y))$

Predicate signature: $nocc : Nat \times Seq \times Nat$
Evaluation signature: 1. $nocc : Nat \times Seq\downarrow \times Nat$
Axioms:
5. $\forall(nocc(E, empty, zero) \Leftrightarrow true)$
6. $\forall(nocc(E, empty, succ(N)) \Leftrightarrow false)$
7. $\forall(nocc(E, seq(X, Y), zero) \Leftrightarrow \neg id(X, E) \wedge nocc(E, Y, zero))$
8. $\forall(nocc(E, seq(X, Y), succ(N)) \Leftrightarrow id(X, E) \wedge nocc(E, Y, N) \vee \neg id(X, E) \wedge nocc(E, Y, succ(N)))$

Predicate signature: $member : Nat \times Seq$
Evaluation signature: 1. $member : Nat \times Seq\downarrow$
Axioms:
9. $\forall(member(E, empty) \Leftrightarrow false)$
10. $\forall(member(E, seq(X, Y)) \Leftrightarrow id(E, X) \vee member(E, Y))$

An alternative sufficient condition for showing the consistency of a theory based on the notion of evaluation can be considered at this point.

Every theory which satisfies properties (1) and (6) in Definition 1 has a total model (see Theorem 2 in the appendix).

A formula whose atoms all match evaluation signatures in a theory has derivation trees containing only atoms which instantiate evaluation signatures in that theory (see Lemma 1 in the appendix).

A formula has an evaluation in a theory if each of its atoms instantiates an evaluation signature in that theory (see Lemma 2 in the appendix).

It is interesting to remark that stratification [37] is not a requirement for our theories. In Example 3, we show a non-stratified theory which satisfies all properties in Definition 1.

Example 3 *(Non-stratified theory).*

Theory $\mathcal{T}_3$

Types
Nat generated by $zero :\rightarrow Nat$ $succ : Nat \rightarrow Nat$

Predicates
Predicate signature: $odd : Nat$
Evaluation signature: 1. $odd : Nat\downarrow$
Axioms:
1. $odd(zero) \Leftrightarrow false$ 2.$\forall(odd(succ(X)) \Leftrightarrow \neg even(succ(X)))$

Predicate signature: $even : Nat$
Evaluation signature: 1. $even : Nat\downarrow$
Axioms:
3. $even(zero) \Leftrightarrow true$ 4. $even(succ(zero)) \Leftrightarrow false$
5. $\forall(even(succ(succ(X))) \Leftrightarrow \neg odd(X))$

Once defined the class of logic theories, we precise the concept of assertion definition.

Definition 2 *(Assertion definition).* Let $\mathcal{T}$ be a theory written according to Definition 1. An *assertion definition* $D(r)$ is a formula of the form $\forall X(r(X) \Leftrightarrow Q\,Y\,R(X, Y))$ where r is the relation defined by the assertion $Q\,Y\,R(X, Y)$, X is a set of variables to be instantiated at runtime by ground terms, $Q\,Y$ is a set of quantified variables on infinite domains (Q is a universal or existential quantifier), $R(X, Y)$ a quantifier-free formula whose atoms all have the form of non-lower f-atoms occurring in $\mathcal{T}$ and every atom in $R(X\downarrow, Y)$ matches an evaluation signature in $\mathcal{T}$.

In Example 4, we show a set of assertion definitions. These definitions have been written from a set of typical assertions taken from the literature [26,42,18,45]. Additionally, we have included a set of predicate specifications in the appendix (Example 10) in order to clarify the formal semantics of such definitions.

Example 4 *(Assertion definitions).*

$$
\begin{array}{lcl}
D(subset) & = & \forall(subset(L,S) \Leftrightarrow \forall E(member(E,L) \Rightarrow member(E,S))) \\
D(perm) & = & \forall(perm(L,S) \Leftrightarrow \forall E,N(nocc(E,L,N) \Leftrightarrow nocc(E,S,N))) \\
D(plateau) & = & \forall(plateau(S,N) \Leftrightarrow \exists S(prefix(S,L) \wedge eqs(S) \wedge size(S,N))) \\
D(swap) & = & \forall(swap(L,R) \Leftrightarrow \exists S,M(append(R,S,L) \wedge append(M,R,L))) \\
D(ordprefix) & = & \forall(ordprefix(L,N) \Leftrightarrow \exists S(oprefix(S,L) \wedge size(S,N))) \\
D(partition) & = & \forall(partition(L,S,R,P) \Leftrightarrow (append(S,R,L) \wedge lts(S,P) \wedge ges(R,P))) \\
D(ord) & = & \forall(ord(L) \Leftrightarrow \forall N,E,F((elem(N,L,E) \wedge elem(succ(N),L,F)) \\
 & & \qquad \Rightarrow le(E,F))) \\
D(sublist) & = & \forall(sublist(S,L) \Leftrightarrow \exists R(append(S,R,L)))
\end{array}
$$

(Informal semantics) The relation $subset(L,S)$ is true iff every element E which is member of L is also member of S [18], the relation $perm(L,S)$ is true iff for any element E, the number of occurrences of E in L is equal to the number of occurrences of E in S and vice versa [26,42,45], the relation $plateau(S,N)$ is true iff S is a subsequence of L with size N whose values are all equal [26], the relation $swap(L,R)$ is true iff R is a subsequence of L and the remaining part of L can be swapped with R again resulting L, the relation $oprefix(L,N)$ is true iff there is some ordered prefix in L of size N, the relation $partition(L,P)$ is true iff the pivot P partitions L into a subsequence S where every element is less than P and a subsequence R where every element is greater than or equal to P [26,18], the relation $ord(S)$ is true iff the elements in S are in ascending order [26,18] and the relation $sublist(S,L)$ is true iff S is a prefix of L.

At this point, we can precise the concepts of assertion satisfaction and assertion evaluation.

Definition 3 *(Assertion satisfaction).* Let $\mathcal{T}$ be a theory written according to Definition 1, $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X,Y))$ an assertion definition written in the language of $\mathcal{T}$ and θ a ground substitution for X. We say that an assertion $r(X)\theta$ is satisfied in $\mathcal{T}$ if and only if $\mathcal{T} \models Q\,Y\,R(X,Y)\theta$.

Definition 4 *(Assertion evaluation).* Let $\mathcal{T}$ be a theory written according to Definition 1, $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X,Y))$ an assertion definition written in the language of $\mathcal{T}$ and θ a ground substitution for X. We say that an assertion $r(X)\theta$ has a positive evaluation in $\mathcal{T}$ if and only if $\mathcal{T} \vdash Q\,Y\,R(X,Y)\theta$. We say that $r(X)\theta$ has a negative evaluation in $\mathcal{T}$ if and only if $\mathcal{T} \vdash \neg Q\,Y\,R(X,Y)\theta$.

Definition 5 *(Inferred pattern).* Let F be a quantifier-free formula in NNF written in the language of a theory $\mathcal{T}$ with free variables $x_1, ..., x_n$. Let F be the f-formula corresponding to F where every variable x_i in F has been translated to an f-variable $\_i$ in F and let r be a relation symbol not occurring in $\mathcal{T}$. Then, the f-formula $\mathsf{r}(\_1,...,\_n) \Leftrightarrow \mathsf{F}(\_1,...,\_n)$ is said to be a *pattern inferred from* F.

For instance, $subset_1(\__1, \__2, \__3) \Leftrightarrow (member(\__1, \__2) \wedge \neg member(\__1, \__3))$ is a pattern inferred from the formula $member(E,L) \wedge \neg member(E,S)$.

Definition 6 *(Auxiliary specification).* Let $\mathsf{A} \Leftrightarrow \mathsf{F}$ be a pattern inferred from a quantifier-free formula F and Ev a set of evaluation signatures, one signature for each relation symbol occurring in F. From these elements, we can construct an auxiliary definition as follows:

1. Ev is propagated to the respective atoms in F.
2. Each f-variable in F is replaced by the respective variable in F.
3. The replacements in 1 and 2 are propagated to A. This step allows to infer both the predicate signature and the evaluation signature.
4. Finally, we close the resulting formula in 3 with a universal quantifier and then we remove all its evaluation annotations.

For instance, the following auxiliary specification has been generated from the pattern $subset_1(\__1, \__2, \__3) \Leftrightarrow (member(\__1, \__2) \wedge \neg member(\__1, \__3))$ and the set of evaluation signatures, $Ev = \{member : Nat \times Seq\downarrow\}$. The pattern has been inferred from a formula $member(E,L) \wedge \neg member(E,S)$.

Predicate signature: $subset_1 : Nat \times Seq \times Seq$
Evaluation signature: 1. $subset_1 : Nat \times Seq\downarrow \times Seq\downarrow$
Axioms:
$\forall(subset_1(E,L,S) \Leftrightarrow member(E,L) \wedge \neg member(E,S))$

Auxiliary specifications can be used to extend theories in a conservative way (see Corollary 3 in the appendix).

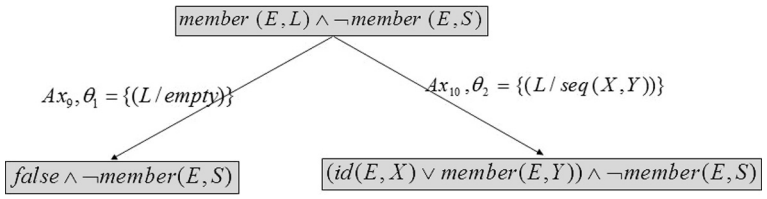

Fig. 3. Unfolding step. Complete derivation tree satisfying the unfolding condition.

4. Unfold/fold transformations

This section defines a particular unfold/fold transformation step based on the concept of similarity between a formula and an f-formula (pattern). This transformation step has been designed to avoid any human intervention. Internally, the unfold/fold transformation step is structured by an unfolding step followed by a folding step to each formula generated in the unfolding step.

4.1. The unfolding step

The goal of the unfolding step is to derive a set of formulae similar to the original one. As we said at the beginning of this paper, termination is one of the requirements we impose to our synthesis method. The satisfaction of this requirement is directly related to the manner of developing unfolding steps.

Definition 7 *(Unfolding condition).* Let G be a quantifier-free formula in the language of a theory $\mathcal{T}$. We say that G satisfies the *unfolding condition* wrt a complete derivation tree $\Delta_{\forall(G),\mathcal{T}}$ iff every derivation in $\Delta_{\forall(G),\mathcal{T}}$ has been done from a same atom and every derived instance in $\Delta_{\forall(G),\mathcal{T}}$ is only composed of atoms whose f-atoms occur in $\mathcal{T}$.

It is not difficult to realize that the unfolding condition imposes restrictions upon the form of the instances we can derive by unfolding steps. This aspect is relevant for ensuring termination in the synthesis process.

Definition 8 *(Unfolding step).* Let G be a quantifier-free formula in the language of a theory $\mathcal{T}$ which satisfies the unfolding condition wrt a complete derivation tree $\Delta_{\forall(G),\mathcal{T}}$. We say that the set of formulae $unf(G) = \{F_1, ..., F_k\}$ is the *unfolding* of G via $\Delta_{\forall(G),\mathcal{T}}$ if and only if $unf(G)$ is the set of all leaf formulae in $\Delta_{\forall(G),\mathcal{T}}$.

For instance, let $G = member(E, L) \wedge \neg member(E, S)$ be a quantifier-free formula in the language of $\mathcal{T}_2$ (see Example 2) and $\Delta_{\forall(G),\mathcal{T}_2}$ the complete derivation tree shown in Fig. 3. As we can see, every derivation in that tree has been done from a same atom and every derived instances is only composed of atoms whose f-atoms occur in $\mathcal{T}_2$. The derived instances are:

$$member(E, empty) \wedge \neg member(E, S)$$
$$member(E, seq(X, Y)) \wedge \neg member(E, S)$$

The unfolding of G via the complete derivation tree shown in Fig. 3 is:

$$\begin{aligned} unf(G) \quad = \quad & \{\ false \wedge \neg member(E, S), \\ & (id(E, X) \vee member(E, Y)) \wedge \neg member(E, S)\ \} \end{aligned}$$

It is possible to write assertion definitions which do not satisfy the unfolding condition. The following definition is an example:

$$D(magic) \quad = \quad \forall(magic(X) \Leftrightarrow \exists Y(id(X, succ(Y)) \wedge id(succ(X), Y)))$$

As we can verify, the unfolding of its right-hand side from the atom $id(X, succ(Y))$ derives instances containing f-atoms which do not occur in $\mathcal{T}_2$ (see specification of *id* in $\mathcal{T}_2$, Example 2):

instance 1: $id(zero, succ(Y)) \wedge \underline{id(succ(zero), Y)}$

instance 2: $id(succ(Z), succ(Y)) \wedge \underline{id(succ(succ(Z)), Y)}$

In a similar way, the unfolding of its right-hand side from the atom $id(succ(X), Y)$ also generates instances containing f-atoms which do not occur in $\mathcal{T}_2$:

instance 1: $\underline{id(X, succ(zero))} \wedge id(succ(X), zero)$

instance 2: $id(succ(X), succ(succ(Z))) \wedge \underline{id(succ(succ(Z)), succ(Z))}$

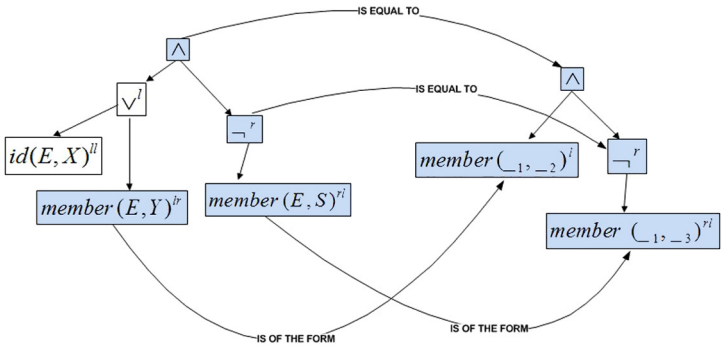

Fig. 4. Similarity between formula F and pattern K.

For such definitions, the synthesis method does not ensure termination. We have explored some alternative formulations for the unfolding condition but, at this moment, such alternatives are not sufficiently general. In addition, the integration of these alternatives in the synthesis method complicates the design of this in a considerable manner. Based on our experience, we can affirm that the writing of assertion definitions which do not satisfy the unfolding condition is not the norm but, when an assertion definition is problematic, the specifier has to rewrite it. For instance, the following assertion definition is a rewriting of $D(magic)$ which satisfies the unfolding condition:

$$\forall(magic(X) \Leftrightarrow \exists Y\,(id(X, succ(Y)) \wedge id(M, Y) \wedge id(M, succ(X))))$$

The derived instances from its right-hand side $G = id(X, succ(Y)) \wedge id(M, Y) \wedge id(M, succ(X))$ are:

$$id(X, succ(Y)) \wedge id(zero, Y) \wedge id(zero, succ(X))$$
$$id(X, succ(Y)) \wedge id(succ(Z), Y) \wedge id(succ(Z), succ(X))$$

and the resulting unfolding is:

$$\begin{aligned} unf(G) \quad = \quad & \{\, id(X, succ(Y)) \wedge id(zero, Y) \wedge false \\ & \quad id(X, succ(Y)) \wedge id(succ(Z), Y) \wedge id(Z, X)\,\} \end{aligned}$$

Finally, we remark the main properties of the unfolding step.

The unfolding step preserves semantics (see Lemma 3 in the appendix).

The unfolding step preserves evaluations (see Lemma 4 in the appendix).

4.2. The folding step

The goal of the folding step is to replace sub-formulae by atoms with the purpose of introducing "recursive calls" into synthesized programs. As we have seen in Sect. 1, a set of patterns (f-formulae) is needed for solving this task in an automatic way. In order to do so, these patterns (more precisely, their right-hand sides) have to be similar to the corresponding sub-formulae.

Similarity is a relationship between a quantifier-free formula F and an f-formula K. We say that F is similar to K if F "includes" a sub-formula identical in form to K. To discover this sub-formula, we will assume that F and K are expressed in form of binary trees (see Fig. 4). By construction, every node in (the tree) F will be either an atom or a boolean constant or a logical connective. Every node in (the tree) K will be either an f-atom or a boolean constant or a logical connective. A sequence of letters in superscript is also included in each node for representing its location in the tree. The location of a node in the tree is codified as the father node's location followed by a suffix l or r depending on whether the node in question is located at the left-hand side or at the right-hand side of its father node respectively. Formally, the expression n^s denotes the node n at position s in the tree. The root node is the only node which has an "empty" location (see an example in Fig. 4).

Definition 9 *(Similarity).* We say that a quantifier-free formula F is *similar* to an f-formula K iff (1) F is atomic, K is atomic and F is of the form K or (2) F is not atomic, K is not atomic and (2.a) for every node n^s in K containing a logical connective or boolean constant, there exists a node m^s in F containing the same logical connective or boolean constant and (2.b) for every node n^t in K containing an f-atom A, there exists a node m^u in F such that m^u contains an atom of the form A and $t = u$ or t is a prefix of u.

Fig. 4 shows in a graphical way the similarity between the quantifier-free formula $F = (id(E, X) \vee member(E, Y)) \wedge \neg member(E, S)$ (at the left-hand side of the figure) and the f-formula $\mathsf{K} = member(\__1, \__2) \wedge \neg member(\__1, \__3)$ (at the right-hand side of the figure).

The existence of similarity between a formula F and an f-formula K partitions the set of atoms in F into those which are mapped by similarity (e.g. $member(E,Y)$ and $member(E,S)$) and those which are not mapped by similarity. These latter ones are referred in this paper as the *remaining atoms* (e.g. $id(E,X)$). We say that F is *identical in form* to K if F is *similar* to K and the set of remaining atoms is empty. For instance, the formula $G = member(E,Y) \wedge \neg member(E,S)$ is identical to $\mathsf{K} = member(\_1,\_2) \wedge \neg member(\_1,\_3)$.

Going back to our example, both $(id(E,X) \vee member(E,Y)) \wedge \neg member(E,S)$ and $false \wedge \neg member(E,S)$ are similar to $member(\_1,\_2) \wedge \neg member(\_1,\_3)$ and $false \wedge \neg member(\_1,\_3)$ respectively. In the first case, a remaining atom $id(E,X)$ is generated while in the second one, the formula is identical to the corresponding f-formula and therefore the set of remaining atoms is empty.

As we will see in the following two definitions, if the set of remaining atoms is empty, the folding step can be solved in a direct way (*Direct folding*). Otherwise, a *rewriting* of the formula is needed in order to facilitate the completion of the folding step (*Folding via rewriting*). Roughly, the rewriting of a formula is a meaning-preserving transformation which replaces the set of remaining atoms by a combination of boolean constants. For instance, the rewriting of $F = (id(E,X) \vee member(E,Y)) \wedge \neg member(E,S)$ produces the following formula:

$$\begin{array}{rcl} rew(F) & = & (true \vee member(E,Y)) \wedge \neg member(E,S) \quad \wedge \quad id(E,X) \quad \vee \\ & & (false \vee member(E,Y)) \wedge \neg member(E,S) \quad \wedge \quad \neg id(E,X) \end{array}$$

Definition 10 *(Rewriting).* We say that a formula $rew(F)$ is the rewriting of a quantifier-free formula F with respect to the right-hand side K of a pattern if and only if:

$$\begin{array}{rcll} rew(F) & = & F & \text{if } F \text{ is identical to } \mathsf{K} \\ rew(F) & = & F_1 \wedge H_1 \vee \ldots \vee F_{2^k} \wedge H_{2^k} & \text{if } F \text{ is similar but not identical to } \mathsf{K} \end{array}$$

In this latter case, k is the number of remaining atoms in F (resulting from calculating the similarity between F and K) and F_i the formula which results from replacing in F all remaining atoms by a combination of k boolean constants. In order to preserve equivalence, all boolean combinations have to be considered (that is, $i = 1..2^k$) and, in addition, we have to add to each F_i a conjunction of literals H_i according to the following criterion: if a remaining atom A in F was replaced by the constant $true$ then A is added to H_i and if A was replaced by $false$ then $\neg A$ is added to H_i. Every rewriting ends with the simplification of each F_i preserving the boolean constants and logical connectives which occur in the original formula (i.e. F).

For instance, the rewriting of $F = (id(E,X) \vee member(E,Y)) \wedge \neg member(E,S)$ wrt the right-hand side of a pattern $subset_1(\_1,\_2,\_3) \Leftrightarrow member(\_1,\_2) \wedge \neg member(\_1,\_3)$ is (after simplifying each F_i):

$$\begin{array}{rcl} rew(F) & = & \underbrace{true \wedge \neg member(E,S)}_{F_1} \quad \wedge \quad \underbrace{id(E,X)}_{H_1} \quad \vee \\ & & \underbrace{member(E,Y) \wedge \neg member(E,S)}_{F_2} \quad \wedge \quad \underbrace{\neg id(E,X)}_{H_2} \end{array}$$

From the formulae F_i and H_i in $rew(F)$ we can infer new patterns. In our example, these new patterns are:

Sub-formula	Pattern
F_1	$subset_2(\_1,\_2) \Leftrightarrow true \wedge \neg member(\_1,\_2)$
H_1	$subset_3(\_1,\_2) \Leftrightarrow id(\_1,\_2)$
H_2	$subset_4(\_1,\_2) \Leftrightarrow \neg id(\_1,\_2)$

Rewriting preserves semantics (see Lemma 5 in the appendix).

Rewriting preserves evaluations (see Lemma 6 in the appendix).

By construction, every sub-formula F_i in $rew(F)$ is identical to the right-hand side of a pattern F_i and also every sub-formula H_i in $rew(F)$ is identical to the right-hand side of a pattern H_i. The first task in the folding step is to *instantiate patterns* for folding both F_i and H_i in $rew(F)$. This instantiation is done as follows: each f-variable in the right-hand side of the pattern $\mathsf{F}_i(\mathsf{H}_i)$ is replaced by the respective variable in $F_i(H_i)$. This replacement is propagated to the left-hand side of the pattern. For instance,

Sub-formula	Pattern instantiation
F_1	$subset_2(E,S) \Leftrightarrow true \wedge \neg member(E,S)$
F_2	$subset_1(E,Y,S) \Leftrightarrow member(E,Y) \wedge \neg member(E,S)$
H_1	$subset_3(E,X) \Leftrightarrow id(E,X)$
H_2	$subset_4(E,X) \Leftrightarrow \neg id(E,X)$

Definition 11 *(Folding step).* We say that a formula $fold(F)$ is the folding of a quantifier-free formula F if and only if:

Folding via rewriting:
$fold(F) = f_1 \wedge h_1 \vee ... \vee f_{2^k} \wedge h_{2^k}$ if $rew(F) = F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k}$ and $f_i \Leftrightarrow F_i$ is a pattern instantiation and h_i is a conjunction of the left-hand sides of k pattern instantiations.

Direct folding:
$fold(F) = f$ if $rew(F) = F$ and $f \Leftrightarrow F$ is a pattern instantiations.

For instance, the folding via rewriting of

$$F = \underbrace{true \wedge \neg member(E,S)}_{F_1} \wedge \underbrace{id(E,X)}_{H_1} \vee \underbrace{member(E,Y) \wedge \neg member(E,S)}_{F_2} \wedge \underbrace{\neg id(E,X)}_{H_2}$$

with pattern instantiations

$$subset_1(E,Y,S) \Leftrightarrow member(E,Y) \wedge \neg member(E,S)$$
$$subset_2(E,S) \Leftrightarrow true \wedge \neg member(E,S)$$
$$subset_3(E,X) \Leftrightarrow id(E,X)$$
$$subset_4(E,X) \Leftrightarrow \neg id(E,X)$$

is

$$fold(F) = \underbrace{subset_2(E,S)}_{f_1} \wedge \underbrace{subset_3(E,X)}_{h_1} \vee \underbrace{subset_1(E,Y,S)}_{f_2} \wedge \underbrace{subset_4(E,X)}_{h_2}$$

and the direct folding of

$$F = \underbrace{false \wedge \neg member(E,S)}_{F}$$

with a pattern instantiation

$$subset_5(E,S) \Leftrightarrow false \wedge \neg member(E,S)$$

is

$$fold(F) = \underbrace{subset_5(E,S)}_{f}$$

Finally, we remark the main properties of the folding step.

The folding step preserves semantics (see Corollary 5 in the appendix).

The folding step preserves evaluations (see Corollary 6 in the appendix).

4.3. The unfold/fold transformation step

Once defined by separate both the unfolding step and the folding step, we define the manner of composing them in what we call a transformation step.

The unfold/fold transformation step is a function which takes as inputs an auxiliary specification $S(r_i)$ and a pattern and produces as outputs a recursive version of $S(r_i)$ called *synthesized specification* and denoted by $S^*(r_i)$ and a set of patterns from which we may derive new auxiliary specifications. For instance, the unfold/fold transformation step of the auxiliary specification $S(subset_1) = \forall(subset_1(E,L,S) \Leftrightarrow member(E,L) \wedge \neg member(E,S))$ wrt the pattern $subset_1(\_1,\_2,\_3) \Leftrightarrow member(\_1,\_2) \wedge \neg member(\_1,\_3)$ produces the following synthesized specification $S^*(subset_1)$:

$$\forall(subset_1(E,seq(X,Y),S) \Leftrightarrow (subset_2(E,S) \wedge subset_3(E,X)) \vee (subset_1(E,Y,S) \wedge subset_4(E,X)))$$
$$\forall(subset_1(E,empty,S) \Leftrightarrow subset_5(E,S))$$

and the following set of patterns:

$subset_1(\__1, \__2, \__3) \Leftrightarrow member(\__1, \__2) \wedge \neg member(\__1, \__3)$
$subset_2(\__1, \__2) \Leftrightarrow true \wedge \neg member(\__1, \__2)$
$subset_3(\__1, \__2) \Leftrightarrow id(\__1, \__2)$
$subset_4(\__1, \__2) \Leftrightarrow \neg id(\__1, \__2)$
$subset_5(\__1, \__2) \Leftrightarrow false \wedge \neg member(\__1, \__2)$

from which we can derive four *new* auxiliary specifications:

$S(subset_2) = \forall(subset_2(X, Y) \Leftrightarrow true \wedge \neg member(X, Y))$
$S(subset_3) = \forall(subset_3(X, Y) \Leftrightarrow id(X, Y))$
$S(subset_4) = \forall(subset_4(X, Y) \Leftrightarrow \neg id(X, Y))$
$S(subset_5) = \forall(subset_5(X, Y) \Leftrightarrow false \wedge \neg member(X, Y))$

The unfold/fold transformation step is internally structured by an unfolding step followed by a folding step for each formula generated in the unfolding step and finally a combination of the results in order to construct the synthesized specification.

The synthesized specification is constructed in the following manner. The left-hand sides of the axioms in the synthesized specification result from applying to the left-hand side of the auxiliary specification the substitutions computed in the unfolding step. In our example, the substitutions computed in $unf(member(E, L) \wedge \neg member(E, S))$ are $\theta_1 = L/empty$ and $\theta_2 = L/seq(X, Y)$ and the application of these substitutions to the left-hand side of the auxiliary specification produces (see Subsect. 4.1):

(1) $subset_1(E, empty, S) \qquad subset_1(E, seq(X, Y), S)$

The right-hand sides of the axioms in the synthesized specification result from applying a folding step to each formula generated by the unfolding step. In our example, the formulae generated by the unfolding step are (see Subsect. 4.1):

(2) $false \wedge \neg member(E, S) \quad (id(E, X) \vee member(E, Y)) \wedge \neg member(E, S)$

The folding of each formula in (2) produces (see Subsect. 4.2):

(3) $subset_5(E, S) \qquad (subset_2(E, S) \wedge subset_3(E, X)) \vee (subset_1(E, Y, S) \wedge subset_4(E, X))$

Finally, the combination of (1) and (3) generates the synthesized specification $S^*(subset_1)$.

$$\forall(subset_1(E, seq(X, Y), S) \Leftrightarrow (subset_2(E, S) \wedge subset_3(E, X)) \vee (subset_1(E, Y, S) \wedge subset_4(E, X)))$$
$$\forall(subset_1(E, empty, S) \Leftrightarrow subset_5(E, S))$$

Finally, we remark the main properties of the unfold/fold transformation.

The unfold/fold transformation preserves semantics (see Corollary 7 in the appendix).

The unfold/fold transformation preserves evaluations (see Corollary 8 in the appendix).

4.4. Inference of patterns

As we have seen, the folding step needs to infer patterns from f-formulae. The inference of such f-formulae is a non-deterministic task. For example, from a formula $(id(E, X) \vee member(E, Y)) \wedge \neg member(E, S)$, we can infer two possible f-formulae:

$member(\__1, \__2) \wedge \neg member(\__1, \__3) \qquad id(\__1, \__2) \wedge \neg member(\__1, \__3)$

A manner of reducing non-determinism is by establishing a (partial) order among the f-formulae.

We denote by $def(p)$ the set of all predicates directly or indirectly used in the definition of a predicate p in a given theory. For instance, in $\mathcal{T}_2$ (Example 2):

$$def(id) = \{id\},\ def(nocc) = \{nocc, id\},\ def(member) = \{member, id\}$$

The *definition level of a predicate* p in a theory can be algorithmically calculated as follows:

if there is not any predicate r such that $r \neq p \wedge r \in def(p)$ then
$level(p) = 1$
elsif there is some predicate r such that $r \neq p \wedge r \in def(p) \wedge p \in def(r)$ then
$level(p) = level(r)$
else
$l = max\{level(r) \mid r \neq p \wedge r \in def(p)\}$
$level(p) = l + 1$

For instance, according to this algorithm, the definition levels assigned to the predicate symbols in $\mathcal{T}_2$ (Example 2) are:

$$level(id) = 1,\ level(nocc) = 2,\ level(member) = 2$$

The notion of definition level can be easily extended to f-atoms and f-formulae. We will assume that the boolean constants *true* and *false* have *definition levels* equal to 0. The *definition level of an f-atom* is equal to the definition level of the predicate symbol occurring in that f-atom. The *definition level of an f-formula* K is equal to the tuple of definition levels corresponding to its f-atoms and boolean constants taken from left to right. For instance, the definition level of $member(\__1, \__2) \wedge \neg member(\__1, \__3)$ is $(2, 2)$, the definition level of $id(\__1, \__2) \wedge \neg member(\__1, \__3)$ is $(1, 2)$ and the definition level of $false \wedge \neg member(\__1, \__3)$ is $(0, 2)$.

Given two f-formulae F and G whose definition levels are $L_1 = (l_1, ..., l_n)$ and $L_2 = (l'_1, ..., l'_n)$ respectively, we say that G *is preferred to* F if and only if $l_i \leq l'_i$ for every $i \in \{1..n\}$. We will use the relation of preference in situations where it is possible to infer multiple patterns. For example, from a formula $(id(E, X) \vee member(E, Y)) \wedge \neg member(E, S)$, it is possible to infer two f-formulae:

(1) $id(\__1, \__2) \wedge \neg member(\__1, \__3)$ $\quad L_1 = (1, 2)$
(2) $member(\__1, \__2) \wedge \neg member(\__1, \__3)$ $\quad L_2 = (2, 2)$

However (2) is preferred to (1). Therefore (2) is the selected f-formula in the inference process.

5. The synthesis method

The synthesis of an assertion definition is defined as a finite sequence of unfold/fold transformation steps. Fig. 5 shows the synthesis method in diagrammatic form.

Each transformation step "consumes" an auxiliary specification $S(r_i)$ and "produces" a synthesized specification $S^*(r_i)$ and possibly new patterns from which we may derive new auxiliary specifications to be transformed in subsequent steps. This synthesis process is finite because, by construction, it is not possible to infer an unbounded amount of new patterns (see Theorem 3 in Appendix).

The last step in the synthesis method extracts a positive logic program from the set of synthesized specifications. By construction, every synthesized specification $S^*(r_i)$ (see Subsect. 4.3) may contain axioms constructed by folding via rewriting $\forall(r_i \Leftrightarrow f_1 \wedge h_1 \vee ... \vee f_{2^k} \wedge h_{2^k})$ or axioms constructed by direct folding $\forall(r_i \Leftrightarrow f)$. By definition of rewriting, $(h_i)_{\mathcal{H}} \cap (h_j)_{\mathcal{H}} = \emptyset$, with $i, j \in \{1..2^k\}$ and $i \neq j$. Therefore, each axiom $\forall(r_i \Leftrightarrow f_1 \wedge h_1 \vee ... \vee f_{2^k} \wedge h_{2^k})$ can be rewritten into the equivalent set of axioms:

$$\forall(r_i \Leftrightarrow f_1 \wedge h_1)$$
$$...$$
$$\forall(r_i \Leftrightarrow f_{2^k} \wedge h_{2^k})$$

From the resulting set of axioms, we can derive the corresponding set of Horn clauses by simply taking the if-part of each axiom.

$$\forall(r_i \Leftarrow f_1 \wedge h_1)$$
$$...$$
$$\forall(r_i \Leftarrow f_{2^k} \wedge h_{2^k})$$

In case of direct folding:

$$\forall(r_i \Leftarrow f)$$

From the resulting set of clauses, we remove the clauses of the form $r_i \Leftarrow false$ and, in a transitive manner, all those clauses having r_i in their bodies. Finally, the clauses of the form $r_i \Leftarrow true$ are replaced by $r_i \Leftarrow$.

5.1. Synthesis example

We summarize the synthesis process with a complete example.

We begin the synthesis process from an assertion definition (*see Step 1 in Fig. 5*):

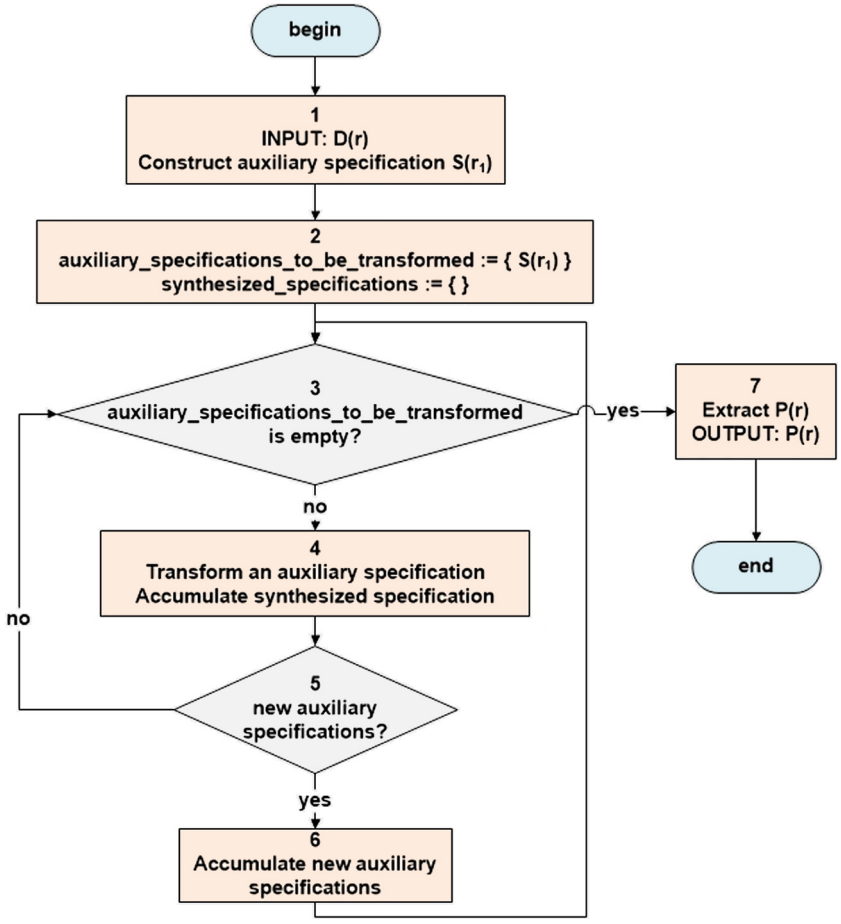

Fig. 5. The synthesis method.

$$D(subset) = \forall L, S(subset(L,S) \Leftrightarrow \forall E(member(E,L) \Rightarrow member(E,S)))$$

We construct the auxiliary specification $S(subset_1)$ (note that its right-hand side is in NNF) (*see Step 1 in Fig. 5*):

$$S(subset_1) = \forall E, L, S(subset_1(E,L,S) \Leftrightarrow (member(E,L) \wedge \neg member(E,S)))$$

Then, we enter in the synthesis cycle (*see Steps 2 and 3 in Fig. 5*). In a first iteration, we synthesize $S^*(subset_1)$ (see Step 4 in Fig. 5):

$$\begin{aligned}\forall(subset_1(E, seq(X,Y), S) \Leftrightarrow\ & (subset_2(E,S) \wedge subset_3(E,X)) \vee \\ & (subset_1(E,Y,S) \wedge subset_4(E,X)))\\ \forall(subset_1(E, empty, S) \Leftrightarrow\ & subset_5(E,S))\end{aligned}$$

The new patterns generated in this first iteration are:

$$\begin{aligned}&subset_1(\__1,\__2,\__3) \Leftrightarrow member(\__1,\__2) \wedge \neg member(\__1,\__3)\\ &subset_2(\__1,\__2) \Leftrightarrow true \wedge \neg member(\__1,\__2)\\ &subset_3(\__1,\__2) \Leftrightarrow id(\__1,\__2)\\ &subset_4(\__1,\__2) \Leftrightarrow \neg id(\__1,\__2)\\ &subset_5(\__1,\__2) \Leftrightarrow false \wedge \neg member(\__1,\__2)\end{aligned}$$

from which we can derive four *new* auxiliary specifications (*see Steps 5 and 6 in Fig. 5*):

$$\begin{aligned}&S(subset_2) = \forall(subset_2(X,Y) \Leftrightarrow true \wedge \neg member(X,Y))\\ &S(subset_3) = \forall(subset_3(X,Y) \Leftrightarrow id(X,Y))\\ &S(subset_4) = \forall(subset_4(X,Y) \Leftrightarrow \neg id(X,Y))\\ &S(subset_5) = \forall(subset_5(X,Y) \Leftrightarrow false \wedge \neg member(X,Y))\end{aligned}$$

In a second iteration, we synthesize $S^*(subset_2)$ (*see Steps 3 and 4 in Fig. 5*):

$$\forall(subset_2(E, seq(X, Y)) \Leftrightarrow (subset_2(E, Y) \wedge subset_4(E, X)) \vee (subset_3(E, X) \wedge subset_8)$$
$$\forall(subset_2(E, empty) \Leftrightarrow subset_7)$$

The new patterns generated in this second iteration are:

$$subset_7 \Leftrightarrow true \wedge \neg false$$
$$subset_8 \Leftrightarrow true \wedge \neg true$$

From these patterns, we can derive two *new* auxiliary specifications. The synthesis of these is trivial because the right-sides of both specifications are composed of boolean constants only (*see Steps 5 and 6 in Fig. 5*):

$$S(subset_7) = subset_7 \Leftrightarrow true \wedge \neg false$$
$$S(subset_8) = subset_8 \Leftrightarrow true \wedge \neg true$$

In a third iteration, we synthesize $S^*(subset_3)$ (*see Steps 3 and 4 in Fig. 5*):

$$subset_3(zero, zero) \Leftrightarrow subset_9$$
$$\forall(subset_3(succ(X), zero) \Leftrightarrow subset_{10})$$
$$\forall(subset_3(zero, succ(Y)) \Leftrightarrow subset_{11})$$
$$\forall(subset_3(succ(X), succ(Y)) \Leftrightarrow subset_3(X, Y))$$

The new patterns generated in this third iteration are:

$$subset_9 \Leftrightarrow true$$
$$subset_{10} \Leftrightarrow false$$

From these patterns we can derive two *new* auxiliary specifications. Again, the synthesis of these specifications is trivial (*see Steps 5 and 6 in Fig. 5*):

$$S(subset_9) = subset_9 \Leftrightarrow true$$
$$S(subset_{10}) = subset_{10} \Leftrightarrow false$$

In a fourth iteration, we synthesize $S^*(subset_4)$ (*see Steps 3 and 4 in Fig. 5*):

$$subset_4(zero, zero) \Leftrightarrow subset_{10}$$
$$\forall(subset_4(succ(X), zero) \Leftrightarrow subset_9)$$
$$\forall(subset_4(zero, succ(Y)) \Leftrightarrow subset_9)$$
$$\forall(subset_4(succ(X), succ(Y)) \Leftrightarrow subset_4(X, Y))$$

No new patterns are generated in this fourth iteration (*see Step 5 in Fig. 5*).
In a fifth iteration, we synthesize $S^*(subset_5)$ (*see Steps 3 and 4 in Fig. 5*):

$$\forall(subset_5(E, seq(X, Y)) \Leftrightarrow (subset_5(E, Y) \wedge subset_4(E, X)) \vee (subset_3(E, X) \wedge subset_{11}))$$
$$\forall(subset_5(E, empty) \Leftrightarrow subset_7)$$

The new patterns generated in this fifth iteration are:

$$subset_6 \Leftrightarrow false \wedge \neg false$$
$$subset_{11} \Leftrightarrow false \wedge \neg true$$

From these patterns, we can derive two *new* auxiliary specifications. Again, the synthesis of these specifications is trivial (*see Steps 5 and 6 in Fig. 5*):

$$S(subset_6) = subset_6 \Leftrightarrow false \wedge \neg false$$
$$S(subset_{11}) = subset_{11} \Leftrightarrow false \wedge \neg true$$

The last step in the synthesis process is the extraction of $P(subset_1)$ from the set of synthesized specifications $S^*(subset_1)$,...,$S^*(subset_{11})$. The final program $P(subset)$ is completed by adding the clause $\forall(subset(L, S) \Leftarrow subset_1(E, L, S))$ to $P(subset_1)$ (*see Step 7 in Fig. 5*):

Example 5 *(Synthesized program $P(subset)$).*

$$\forall(subset(L,S) \Leftarrow subset_1(E,L,S))$$
$$\forall(subset_1(E,seq(X,Y),S) \Leftarrow subset_2(E,S) \wedge subset_3(E,X))$$
$$\forall(subset_1(E,seq(X,Y),S) \Leftarrow subset_1(E,Y,S) \wedge subset_4(E,X))$$
$$\forall(subset_1(E,empty,S) \Leftarrow subset5(E,S))$$
$$\forall(subset_2(E,seq(X,Y)) \Leftarrow subset_2(E,Y) \wedge subset_4(E,X))$$
$$\forall(subset_2(E,empty) \Leftarrow subset_7)$$
$$subset_3(zero,zero) \Leftarrow subset_9$$
$$\forall(subset_3(succ(X),succ(Y)) \Leftarrow subset_3(X,Y))$$
$$\forall(subset_4(zero,succ(Y)) \Leftarrow subset_9)$$
$$\forall(subset_4(succ(X),succ(Y)) \Leftarrow subset_4(X,Y))$$
$$\forall(subset_4(succ(X),zero) \Leftarrow subset_9)$$
$$\forall(subset_5(E,seq(X,Y)) \Leftarrow subset_5(E,Y) \wedge subset_4(E,X))$$
$$subset_7 \Leftarrow$$
$$subset_9 \Leftarrow$$

Finally, we remark the main properties of the synthesis method.

The synthesis method is terminating (see Theorem 3 in the appendix).

The synthesis method preserves semantics (see Theorem 4 in the appendix).

The synthesis method preserves evaluations (see Theorem 5 in the appendix).

6. Experimentation

We have constructed an experimental synthesizer in order to validate the feasibility of our proposal. The synthesizer has been designed according to the typical architecture of a language processor: lexical analyzer, syntax analyzer, semantic analyzer and code generator. All these analyzers have been implemented in Java [31] using a parser generator called ANTLR [46]. The code generator generates SWI-PROLOG code [62]. Up to this moment, the synthesized code is not optimal. We plan to add a code optimizer to our prototype in the near future.

We have tested the synthesizer with a wide variety of assertion definitions. Some of these tests are shown in the following examples.

Example 6 *(Tests (I)).* For each assertion definition shown in Example 4, we report the number of iterations, auxiliary specifications and definite clauses generated in the synthesis process.

Assertion definition $D(r)$	Iterations	Auxiliary specifications (patterns)	Definite clauses $P(r)$
subset	5	11	14
perm	19	25	73
plateau	17	27	41
swap	22	28	51
ordprefix	18	24	48
partition	22	32	84
ord	24	32	38
sublist	5	7	15

Example 7 *(Tests (II)).* The following tests have been done in a computer Intel Core i7, CPU 2.10 GHz, RAM 6 GB with OS Windows 7 (64 bits). For each assertion definition in Example 4, we measure the mean synthesis time (in milliseconds) of 100 repetitive runs.

Assertion definition ($D(r)$)	Mean time (ms)
subset	3.4
perm	15.3
plateau	10.5
swap	13.2
ordprefix	9.5
partition	16.3
ord	9.3
sublist	4.9

Example 8 *(Tests (III)).* To give an idea about the number of formulae generated in intermediate steps of the synthesis process, we report, for each assertion definition shown in Example 4, the number of synthesized axioms just before starting the extraction of the logic program (step 7 in Fig. 5).

Assertion definition ($D(r)$)	Synthesized axioms (intermediate step)
subset	23
perm	100
plateau	73
swap	93
ordprefix	84
partition	144
ord	80
sublist	32

Example 9 *(Tests (IV)).* Knowing what types and predicates have been needed in the writing of an assertion definition may help the reader in valuating the expressivity of our approach. In the following table, for each assertion definition in Example 4, we show the types, predicates and total number of axioms we have needed in its formalization. Example 10 in the appendix shows the specification of each of these predicates.

Assertion definition	Types	Predicates	Number of axioms (predicates)
subset	$\{Nat, Seq\}$	$\{id, member\}$	6
perm	$\{Nat, Seq\}$	$\{id, nocc\}$	8
plateau	$\{Nat, Seq\}$	$\{id, prefix, eqs, size\}$	15
swap	$\{Nat, Seq\}$	$\{id, idSeq, append\}$	16
ordprefix	$\{Nat, Seq\}$	$\{id, le, oprefix, size\}$	21
partition	$\{Nat, Seq\}$	$\{id, idSeq, append, lt, ge, lts, ges\}$	28
ord	$\{Nat, Seq\}$	$\{id, le, elem\}$	12
sublist	$\{Nat, Seq\}$	$\{id, idSeq, append\}$	16

7. Checking assertions with synthesized logic programs

In this section, we justify the manner of checking assertions with synthesized logic programs by the method proposed in this paper.

Suppose an assertion definition $D(r) = \forall(r(X) \Leftrightarrow Q\,Y\,R(X,Y))$ and a positive logic program $P(r)$ which has been synthesized for checking ground atoms of the form $r(X)\theta$. We recall that the design of $P(r)$ depends on the quantifier Q in $D(r)$: $P(r)$ searches for refutations if $Q = \forall$ and it searches for proofs if $Q = \exists$. In particular, $P(r)$ is implemented by a clause $\forall(r(X) \Leftarrow r_1(X,Y))$ which defines r in terms of a new relation symbol r_1 and by a positive logic program $P(r_1)$ synthesized from a specification $S(r_1) = \forall X, Y(r_1(X,Y) \Leftrightarrow \neg R(X,Y))$ if $Q = \forall$ or from a specification $S(r_1) = \forall X, Y(r_1(X,Y) \Leftrightarrow R(X,Y))$ if $Q = \exists$.

For the purpose of checking an assertion $r(X)\theta$, we propose to compute $P(r) \cup \{G\}$, being G the goal $\Leftarrow r(X)\theta$. Again, we distinguish two cases.

For $Q = \forall$, we have that $D(r) = \forall X(r(X) \Leftrightarrow \forall Y R(X,Y))$. If the empty answer is computed for $P(r) \cup \{G\}$ then $r(X)\theta$ is a logical consequence of $P(r)$ what means that $\exists Y r_1(X,Y)\theta$ is a logical consequence of $P(r_1)$. By total correctness of $P(r_1)$, $\exists Y r_1(X,Y)\theta$ is true. By equivalence, $\exists Y \neg R(X,Y)\theta$ is true, or equivalently, $\forall Y R(X,Y)\theta$ is false and therefore, by equivalence, $r(X)\theta$ is false (*refutation*). Otherwise (i.e. no answer is computed for $P(r) \cup \{G\}$), $r(X)\theta$ is not a logical consequence of $P(r)$ what means that $\exists Y r_1(X,Y)\theta$ is not a logical consequence of $P(r_1)$. By total correctness of $P(r_1)$, $\exists Y r_1(X,Y)\theta$ is false. By equivalence, $\exists Y \neg R(X,Y)\theta$ is false, or equivalently, $\forall Y R(X,Y)\theta$ is true and therefore, by equivalence, $r(X)\theta$ is true (*impossibility of refutation*).

For $Q = \exists$, we have that $D(r) = \forall X(r(X) \Leftrightarrow \exists Y R(X,Y))$. If the empty answer is computed for $P(r) \cup \{G\}$ then $r(X)\theta$ is a logical consequence of $P(r)$ what means that $\exists Y r_1(X,Y)\theta$ is a logical consequence of $P(r_1)$. By total correctness of $P(r_1)$, $\exists Y r_1(X,Y)\theta$ is true. By equivalence, $\exists Y R(X,Y)\theta$ is true and therefore, by equivalence, $r(X)\theta$ is true (*proof*). Otherwise (i.e. no answer is computed for $P(r) \cup \{G\}$), $r(X)\theta$ is not a logical consequence of $P(r)$ what means that $\exists Y r_1(X,Y)\theta$ is not a logical consequence of $P(r_1)$. By total correctness of $P(r_1)$, $\exists Y r_1(X,Y)\theta$ is false. By equivalence, $\exists Y R(X,Y)\theta$ is false and therefore, by equivalence, $r(X)\theta$ is false (*impossibility of proof*).

The following table summarizes the checking of an assertion with a synthesized logic program.

Assertion definition	$P(r) \cup \{\Leftarrow r(X)\theta\}$	Conclusion
$D(r) = \forall X(r(X) \Leftrightarrow \forall Y R(X,Y))$	success	$r(X)\theta$ is *false*
$D(r) = \forall X(r(X) \Leftrightarrow \forall Y R(X,Y))$	failure	$r(X)\theta$ is *true*
$D(r) = \forall X(r(X) \Leftrightarrow \exists Y R(X,Y))$	success	$r(X)\theta$ is *true*
$D(r) = \forall X(r(X) \Leftrightarrow \exists Y R(X,Y))$	failure	$r(X)\theta$ is *false*

7.1. Examples

In a first example, let $D(subset)$ be the assertion definition (see Example 4) and $P(subset)$ (see Example 5) the corresponding synthesized logic program. To check an assertion such as $subset(L,S)\theta$, with $\theta = \{L/seq(zero, empty), S/empty\}$, we propose to compute $P(subset) \cup \{\Leftarrow subset(L,S)\theta\}$. As we can verify, the empty answer is computed for $P(subset) \cup \{\Leftarrow subset(seq(zero, empty), empty)\}$ what means that $subset(seq(zero, empty), empty)$ is a logical consequence of $P(subset)$. *Due to* $P(subset)$ *searches for refutations*, we have to conclude that $subset(seq(zero, empty), empty)$ is false.

In a second example, suppose $D(plateau)$ as the assertion definition (see Example 4) and $P(plateau)$ as the corresponding synthesized logic program (see Example 11 in the appendix). For checking an assertion such as $plateau(L,N)\theta$, with $\theta = \{S/seq(zero, empty), N/succ(succ(zero))\}$, we propose to compute $P(plateau) \cup \{\Leftarrow plateau(L,N)\theta\}$. As we can verify, no answer is computed for $P(plateau) \cup \{\Leftarrow plateau(seq(zero, empty), succ(succ(zero)))\}$ what means that the original assertion is not a logical consequence of $P(plateau)$. *Due to* $P(plateau)$ *searches for proofs*, we have to conclude that $plateau(seq(zero, empty), succ(succ(zero)))$ is false.

8. Related work

As we said at the beginning of this paper, the main characteristic of an assertion checker is its ability to evaluate logic formulae. In current technology [40,2,5,36], quantification is restricted to *finite* sets of elements. Most proposals provide direct support for this kind of quantification using keywords such as forall, all, exists and some. We illustrate this point with the following examples.

```
//Spec# assertion
  ensures Forall{int i in 0:index; old(this[i]) == this[i]}

//JML assertion
  /*@ invariant (\forall int i;  0 ≤ i && i < 4; 0 ≤ pin[i] && pin[i] ≤ 9); @*/

//Eiffel assertion
  (lower|..|upper).for_all((agent(i:INTEGER):BOOLEAN
          do Result := (item(i) = a.item(i)+b.item(i)) end))

//ADL assertion
  forall (long i : ADL_long_range(1,10))  i < 100; ;
```

Program synthesis is a valuable aid to overcome the restriction of the finite quantification. Program synthesis refers to the systematic elaboration of a program from a specification [14]. A lot of papers have been written on different program synthesis methods [28,16,19,47,20,49,3]. In constructive synthesis [12,6,22], a conjecture based on the specification is proven in a constructive manner and from this proof a program is extracted. In deductive synthesis [35], a program is deduced directly from the specification by means of transformations. In schema-based synthesis [21], a class of actual programs is abstracted in order to guide and enhance the synthesis process. Another approach is inductive synthesis [19] where a program is induced from an incomplete specification or from a set of examples. An interesting survey of the recent progress in program synthesis can be found in [27]. According to this classification, our synthesis method can be considered as a deductive method which generates positive (or definite) logic programs by means of unfold/fold transformations.

Two main lines of development have been observed for the unfold/fold transformation technique applied to logic programming. One focused on the design of transformation systems which preserve the semantics of definite or general programs and another line primarily aimed at the synthesis of positive logic programs from arbitrary first-order specifications. In relation to the first line of development, we find relevant works about the preservation of the least-Herbrand model semantics in definite logic programs [63,32], preservation of the perfect model semantics in locally stratified programs [38] and preservation of the well-founded semantics in general logic programs [55]. It is clear that our assertion language is more expressive than the class of definite logic programs and the class of stratified logic programs (see Example 3). On the other side, it is not difficult to see that a theory extended with an assertion definition can be translated to (the completion of) a general logic program [37] and therefore the results in [55] might as well be applied to our formalism. However, the transformation system proposed in [55] is clearly more restrictive than ours. To clarify this point, suppose we want to synthesize a logic program for checking $D(subset)$ (see Example 4). After translating $D(subset)$ to a general logic program according to [55], we have:

1. $id(zero, zero) \Leftarrow$
2. $\forall(id(succ(X), succ(Y)) \Leftarrow)$
3. $\forall(member(E, seq(X,Y)) \Leftarrow id(E,X))$
4. $\forall(member(E, seq(X,Y)) \Leftarrow member(E,Y))$
5. $\forall(p(L,S) \Leftarrow member(E,L) \wedge \neg member(E,S))$
6. $\forall(subset(L,S) \Leftarrow \neg p(L,S))$

Again, according to [55], the selected atom in an unfolding step must be a positive literal. However, as we can see, clause 6 does not have any positive literal in its body and therefore no transformation is possible from 6.

Suppose a second example where we want to synthesize a logic program for checking $D(nperm) = \forall(nperm(L,S) \Leftrightarrow \exists E, N \neg(nocc(E,L,N) \Leftrightarrow nocc(E,S,N)))$ ($nperm(L,S)$ is true iff L is not a permutation of S). After translating $D(nperm)$ to a general logic program according to [55], we have:

1. $id(zero, zero) \Leftarrow$
2. $\forall(id(succ(X), succ(Y)) \Leftarrow)$
3. $\forall(nocc(E, empty, zero) \Leftarrow)$
4. $\forall(nocc(E, seq(X,Y), succ(N)) \Leftarrow id(E,X) \wedge nocc(E,Y,N))$
5. $\forall(nocc(E, seq(X,Y), N) \Leftarrow \neg id(E,X) \wedge nocc(E,Y,N))$
6. $\forall(nperm(L,S) \Leftarrow nocc(E,L,N) \wedge \neg nocc(E,S,N))$
7. $\forall(nperm(L,S) \Leftarrow \neg nocc(E,L,N) \wedge nocc(E,S,N))$

After unfolding 6 from the atom $nocc(E,L,N)$ we have:

8. $\forall(nperm(empty, S) \Leftarrow \neg nocc(E,S,zero))$
9. $\forall(nperm(seq(X,Y),S) \Leftarrow id(X,E) \wedge nocc(E,Y,N) \wedge \neg nocc(E,S,succ(N)))$
10. $\forall(nperm(seq(X,Y),S) \Leftarrow \neg id(X,E) \wedge nocc(E,Y,N) \wedge \neg nocc(E,S,N))$

At this point, the folding of 10 wrt 6 is not allowed because, as defined in [55], every internal variable in 6 (i.e. E and N) must occur only in the sub-formula to be folded (i.e. $nocc(E,Y,N) \wedge \neg nocc(E,S,N)$) in 10. However, this condition does not hold for E (it also occurs in $\neg id(E,X)$) thus wasting an opportunity for folding.

In [53], two unfold/fold transformation systems for first-order programs are presented. The systems comprise an unfolding rule, a folding rule and a replacement rule. They are intended to work with a first-order theory Δ specifying the meaning of primitive relations, on top of which new relations are built by first-order programs Γ. They preserve both the provability relationship and the logical consequence relationship in three-valued logic. The specification formalism is similar to ours. In fact, any theory in our formalism can be translated to a theory $\Delta \cup \Gamma$. However, the transformation systems are different to ours in two main aspects: (a) they have been designed for proving properties (theoretical nature) but not for constructing a tool and (b) unfold/fold transformations are defined in a three-valued logic.

A transformation technique is introduced in [7] which, given definitions of a set of predicates p_i, synthesizes the definitions of new predicates $\tilde{p}_i$ which can be used, under a suitable refutation procedure, to compute the finite failure set of p_i. The technique exhibits some computational advantages such as the possibility of computing non-ground negative goals still preserving the capability of producing answers. Although the purpose of this work is clearly different to ours, the technique presents some similarities. For instance, predicates are specified by if-and-only-if axioms and the computation of the finite failure is done through positive logic programs which are obtained from the original ones by negating both sides of their predicate definitions. Negative literals are viewed as atoms defined upon new relation symbols. This design resembles our manner of synthesizing programs for checking assertion definitions having universal quantification in their right-hand sides.

In [51], unfold/fold transformations are applied to positive logic programs for avoiding both existential variables and variables which occur more than once in the body of a clause. The transformation of a program is conceived as an iterative process. Each iteration in this process is implemented by a sequential composition of an unfolding step, a definition step and folding step. Although the transformation method is similar to our synthesis method (see Sects. 4 and 5), its purpose is completely different. In [51], transformations are focused on program optimizations. Our transformations, by contrast, are focused on program synthesis. This difference is key for justifying why our specification formalism and conditions for ensuring termination are different to the respective ones in [51].

In [47], a transformational method is given both for proving properties in the context of a program and for synthesizing logic programs from implicit definitions. An implicit definition of a predicate $newp$ is a formula of the form: $\forall X(\exists Y F(X,Y) \Leftrightarrow \exists Z(H(X,Z) \wedge newp(X,Z))$ where every predicate in $F(X,Y)$ and $H(X,Z)$ is defined in a program P. Despite the constructive nature of the method, this includes a step (knows as Phase (5)) for which no algorithm exists.

An interesting work about model checking via unfold/fold transformation is given in [48]. Program transformations are used to prove in a semiautomatic way that a closed first-order logic formula holds in the perfect model of a locally stratified logic program. A strategy called UFS strategy is defined for guiding the application of the transformation rules. Two classes of formulae, called tree-typed formulas and tree-typed clausal formulas, and two classes of programs, called MR programs and DL programs are identified for which a deterministic strategy called dUFS strategy can be defined. We have compared the expressivity of these two classes of programs with the class of theories we propose in this paper and we can conclude than these latter ones are more expressive.

In [54], a deterministic algorithm for logic program synthesis is given. The algorithm takes a first-order program and produces a positive logic program (or may fail). The synthesis is transformational and is guided by a program schema called universal continuation form. In this paper, a first-order program is a finite set of first-order clauses of the form $A \Leftarrow F$, where F is empty or a formula in which any universally quantified sub-formula is of the form $\forall Y(F_1 \Rightarrow F_2)$ with $Y \subseteq variables(F_1)$. The compilation always terminates but may fail due to the lack of logical power. The compiled program is assured to be partially correct. In a similar line, a fully automatic program derivation is proposed in [13] for a class of

relations with specifications of the form $\forall(r(X,Z) \Leftrightarrow \forall Y(s(X,Y,Z) \Rightarrow a(Y,Z)))$, being r a relation symbol with a restrictive recursive specification and a an arbitrary relation.

There are many other methods for synthesizing logic programs where human interaction plays a key role [9,49,28,22]. These methods are less interesting for us.

Many verification problems can be solved by checking formulae in first-order logic. Nowadays, a new generation of SMT solvers is being developed with the ability of checking the satisfiability of certain kinds of quantified formulae [41]. For instance, Symplify was pioneer in this field by applying heuristics to the instantiation of quantifiers [15]. Moderns solvers such as CVC3 [24] and Z3 [4] have improved such heuristics by making use of patterns/triggers but they remain incomplete when facing with universal quantification due to the inability to develop proofs by induction [8]. Recently, SAT/SMT solvers have been used to synthesize imperative programs from program templates [60,58,57,59] and program assertions such as invariants and pre-conditions [25,56]. The use of templates becomes a relevant guide in order to make feasible the mechanization of the whole synthesis process.

It is usually recognized that folding steps during program transformation correspond to applications of inductive hypothesis during proofs by induction [48]. Therefore, we can say that our synthesis method makes use of induction for synthesizing positive logic programs. To guide the development of such proofs, we also make use of patterns but with a different purpose to SMT solvers: while these latter ones construct proofs by preserving equisatisfiability, our synthesizer has been designed to preserve logical equivalence.

Finally, this paper represents an extended and improved version of a previous paper [23]. The main differences with the proposal in [23] are: (1) the transformation step only contains one unfolding step. By contrast, this restriction does not exist in [23] (multiple unfolding steps can be done in a transformation step) and (2) in this paper, patterns are inferred on demand. However, in [23], the set of all possible patterns is inferred before starting the synthesis process. As our experiments show, the synthesizer proposed in this paper is more efficient than the one proposed in [23].

9. Conclusions

We have described a method based on unfold/fold transformations that synthesizes positive logic programs with the purpose of checking assertion definitions of the form $\forall X(r(X) \Leftarrow Q\,Y\,R(X,Y))$ where r is a predicate symbol, $Q\,Y$ is a set of quantified variables over infinite domains (Q is a universal or existential quantifier) and $R(X,Y)$ a quantifier-free first order formula. The method is completely automatic (it is terminating) and preserves total correctness. We have implemented an experimental synthesizer in Java [31,46] that synthesizes SWI-Prolog code [62]. We have tested the synthesizer with a variety of non-trivial assertions. Some of these tests have been reported in Sect. 6 to show that our proposal is feasible in practice.

The results of this paper constitute a first step towards the design of expressive assertion languages. To the best of our knowledge, there is not any proposal in the field of assertion checking that accepts some kind of infinite quantification in its assertion language [36,40,5,29,33,30,61,50].

The next step in our work is to integrate the synthesizer within the infrastructure of an assertion checker. We intend to use tools such as InterProlog [10] for solving the interface problem between the language in which the assertion is written (Prolog) and the programming language (for instance, Java).

Appendix

Theorem 1. *Let $\mathcal{T}$ be a theory where every predicate specification is total and non-overlapping. Then, $\mathcal{T}$ has a total model if it can be translated to a theory $\Delta \cup \Gamma$, being Δ a consistent base theory and Γ a call-consistent first-order program.*

Proof. (1) Let Δ be a theory composed of all types of $\mathcal{T}$ and, for each of these, a total and non-overlapping specification for an identity relation *id* such that $id \not>_{-} id$ in Δ (see definition of $>_{-}$ in Sect. 3).

(2) Let Γ be a theory composed of predicate definitions (see Sect. 3) only, one for each relation r in $\mathcal{T}$ distinct from an identity such that $r \not>_{-} r$ in Γ.

(3) From (1), the Herbrand universe of $\Delta \cup \Gamma$ coincides with the Herbrand universe of $\mathcal{T}$. In addition, from (1) and (2), Δ is a consistent (base) theory and Γ is a call-consistent first-order program [34,52]. Therefore, $\Delta \cup \Gamma$ has a model [53].

(4) From (3) and, by construction of $\Delta \cup \Gamma$, every model of $\Delta \cup \Gamma$ is also a model of $\mathcal{T}$.

(5) By totality of specifications in Δ and Γ and from (3), $\Delta \cup \Gamma$ has a total model.

(6) From (4) and (5), $\mathcal{T}$ has a total model.

Theorem 2. *Every theory $\mathcal{T}$ which satisfies properties (1) and (6) in Definition 1 has a total model.*

Proof. (1) By hypothesis, every specification in $\mathcal{T}$ is total. Hence, for every atom $A \in \mathcal{B}_{\mathcal{T}}$, there exists (at least) one derivation tree $\Delta_{A,\mathcal{T}}$ in $\mathcal{T}$.

(2) By hypothesis, every specification in $\mathcal{T}$ is evaluable wrt a set of evaluation signatures. Hence, $\Delta_{A,\mathcal{T}}$ is an evaluation.

(3) By hypothesis, every specification in $\mathcal{T}$ is non-overlapping. This means that $\Delta_{A,\mathcal{T}}$ is unique (either positive $\mathcal{T} \vdash A$ or negative $\mathcal{T} \vdash \neg A$).

(4) The conclusion is reached from (1) and (3) by taking $\mathcal{T} \models A$ if $\mathcal{T} \vdash A$ and $\mathcal{T} \models \neg A$ if $\mathcal{T} \vdash \neg A$.

Lemma 1. *Let $\mathcal{T}$ be a theory which preserves the construction of evaluations. Let $\forall(G)$ be a formula whose atoms all match evaluation signatures in $\mathcal{T}$. Then, every atom occurring in any formula in $\Delta_{\forall(G),\mathcal{T}}$ matches an evaluation signature in $\mathcal{T}$.*

Proof. By preservation of evaluations in $\mathcal{T}$, every derivation step from $\forall(G)$ will replace some atom in $\forall(G)$ by the instance of the right-hand side of some axiom in $\mathcal{T}$ whose atoms all match evaluation signatures in $\mathcal{T}$. This reasoning can be applied recurrently to the remaining derivations in $\Delta_{\forall(G),\mathcal{T}}$.

Corollary 1. *If A is an atom which instantiates an evaluation signature $ev(r)$, then $A\theta$ will instantiate $ev(r)$ for any substitution θ.*

Proof. The proof is immediate. By hypothesis, A instantiates $ev(r)$. Every ground parameter in A is preserved in $A\theta$. Therefore $A\theta$ will instantiate $ev(r)$.

Lemma 2. *Let $\mathcal{T}$ be a theory written according to Definition 1. A formula $\forall(G)$ has an evaluation $\Delta_{\forall(G),\mathcal{T}}$ in $\mathcal{T}$ if every atom A in $\forall(G)$ instantiates an evaluation signature in $\mathcal{T}$.*

Proof. (1) By assuming that every atom A in $\forall(G)$ instantiates an evaluation signature in $\mathcal{T}$ and that every specification in $\mathcal{T}$ is evaluable, we conclude that $\forall(A)$ has an evaluation $\Delta_{\forall(A),\mathcal{T}}$ in $\mathcal{T}$.

(2) By taking the derivations in $\Delta_{\forall(A),\mathcal{T}}$ as derivations in $\Delta_{\forall(G),\mathcal{T}}$, we have that every branch β in $\Delta_{\forall(G),\mathcal{T}}$ will end in a leaf formula of the form $\forall(G\theta^{A\theta}_{C})$ with C as a boolean constant ($true$ or $false$) and θ as the sequential composition of the mgu's applied to β.

(3) By Corollary 1, every atom in $\forall(G\theta^{A\theta}_{C})$ will instantiate an evaluation signature in $\mathcal{T}$.

(4) The reasoning (1)..(3) can be applied to the remaining atoms in $\forall(G\theta^{A\theta}_{C})$ in a recurrent manner. The recursion is terminating because the number of atoms in $\forall(G)$ is finite and this number decreases after each iteration. After completing all derivations, every branch in $\Delta_{\forall(G),\mathcal{T}}$ will end necessarily with a boolean constant.

From Lemma 2, we can derive the following corollary.

Corollary 2. *Let $\mathcal{T}$ be a theory written according to Definition 1. Let $\forall(G)$ be a formula whose atoms all instantiate evaluation signatures in $\mathcal{T}$ and $\forall(H)$ a formula exclusively composed of atoms that occur in $\forall(G)$. Then, $\forall(H)$ has evaluation in $\mathcal{T}$.*

Proof. Similar to the proof in Lemma 2 but replacing $\forall(G)$ by $\forall(H)$.

Corollary 3. *Let $S(r_i) = \forall(r_i(X) \Leftrightarrow G(X))$ be an auxiliary specification, being G a formula written in the language of a theory $\mathcal{T}$ with model $\mathcal{M}_{\mathcal{T}}$. Then, $\mathcal{T} \cup S(r_i)$ is a conservative extension of $\mathcal{T}$.*

Proof. By definition of auxiliary specification, r_i is a predicate symbol not occurring in $\mathcal{T}$. Hence, the extension $\mathcal{T} \cup S(r_i)$ does not interfere in $\mathcal{M}_{\mathcal{T}}$. Therefore, we can construct a model for $\mathcal{T} \cup S(r_i)$ in the following terms:

$$\mathcal{M}_{\mathcal{T}\cup S(r_i)} = \mathcal{M}_{\mathcal{T}} \cup \begin{cases} r_i(X)\theta & \text{if } \mathcal{T} \models G(X)\theta \\ \neg r_i(X)\theta & \text{if } \mathcal{T} \models \neg G(X)\theta \end{cases}$$

for every ground substitution θ of X.

Corollary 4. *Let $S(r_i)$ be an auxiliary specification written in the language of a theory $\mathcal{T}$. Then, every atom occurring in $S(r_i)$ matches an evaluation signature in $\mathcal{T} \cup S(r_i)$.*

Proof. By construction, every atom in the right-hand side of $S(r_i)$ matches an evaluation signature in $\mathcal{T}$. Again, by construction, the left-hand side of $S(r_i)$ matches an evaluation signature in $\mathcal{T} \cup S(r_i)$.

Lemma 3 *(Unfolding preserves semantics). Let $unf(G)$ be the unfolding of a formula G whose derivation tree is $\Delta_{G,\mathcal{T}}$. Let $\beta_1,...,\beta_n$ be the branches in $\Delta_{G,\mathcal{T}}$. Let $G\theta_i$ and F_i be the derived instance and the leaf formula respectively in any branch β_i with $i \in \{1..n\}$. Then,*

(1) $\mathcal{T} \models \forall(G\theta_i \Leftrightarrow F_i)$

(2) $\mathcal{T} \models \forall(G \Leftrightarrow \bigvee_i unf(G))$

with $\bigvee_i unf(G)$ as the disjunction of all formulae in $unf(G)$.

Proof. (1) $\mathcal{T} \models \forall(G\theta_i \Leftrightarrow F_i)$.

(1) By construction of β_i, every variable which occurs in F_i also occurs in $G\theta_i$. Hence, every substitution γ that converts $G\theta_i$ into a ground formula $G\theta_i\gamma$ also converts F_i into a ground formula $F_i\gamma$

(2) By definition of derivation, every derivation in β_i replaces an atom by an equivalent formula in $\mathcal{T}$.

(3) From (1) and (2), $\mathcal{T} \models G\theta_i\gamma \Leftrightarrow F_i\gamma$, for every γ. Therefore, $\mathcal{T} \models \forall(G\theta_i \Leftrightarrow F_i)$.

Proof. (2) $\mathcal{T} \models \forall(G(X) \Leftrightarrow \bigvee_i unf(G))$.

(1) By completeness of $\Delta_{G,\mathcal{T}}$, $G_{\mathcal{H}} = \bigcup_{i=1..n}(G\theta_i)_{\mathcal{H}}$.

(2) From (1), for every $G\varphi \in G_{\mathcal{H}}$, there exist θ_i with $i \in \{1..n\}$ and a ground substitution γ such that $G\theta_i\gamma = G\varphi$. This means that, for every φ, there exists $\theta_i\gamma$ such that $\varphi = \theta_i\gamma$. Therefore, $\mathcal{T} \models \forall(G \Rightarrow \bigvee_i unf(G))$.

(3) From (1), for every $G\theta_i\gamma \in \bigcup_{i=1..n}(G\theta_i)_{\mathcal{H}}$, there is a ground substitution φ such that $G\theta_i\gamma = G\varphi$. This means that, for every $\theta_i\gamma$, there exists φ such that $\theta_i\gamma = \varphi$. Therefore, $\mathcal{T} \models \forall(G \Leftarrow \bigvee_i unf(G))$.

(4) From (2) and (3), $\mathcal{T} \models \forall(G \Leftrightarrow \bigvee_i unf(G))$.

Lemma 4 *(Unfolding preserves evaluations). Let $unf(G)$ be the unfolding of a formula G whose derivation tree is $\Delta_{G,\mathcal{T}}$. Let $G\theta_i$ and F_i be the derived instance and the leaf formula respectively in any branch β_i in $\Delta_{G,\mathcal{T}}$.*

If every atom in $\forall(G\theta_i\delta)$ matches an evaluation signature in $\mathcal{T}$ then $\forall(F_i\delta)$ has evaluation in $\mathcal{T}$, being δ a substitution.

Proof. Every leaf formula F_i in $\Delta_{G,\mathcal{T}}$ is reached by applying derivations to G in a theory $\mathcal{T}$ that preserves evaluations. If every atom in $\forall(G\theta_i\delta)$ matches an evaluation signature in $\mathcal{T}$ then, by Lemma 1, every atom in any formula of $\Delta_{\forall(G\theta_i\delta),\mathcal{T}}$ also matches an evaluation signature in $\mathcal{T}$. In particular, this is true for the leaf formula $\forall(F_i\delta)$. Then, by Lemma 2, $\forall(F_i\delta)$ has evaluation in $\mathcal{T}$.

Lemma 5 *(Rewriting preserves semantics). Let F be a quantifier-free formula and $rew(F) = F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k}$ the formula which results from rewriting F. Then,*

$$\mathcal{T} \models \forall(F \Leftrightarrow rew(F))$$

Proof. (Proof of $\Leftarrow$) $\mathcal{T} \models \forall(F \Leftarrow rew(F))$.

For every ground substitution θ such that $(F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k})\theta$ is ground, if $\mathcal{T} \models (F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k})\theta$ then,

(1) By completeness of rewriting, there exists one $j \in \{1..2^k\}$ such that $\mathcal{T} \models (F_j \wedge H_j)\theta$.

(2) By definition of rewriting, $F^{A_1,...,A_k}_{C_1,...,C_k} = F_j$, being $C_1, ..., C_k$ a combination of boolean constants replacing in F the respective atoms $A_1, ..., A_k$. From (1), $\mathcal{T} \models F_j\theta$. Hence, $\mathcal{T} \models F\theta^{A_1\theta,...,A_k\theta}_{C_1,...,C_k}$.

(3) From (1), $\mathcal{T} \models H_j\theta$ and, by construction of H_j, each boolean constant C_i in $F\theta^{A_1\theta,...,A_k\theta}_{C_1,...,C_k}$ will represent the evaluation of $A_i\theta$ in $\mathcal{T}$. Hence, $\mathcal{T} \models F\theta$.

(4) From (1) .. (3), $\mathcal{T} \models F\theta \Leftarrow (F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k})\theta$, for every ground substitution θ. Therefore $\mathcal{T} \models \forall(F \Leftarrow rew(F))$.

(Proof of $\Rightarrow$) $\mathcal{T} \models \forall(F \Rightarrow rew(F))$.

For every ground substitution θ such that $F\theta$ is ground, if $\mathcal{T} \models F\theta$ then,

(1) $\mathcal{T} \models F\theta^{A_1\theta,...,A_k\theta}_{C_1,...,C_k}$ for some combination of boolean constants $C_1, ..., C_k$.

(2) By construction, every combination of boolean constants $C_1, ..., C_k$ have been considered in $F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k}$. Therefore, it must exist $j \in \{1..2^k\}$ such that $F\theta^{A_1\theta,...,A_k\theta}_{C_1,...,C_k} = F_j\theta$.

(3) From (2), and by construction of H_j, $\mathcal{T} \models H_j\theta$.

(4) From (2) and (3), $\mathcal{T} \models F_j\theta \wedge H_j\theta$. Therefore $\mathcal{T} \models (F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k})\theta$.

(5) From (1) .. (4), $\mathcal{T} \models F\theta \Rightarrow (F_1 \wedge H_1 \vee ... \vee F_{2^k} \wedge H_{2^k})\theta$, for every ground substitution θ. Therefore, $\mathcal{T} \models \forall(F \Rightarrow rew(F))$.

Lemma 6 *(Rewriting preserves evaluations). Let $rew(F)$ be the rewriting of F. If every atom in $\forall(F\delta)$ instantiates an evaluation signature in $\mathcal{T}$ then $\forall(rew(F)\delta)$ has an evaluation in $\mathcal{T}$, being δ a substitution.*

Proof. By definition of rewriting, $\forall(rew(F)\delta)$ is a formula exclusively composed of atoms that occur in $\forall(F\delta)$. By assuming that every atom in $\forall(F\delta)$ instantiates an evaluation signature in $\mathcal{T}$ then, by Corollary 2, $\forall(rew(F)\delta)$ has evaluation in $\mathcal{T}$.

Corollary 5 *(Folding preserves semantics). Let $fold(F)$ be the folding of a formula F. Then,*

$$\mathcal{T} \cup S(r_i), ..., \cup S(r_{i+k}) \models \forall(F \Leftrightarrow fold(F))$$

with $S(r_i), ..., S(r_{i+k})$ as the auxiliary specifications used in the folding step.

Proof. By definition of auxiliary specification (Definition 6), $fold(F)$ is derived from F by replacing sub-formulae (the right-hand sides of an auxiliary specification) by equivalent atoms (the left-hand sides of auxiliary specifications).

Corollary 6 *(Folding preserves evaluations). Let $fold(F)$ be the folding of a formula F. Then,*

$$\mathcal{T} \cup S(r_i), ..., \cup S(r_{i+k}) \models \forall(F \Leftrightarrow fold(F))$$

with $S(r_i), ..., S(r_{i+k})$ as the auxiliary specifications used in the folding step.

Proof. By definition (Definition 6), every auxiliary specification preserves the construction of evaluations.

Corollary 7 *(Unfold/fold transformation step preserves semantics). Let $S(r_i)$ be an auxiliary specification in the language of a theory $\mathcal{T}$ and $S^*(r_i)$ the synthesized specification resulting from the unfold/fold transformation step of $S(r_i)$. Then,*

$$\mathcal{T} \cup S(r_i) \models r_i\theta \text{ iff } \mathcal{T} \cup S^*(r_i) \cup S(r_{i+1}), ..., \cup S(r_{i+k}) \models r_i\theta$$
$$\mathcal{T} \cup S(r_i) \models \neg r_i\theta \text{ iff } \mathcal{T} \cup S^*(r_i) \cup S(r_{i+1}), ..., \cup S(r_{i+k}) \models \neg r_i\theta$$

with $r_i\theta$ as a ground atom and $S(r_{i+1}), ..., S(r_{i+k})$ as the new auxiliary specification generated in the transformation step.

Proof. The unfold/fold transformation step is internally structured as a sequential composition of meaning-preserving steps (see Corollary 3, Lemma 3, Lemma 5, Corollary 5).

Corollary 8 *(Unfold/fold transformation preserves evaluations). Let $S(r_i)$ be an auxiliary specification in the language of a theory $\mathcal{T}$ and $S^*(r_i)$ the synthesized specification resulting from the unfold/fold transformation step of $S(r_i)$. Then,*

$$\mathcal{T} \cup S(r_i) \vdash r_i\theta \text{ iff } \mathcal{T} \cup S^*(r_i) \cup S(r_{i+1}), ..., \cup S(r_{i+k}) \vdash r_i\theta$$
$$\mathcal{T} \cup S(r_i) \vdash \neg r_i\theta \text{ iff } \mathcal{T} \cup S^*(r_i) \cup S(r_{i+1}), ..., \cup S(r_{i+k}) \vdash \neg r_i\theta$$

with $r_i\theta$ as a ground atom and $S(r_{i+1}), ..., S(r_{i+k})$ as the new auxiliary specification generated in the transformation step.

Proof. The unfold/fold transformation step is internally structured as a sequential composition of steps which preserve evaluations (see Corollary 4, Lemma 4, Lemma 6 and Corollary 6).

Theorem 3 *(The synthesis method is terminating). The amount of iterations in the synthesis process of an assertion definition is finite.*

Proof. By construction, the unfolding step generates formulae which preserve the structure of logical connectives in the right-hand side of the assertion definition and generates atoms all having form of non-lower f-atoms. By definition, the set of f-atoms in any theory is finite. Therefore, the set of f-formulae we can generate by unfolding is always finite. The rewriting step only generates two classes of formulae (denoted by F_i and H_i in Sect. 4). By definition, F_i preserves the structure of logical connectives and contains a subset of atoms generated from the unfolding step. Therefore, the set of f-formulae derived from F_i is also finite. By definition, H_i is a conjunction of literals containing atoms previously generated in the unfolding step. Therefore, the set of f-atoms corresponding to such atoms is also finite. Patterns are inferred from formulae F_i and H_i. Therefore, the set of inferred patterns can not be infinite. Considering that in each iteration we can transform an auxiliary specification derived from the corresponding pattern, the amount of iterations in the transformation process is necessarily finite.

Theorem 4 *(The synthesis method preserves semantics). Let $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X, Y))$ be an assertion definition written in the language of a theory $\mathcal{T}$ and $P(r)$ the positive logic program resulting from the synthesis of $D(r)$. Then,*

(1) *$Q = \forall$ in $D(r)$:*

$$\mathcal{T} \cup D(r) \models \neg r(X)\theta \quad \text{if} \quad P(r) \models r(X)\theta$$
$$\mathcal{T} \cup D(r) \models r(X)\theta \quad \text{if} \quad P(r) \not\models r(X)\theta$$

(2) *$Q = \exists$ in $D(r)$:*

$$\mathcal{T} \cup D(r) \models r(X)\theta \quad \text{if} \quad P(r) \models r(X)\theta$$
$$\mathcal{T} \cup D(r) \models \neg r(X)\theta \quad \text{if} \quad P(r) \not\models r(X)\theta$$

being θ a ground substitution for X.

Proof. (1)

(1) By Corollary 7, each iteration (or equivalently, each unfold/fold transformation step) in the synthesis process of $S(r_1)$ preserves semantics.

(2) By Theorem 3, the synthesis process of $S(r_1)$ is terminating reaching a positive logic program $P(r_1)$.

(3) From (1) and (2), we can conclude that $\mathcal{T} \cup S(r_1) \models \exists Y r_1(X, Y)\theta$ if $P(r_1) \models \exists Y r_1(X, Y)\theta$ and $\mathcal{T} \cup S(r_1) \models \neg\exists Y r_1(X, Y)\theta$ if $P(r_1) \not\models \exists Y r_1(X, Y)\theta$.

(4) From (3) and by definition of r_1, $\mathcal{T} \cup S(r_1) \models \exists Y \neg R(X,Y)\theta$ if $P(r_1) \models \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \models \neg\exists Y \neg R(X,Y)\theta$ if $P(r_1) \not\models \exists Y r_1(X,Y)\theta$.

(5) From (4), $\mathcal{T} \cup S(r_1) \models \neg\forall Y R(X,Y)\theta$ if $P(r_1) \models \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \models \forall Y R(X,Y)\theta$ if $P(r_1) \not\models \exists Y r_1(X,Y)\theta$.

(6) From (5) and by definition of r, $\mathcal{T} \cup D(r) \models \neg r(X)\theta$ if $P(r_1) \models \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup D(r) \models r(X)\theta$ if $P(r_1) \not\models \exists Y r_1(X,Y)\theta$.

(7) From (6) and by definition of r in $P(r)$ (see Sect. 5), $\mathcal{T} \cup D(r) \models \neg r(X)\theta$ if $P(r) \models r(X)\theta$ and $\mathcal{T} \cup D(r) \models r(X)\theta$ if $P(r) \not\models r(X)\theta$.

Proof. (2)

(1) By Corollary 7, each iteration (or equivalently, each unfold/fold transformation step) in the synthesis process of $S(r_1)$ preserves semantics.

(2) By Theorem 3, the synthesis process of $S(r_1)$ is terminating reaching a positive logic program $P(r_1)$.

(3) From (1) and (2), we can conclude that $\mathcal{T} \cup S(r_1) \models \exists Y r_1(X,Y)\theta$ if $P(r_1) \models \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \models \neg\exists Y r_1(X,Y)\theta$ if $P(r_1) \not\models \exists Y r_1(X,Y)\theta$.

(4) From (3) and by definition of r_1, $\mathcal{T} \cup S(r_1) \models \exists Y R(X,Y)\theta$ if $P(r_1) \models \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \models \neg\exists Y R(X,Y)\theta$ if $P(r_1) \not\models \exists Y r_1(X,Y)\theta$.

(5) From (4) and by definition of r, $\mathcal{T} \cup D(r) \models r(X)\theta$ if $P(r_1) \models \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup D(r) \models \neg r(X)\theta$ if $P(r_1) \not\models \exists Y r_1(X,Y)\theta$.

(6) From (5) and by definition of r in $P(r)$ (see Sect. 5), $\mathcal{T} \cup D(r) \models r(X)\theta$ if $P(r) \models r(X)\theta$ and $\mathcal{T} \cup D(r) \models \neg r(X)\theta$ if $P(r) \not\models r(X)\theta$.

Theorem 5 *(Synthesis preserves evaluations). Let $D(r) = \forall X(r(X) \Leftrightarrow Q\,Y\,R(X,Y))$ be an assertion definition written in the language of a theory $\mathcal{T}$ and $P(r)$ the positive logic program resulting from the synthesis of $D(r)$. Then,*

(1) *$Q = \forall$ in $D(r)$:*

$\mathcal{T} \cup D(r) \vdash r(X)\theta$ *if* $P(r) \cup \{\Leftarrow r(X)\theta\}$ *has no SLD-refutation.*
$\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ *if* $P(r) \cup \{\Leftarrow r(X)\theta\}$ *has an SLD-refutation.*

(2) *$Q = \exists$ in $D(r)$:*

$\mathcal{T} \cup D(r) \vdash r(X)\theta$ *if* $P(r) \cup \{\Leftarrow r(X)\theta\}$ *has an SLD-refutation.*
$\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ *if* $P(r) \cup \{\Leftarrow r(X)\theta\}$ *has no SLD-refutation.*

being θ a ground substitution for X.

Proof. (1)

(1) By Corollary 8, every unfold/fold transformation step in the synthesis of $S(r_1)$ preserves the construction of evaluations for formulae of the form $\exists Y r_1(X,Y)\theta$. Therefore, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ if and only if $\mathcal{T} \cup S^*(r_1) \cup ... \cup S^*(r_n) \vdash \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ if and only if $\mathcal{T} \cup S^*(r_1) \cup ... \cup S^*(r_n) \vdash \neg\exists Y r_1(X,Y)\theta$, being $S^*(r_1)$, ..., $S^*(r_n)$ the set of synthesized specifications.

(2) From (1) and by construction of $P(r_1)$, it is not difficult to realize that $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ if there is some success branch in the SLD-tree for $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ (that is, $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has an SLD-refutation [37]) and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ if there is not any success branch in the SLD-tree for $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ (that is, $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has no SLD-refutation [37]). No infinite SLD-derivations are possible for $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$.

(3) By definition of r_1, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup S(r_1) \vdash \exists Y \neg R(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y \neg R(X,Y)\theta$.

(4) From (3) and by equivalence, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup S(r_1) \vdash \neg\forall Y R(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup S(r_1) \vdash \forall Y R(X,Y)\theta$.

(5) From (4) and by definition of r, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup D(r) \vdash r(X)\theta$.

(6) From (2) and (5), $\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ if $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has an SLD-refutation and $\mathcal{T} \cup D(r) \vdash r(X)\theta$ if $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has no SLD-refutation.

(7) From (6) and by definition of r in $P(r)$ (see Sect. 5), $\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ if $P(r) \cup \{\Leftarrow r(X)\theta\}$ has an SLD-refutation and $\mathcal{T} \cup D(r) \vdash r(X)\theta$ if $P(r) \cup \{\Leftarrow r(X)\theta\}$ has no SLD-refutation.

Proof. (2)

(1) By Corollary 8, every unfold/fold transformation step in the synthesis of $S(r_1)$ preserves the construction of evaluations for formulae of the form $\exists Y r_1(X,Y)\theta$. Therefore, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ if and only if $\mathcal{T} \cup S^*(r_1) \cup ... \cup S^*(r_n) \vdash \exists Y r_1(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ if and only if $\mathcal{T} \cup S^*(r_1) \cup ... \cup S^*(r_n) \vdash \neg\exists Y r_1(X,Y)\theta$, being $S^*(r_1)$, ..., $S^*(r_n)$ the set of synthesized specifications.

(2) From (1) and by construction of $P(r_1)$, it is not difficult to realize that $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ if there is some success branch in the SLD-tree for $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ (that is, $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has an SLD-refutation [37])

and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ if there is not any success branch in the SLD-tree for $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ (that is, $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has no SLD-refutation [37]). No infinite SLD-derivations are possible for $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$.

(3) By definition of r_1, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup S(r_1) \vdash \exists Y R(X,Y)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y R(X,Y)\theta$.

(4) From (3) and by definition of r, $\mathcal{T} \cup S(r_1) \vdash \exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup D(r) \vdash r(X)\theta$ and $\mathcal{T} \cup S(r_1) \vdash \neg\exists Y r_1(X,Y)\theta$ iff $\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$.

(5) From (2) and (4), $\mathcal{T} \cup D(r) \vdash r(X)\theta$ if $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has an SLD-refutation and $\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ if $P(r_1) \cup \{\Leftarrow \exists Y r_1(X,Y)\theta\}$ has no SLD-refutation.

(6) From (7) and by definition of r in $P(r)$ (see Sect. 5), $\mathcal{T} \cup D(r) \vdash r(X)\theta$ if $P(r) \cup \{\Leftarrow r(X)\theta\}$ has an SLD-refutation and $\mathcal{T} \cup D(r) \vdash \neg r(X)\theta$ if $P(r) \cup \{\Leftarrow r(X)\theta\}$ has no SLD-refutation.

Example 10 *(Predicate specifications).* This example shows the set of predicate specifications used in the writing of the assertion definitions in Example 4.

Predicate signature: $append : Seq \times Seq \times Seq$
Evaluation signature: 1. $append : Seq \times Seq \times Seq\downarrow$
Axioms:
$append(empty, empty, empty) \Leftrightarrow true$
$\forall(append(empty, seq(E,L), empty) \Leftrightarrow false)$
$\forall(append(seq(X,Y), empty, empty) \Leftrightarrow false)$
$\forall(append(seq(X,Y), seq(E,L), empty) \Leftrightarrow false)$
$\forall(append(empty, empty, seq(V,W)) \Leftrightarrow false)$
$\forall(append(empty, seq(E,L), seq(V,W)) \Leftrightarrow id(E,V) \wedge idSeq(L,W))$
$\forall(append(seq(X,Y), emptyseq, seq(V,W)) \Leftrightarrow id(X,V) \wedge idSeq(Y,W))$
$\forall(append(seq(X,Y), seq(E,L), seq(V,W)) \Leftrightarrow id(X,V) \wedge append(Y, seq(E,L), W))$

Predicate signature: $prefix : Seq \times Seq$
Evaluation signature: 1. $prefix : Seq \times Seq\downarrow$ 2. $prefix : Seq\downarrow \times Seq$
Axioms:
$prefix(empty, empty) \Leftrightarrow true$
$\forall(prefix(empty, seq(E,L)) \Leftrightarrow true)$
$\forall(prefix(seq(E,L), empty) \Leftrightarrow false)$
$\forall(prefix(seq(N,M), seq(E,L)) \Leftrightarrow id(N,E) \wedge prefix(M,L))$

Predicate signature: $size : Seq \times Nat$
Evaluation signature: 1. $size : Seq \times Nat\downarrow$
Axioms:
$size(empty, zero) \Leftrightarrow true$
$\forall(size(empty, succ(N)) \Leftrightarrow false)$
$\forall(size(seq(X,L), zero) \Leftrightarrow false)$
$\forall(size(seq(X,L), succ(N)) \Leftrightarrow size(L,N))$

Predicate signature: $oprefix : Seq \times Seq$
Evaluation signature: 1. $oprefix : Seq \times Seq\downarrow$ 2. $oprefix : Seq\downarrow \times Seq$
Axioms:
$oprefix(empty, empty) \Leftrightarrow true$
$\forall(oprefix(empty, seq(F, empty)) \Leftrightarrow true)$
$\forall(oprefix(empty, seq(F, seq(V,W))) \Leftrightarrow true)$
$\forall(oprefix(seq(E, empty), empty) \Leftrightarrow false)$
$\forall(oprefix(seq(E, seq(X,Y)), empty) \Leftrightarrow false)$
$\forall(oprefix(seq(E, emptyseq), seq(F, emptyseq)) \Leftrightarrow id(E,F))$
$\forall(oprefix(seq(E, emptyseq), seq(F, seq(V,W))) \Leftrightarrow id(E,F))$
$\forall(oprefix(seq(E, seq(X,Y)), seq(F, empty)) \Leftrightarrow false)$
$\forall(oprefix(seq(E, seq(X,Y)), seq(F, seq(V,W))) \Leftrightarrow$
$id(E,F) \wedge le(F,V) \wedge oprefix(seq(X,Y), seq(V,W)))$

Predicate signature: $lts : Seq \times Nat$
Evaluation signature: 1. $lts : Seq\downarrow \times Nat$
Axioms:
$\forall(lts(empty, N) \Leftrightarrow true)$
$\forall(lts(seq(X,Y), N) \Leftrightarrow lt(X,N) \wedge lts(Y,N))$

Predicate signature: $ges : Seq \times Nat$
Evaluation signature: 1. $ges : Seq{\downarrow} \times Nat$
Axioms:
$\forall(ges(empty, N) \Leftrightarrow true)$
$\forall(ges(seq(X, Y), N) \Leftrightarrow ge(X, N) \wedge ges(Y, N))$

Predicate signature: $eqs : Seq$
Evaluation signature: 1. $eqs : Seq{\downarrow}$
Axioms:
$eqs(empty) \Leftrightarrow true$
$\forall(eqs(seq(X, empty)) \Leftrightarrow true)$
$\forall(eqs(seq(X, seq(V, W))) \Leftrightarrow id(X, V) \wedge eqs(seq(V, W)))$

Predicate signature: $le : Nat \times Nat$
Evaluation signature: 1. $le : Nat{\downarrow} \times Nat$ 2. $le : Nat \times Nat{\downarrow}$
Axioms:
$le(zero, zero) \Leftrightarrow true$
$\forall(le(zero, succ(Y)) \Leftrightarrow true)$
$\forall(le(succ(X), zero) \Leftrightarrow true)$
$\forall(le(succ(X), succ(Y)) \Leftrightarrow le(X, Y))$

Predicate signature: $lt : Nat \times Nat$
Evaluation signature: 1. $lt : Nat{\downarrow} \times Nat$ 2. $lt : Nat \times Nat{\downarrow}$
Axioms:
$lt(zero, zero) \Leftrightarrow false$
$\forall(lt(zero, succ(Y)) \Leftrightarrow true)$
$\forall(lt(succ(X), zero) \Leftrightarrow false)$
$\forall(lt(succ(X), succ(Y)) \Leftrightarrow lt(X, Y))$

Predicate signature: $ge : Nat \times Nat$
Evaluation signature: 1. $ge : Nat{\downarrow} \times Nat$ 2. $ge : Nat \times Nat{\downarrow}$
Axioms:
$ge(zero, zero) \Leftrightarrow true$
$\forall(ge(zero, succ(Y)) \Leftrightarrow false)$
$\forall(ge(succ(X), zero) \Leftrightarrow true)$
$\forall(ge(succ(X), succ(Y)) \Leftrightarrow ge(X, Y))$

Predicate signature: $idSeq : Seq \times Seq$
Evaluation signature: 1. $idSeq : Seq{\downarrow} \times Seq$ 2. $idSeq : Seq \times Seq{\downarrow}$
Axioms:
$idSeq(emptyseq, emptyseq) \Leftrightarrow true$
$\forall(idSeq(emptyseq, seq(V, W)) \Leftrightarrow false)$
$\forall(idSeq(seq(X, Y), emptyseq) \Leftrightarrow false)$
$\forall(idSeq(seq(X, Y), seq(V, W)) \Leftrightarrow id(X, V) \wedge idSeq(Y, W))$

Predicate signature: $elem : Nat \times Seq \times Nat$
Evaluation signature: 1. $elem : Nat{\downarrow} \times Seq{\downarrow} \times Nat$
Axioms:
$\forall(elem(zero, emptyseq, E) \Leftrightarrow false)$
$\forall(elem(succ(N), emptyseq, E) \Leftrightarrow false)$
$\forall(elem(zero, seq(X, Y), E) \Leftrightarrow id(X, E))$
$\forall(elem(succ(N), seq(X, Y), E) \Leftrightarrow elem(N, Y, E))$

Example 11 *(Synthesized logic program $P(plateau)$).* This example shows the positive logic program generated by our synthesizer for checking the assertion definition $D(plateau)$ (see Example 4).

$\forall(plateau(S, N) \Leftarrow plateau_1(S, L, N)))$
$\forall(plateau_1(seq(X, Y), empty, N) \Leftarrow plateau_2(X, Y, N))$
$\forall(plateau_1(seq(X, Y), seq(V, W), N) \Leftarrow plateau_3(Y, W, X, N), plateau_4(X, V))$
$\forall(plateau_1(seq(X, Y), seq(V, W), N) \Leftarrow plateau_2(X, Y, N), plateau_5(X, V))$

$\forall(plateau_1(empty, seq(V, W), N) \Leftarrow plateau_6(N))$
$\forall(plateau_1(empty, empty, N) \Leftarrow plateau_6(N))$
$\forall(plateau_2(N, Y, succ(M)) \Leftarrow plateau_9(N, Y, M))$
$\forall(plateau_2(N, Y, zero) \Leftarrow plateau_{10}(N, Y))$
$\forall(plateau_3(Y, W, N, succ(M)) \Leftarrow plateau_{13}(Y, W, N, M))$
$\forall(plateau_3(Y, W, N, zero) \Leftarrow plateau_{14}(Y, W, N))$
$plateau_4(zero, zero) \Leftarrow plateau_{16}$
$\forall(plateau_4(succ(N), succ(M)) \Leftarrow plateau_4(N, M))$
$\forall(plateau_5(succ(X), zero) \Leftarrow plateau_{16})$
$\forall(plateau_5(zero, succ(M)) \Leftarrow plateau_{16})$
$\forall(plateau_5(succ(N), succ(M)) \Leftarrow plateau_5(N, M))$
$plateau_6(zero) \Leftarrow plateau_7$
$plateau_7 \Leftarrow true$
$\forall(plateau_9(N, seq(V, W), M) \Leftarrow plateau_2(V, W, M), plateau_4(N, V))$
$\forall(plateau_9(N, seq(V, W), M) \Leftarrow plateau_{12}(V, W, M), plateau_5(N, V))$
$\forall(plateau_{10}(N, seq(V, W)) \Leftarrow plateau_{10}(V, W), plateau_4(N, V))$
$\forall(plateau_{12}(X, Y, succ(M)) \Leftarrow plateau_{21}(Y, M))$
$\forall(plateau_{13}(seq(X, Y), empty, N, M) \Leftarrow plateau_2(X, Y, M), plateau_4(N, X))$
$\forall(plateau_{13}(seq(X, Y), seq(V, W), N, M) \Leftarrow plateau_{19}(Y, W, X, M), plateau_4(X, V), plateau_5(N, X))$
$\forall(plateau_{13}(seq(X, Y), seq(V, W), N, M) \Leftarrow plateau_2(X, Y, M), plateau_5(X, V), plateau_4(N, X))$
$\forall(plateau_{13}(seq(X, Y), empty, N, M) \Leftarrow plateau_{12}(X, Y, M), plateau_5(N, X))$
$\forall(plateau_{13}(seq(X, Y), seq(V, W), N, M) \Leftarrow plateau_3(Y, W, X, M), plateau_4(X, V), plateau_4(N, X))$
$\forall(plateau_{13}(empty, empty, N, M) \Leftarrow plateau_6(M))$
$\forall(plateau_{13}(seq(X, Y), seq(V, W), N, M) \Leftarrow plateau_{12}(X, Y, M), plateau_5(X, V), plateau_5(N, X))$
$\forall(plateau_{13}(empty, seq(X, Y), N, M) \Leftarrow plateau_6(M))$
$\forall(plateau_{14}(seq(X, Y), empty, N) \Leftarrow plateau_{10}(X, Y), plateau_4(N, X))$
$\forall(plateau_{14}(seq(X, Y), seq(V, W), N) \Leftarrow plateau_{14}(Y, W, X), plateau_4(X, V), plateau_4(N, X))$
$\forall(plateau_{14}(seq(X, Y), seq(V, W), N) \Leftarrow plateau_{10}(X, Y), plateau_5(X, V), plateau_4(N, X))$
$\forall(plateau_{14}(seq(X, Y), seq(V, W), N) \Leftarrow plateau_{18}(Y, W), plateau_4(X, V), plateau_5(N, X))$
$plateau_{16} \Leftarrow true$
$\forall(plateau_{18}(seq(X, Y), seq(V, W)) \Leftarrow plateau_{18}(Y, W), plateau_4(X, V))$
$\forall(plateau_{19}(Y, W, N, succ(M)) \Leftarrow plateau_{25}(Y, W, M))$
$\forall(plateau_{19}(Y, W, N, zero) \Leftarrow plateau_{18}(Y, W))$
$\forall(plateau_{21}(seq(X, Y), succ(N)) \Leftarrow plateau_{21}(Y, N))$
$\forall(plateau_{25}(seq(X, Y), seq(V, W), N) \Leftarrow plateau_{12}(X, Y, N), plateau_5(X, V))$
$\forall(plateau_{25}(seq(X, Y), seq(V, W), N) \Leftarrow plateau_{19}(Y, W, X, N), plateau_4(X, V))$
$\forall(plateau_{25}(seq(X, Y), empty, N) \Leftarrow plateau_{12}(X, Y, N))$

References

[1] K.R. Apt, R.R. Bol, Logic programming and negation: a survey, J. Log. Program. 19–20 (1994) 9–71.
[2] SunTest, The Open Group Research Institute, ADL 2.0 for Java Language Reference Manual, v1.2, 1998.
[3] D. Basin, Y. Deville, P. Flener, A. Hamfelt, J. Nilsson, Synthesis of programs in computational logic, in: Program Development in Computational Logic, in: Lect. Notes Comput. Sci., vol. 3049, Springer, Berlin, 2004, pp. 30–65.
[4] N. Bjorner, Taking satisfiability to the next level with Z3 (Abstract), in: IJCAR, 2012, pp. 1–8.
[5] M. Barnett, K. Rustan M. Leino, W. Schulte, The Spec# programming system, an overview, Manuscript KRLM 136, Microsoft Research, 2004.
[6] D. Basin, S. Matthews, Adding metatheoretic facilities to first-order theories, J. Log. Comput. 6 (6) (1996) 835–849.
[7] R. Barbuti, P. Mancarella, D. Pedreschi, F. Turini, A transformational approach to negation in logic programming, J. Log. Program. 8 (1990) 201–228.
[8] N. Bjorner, K.L. McMillan, A. Rybalchenko, On solving universally quantified Horn clauses, in: SAS, 2013, pp. 105–125.
[9] A. Bundy, A. Smaill, G. Wiggins, The synthesis of logic programs from inductive proofs, in: Computational Logic, Symposium Proceedings, Springer-Verlag, 1990, pp. 135–149.
[10] M. Calejo, InterProlog: towards a declarative embedding of logic programming in Java, in: 9th European Conference on Logic in Artificial Intelligence, Lisbon, 2004.
[11] Negation as failure, in: Logic and Databases, Plenum Press, New York, 1978, pp. 293–322.
[12] T. Coquant, G. Huet, The calculus of constructions, Inf. Comput. 76 (2/3) (1988) 95–120.

[13] G. Dayantis, Logic program derivation for a class of first-order relations, in: IJCAI, 1987.
[14] Y. Deville, K.K. Lau, Logic program synthesis, J. Log. Program. (12) (1993) 1–32.
[15] D. Detlefs, G. Nelson, J.B. Saxe, Simplify: a theorem prover for program checking, J. ACM 52 (3) (2005) 365–473.
[16] Y. Deville, Logic Programming. Systematic Program Development, Addison–Wesley, 1990.
[17] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.
[18] G. Dromey, Program Derivation. The Development of Programs from Specifications, Addison–Wesley, 1989.
[19] P. Flener, Logic Program Synthesis from Incomplete Information, Kluwer Academics Publishers, 1995.
[20] P. Flener, Achievements and prospects of program synthesis, in: Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski, Part I, in: Lect. Notes Comput. Sci., vol. 2407, Springer, Berlin, 2002, pp. 1–43.
[21] P. Flener, K.K. Lau, M. Ornaghi, Correct schema-guided synthesis of steadfast programs, in: Proc. of ASE'97, IEEE Computer Society Press, 1997, pp. 153–160.
[22] L. Fribourg, Extracting logic programs from proofs that use extended Prolog execution and induction, in: ICLP90, The MIT Press, 1990, pp. 685–699.
[23] F.J. Galán, J.M. Cañete, A method for compiling and executing expressive assertions, in: 4th International Conference Integrated Formal Methods, in: Lect. Notes Comput. Sci., vol. 2999, Springer, Berlin, 2004, pp. 521–540.
[24] Y. Ge, C. Barrett, C. Tinelli, Solving quantified verification conditions using satisfiability modulo theories, in: Proceedings of the 21st International Conference on Automated Deduction, CADE '07, in: Lect. Notes Artif. Intell., vol. 4603, Springer-Verlag, Bremen, Germany, July 2007, pp. 167–182.
[25] S. Gulwani, B. McCloskey, A. Tiwari, Lifting abstract interpreters to quantified logical domains, in: POPL, 2008, pp. 235–246.
[26] D. Gries, The Science of Programming, Springer-Verlag, 1981.
[27] S. Gulwani, Dimensions in program synthesis, in: PPDP, 2010, pp. 13–24.
[28] C.J. Hogger, Derivation of logic programs, J. ACM 28 (1981) 372–392.
[29] The Jass Page, http://csd.informatik.uni-oldenburg.de/~jass/index.html.
[30] Man-Machine-Systems, design by contract for Java using JMSAssert, http://www.mmsindia.com/DBCForJava.html, 2009.
[31] Java technology, http://java.sun.com.
[32] T. Kanamori, H. Fujita, Unfold/fold logic program transformation with counters, ICOT Tech. Report TR-179, 1986.
[33] R. Kramer, iContract: the Java(tm) design by contract(tm) tool, in: Proceedings of the Technology of Object-Oriented Languages and Systems, 1998.
[34] K. Kunen, Signed data dependencies in logic programs, J. Log. Program. 7 (1989) 231–245.
[35] K. Lau, M. Ornaghi, On specification frameworks and deductive synthesis of logic programs, in: Lect. Notes Comput. Sci., vol. 883, Springer-Verlag, 1994, pp. 104–121.
[36] G. Leavens, A. Baker, C. Ruby, Preliminary design of JML: a behavioral interface specification language for Java, ACM SIGSOFT Softw. Eng. Notes 31 (3) (March 2006) 1–38.
[37] J.W. Lloyd, Foundations of Logic Programming, 2nd ed., Springer-Verlag, 1987.
[38] M.J. Maher, A transformation system for deductive database modules with perfect models semantics, in: Proc. 9th Conf. on Foundations of Software Technology and Theoretical Computer Science, 1989, pp. 89–98.
[39] Z. Manna, Mathematical Theory of Computation, McGraw-Hill Comput. Sci. Ser., McGraw–Hill, 1974.
[40] B. Meyer, Eiffel: The Language, Prentice Hall, 1992.
[41] L. de Moura, N. Bjørner, Satisfiability modulo theories: introduction and applications, Commun. ACM 54 (9) (2011) 69–77.
[42] Z. Manna, R. Waldinger, The Logical Basis for Computer Programming. Volume 1: Deductive Reasoning, Addison–Wesley, 1985.
[43] Z. Manna, R. Waldinger, The Deductive Foundations of Computer Programming, Addison–Wesley, 1993.
[44] A. Nonnengart, C. Weidenbach, Computing small clause normal forms, in: Handbook of Automated Reasoning, Elsevier Science Publishers B.V., 1999.
[45] H.A. Partsch, Specification and Transformation of Programs. A Formal Approach to Software Development, Springer-Verlag, 1990.
[46] Terence Parr, ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/.
[47] A. Pettorossi, M. Proietti, Synthesis and transformation of logic programs using unfold/fold proof, J. Log. Program. 41 (1999) 197–230.
[48] A. Pettorossi, M. Proietti, Perfect model checking via unfold/fold transformations, Technical report R.513 IASI-CNR, Rome, September, 2000.
[49] A. Pettorossi, M. Proietti, Program derivation = rules + strategies, in: Computational Logic, in: Lect. Notes Artif. Intell., vol. 2047, 2002, pp. 273–309.
[50] R. Plosch, Evaluation of assertion support for the Java programming language, J. Object Technol. 1 (2002).
[51] M. Proietti, A. Pettorossi, Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs, Theor. Comput. Sci. 142 (1995) 89–124.
[52] T. Sato, Completed logic programs and their consistency, J. Log. Program. 9 (1990) 33–44.
[53] T. Sato, Equivalence-preserving first-order unfold/fold transformation systems, Theor. Comput. Sci. 105 (1992) 57–84.
[54] T. Sato, H. Tamaki, First-order compiler: a deterministic logic program synthesis algorithm, J. Symb. Comput. 8 (1989) 605–627.
[55] H. Seki, Unfold/fold transformation of general logic programs for the well-founded semantics, J. Log. Program. (16) (1993) 5–23.
[56] S. Srivastava, S. Gulwani, Program verification using templates over predicate abstraction, in: PLDI, 2009, pp. 223–234.
[57] S. Srivastava, S. Gulwani, S. Chaudhuri, J.S. Foster, Path-based inductive synthesis for program inversion, in: PLDI, 2011, pp. 492–503.
[58] S. Srivastava, S. Gulwani, J.S. Foster, From program verification to program synthesis, in: POPL, 2010, pp. 313–326.
[59] S. Srivastava, S. Gulwani, J.S. Foster, Template-based program verification and program synthesis, Int. J. Softw. Tools Technol. Transf. 15 (2013) 497–518.
[60] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V.A. Saraswat, S.A. Seshia, Sketching stencils, in: PLDI, 2007, pp. 167–178.
[61] Sun Microsystems, Programming with Assertions, 2002.
[62] SWI Prolog, http://www.swi-prolog.org/.
[63] H. Tamaki, T. Sato, Unfold/fold transformation of logic programs, in: Proc. 2nd International Logic Program Conference, 1984, pp. 127–137.