Constructing Recursions by Similarity

F. J. Galán, J. M. Cañete and V. J. Diaz Madrigal

Dept. of Languages and Computer Systems.
Faculty of Computer Science of Seville
Dept. de Lenguajes y Sistemas Informáticos.
Av. Reina Mercedes s/n. 41012.Sevilla. fax:34 954557139
e-mail:galanm@lsi.us.es

Abstract

A formal specification can describe software models which are difficult to program. Transformational methods based on fold/unfold strategies have been proposed to palliate this problem. The objective of applying transformations is to filter out a new version of the specification where recursion may be introduced by a folding step. Among many problems, the "eureka" about when and how to define a new predicate is difficult to find automatically. We propose a new version of the folding rule which decides automatically how to introduce new predicates in a specification. Our method is based on finding similarities between formulas represented as parsing trees and it constitutes an assistance to the complex problem of deriving recursive specifications from non recursive ones.

Keywords: *specification transformation, program synthesis, correctness preservation, program specification.*

1 Introduction

Usually, a specification describes software models which are difficult to program. Systematic construction of programs from specifications is known as program synthesis. A huge variety of synthesis mechanisms have been developed [4], [5], [6],[7], [8], [16], [2]. In this work, we are interested in transformational mechanisms; a sequence of meaning-preserving transformation rules (e.g. unfolding, folding, universal instantiation, abstraction, predicate definition, etc.) is applied to a specification until a program is obtained. The objective of applying transformations is to filter out a new version of the specification where recursion may be introduced by a folding step. However, among many others problems, deciding about when and how to define a new predicate (i.e. recursive predicate) is difficult to find automatically. Fold/unfold transformations represent an important investigation subject in the literature [1], [5], [13], [15], [14], [18]. Basically, unfolding represents the replacement of an atom by its definition and folding represents the inverse operation of replacing a subformula by an atom. In the following example, the context $\mathcal{S}$ defines a (many-sorted) first-order language with types *Nat* (natural numbers) constructed from the function symbols 0 and *s* and *Seq*(*Nat*) (sequences of natural numbers) constructed from the function symbols *empty* and *conc*. It defines also the meaning of relation symbols such as = (identity between natural numbers), *nocc* (number of occurrences of an element in a sequence) and *perm* (permutations of a sequence of natural numbers).

$\mathcal{S}$ = {Types : *Nat* generated by 0; s *Seq (Nat)* generated by *empty; conc*

$D_{=}$: $0 = 0 \Leftrightarrow true$ $0 = s(x) \Leftrightarrow false$
$s(x) = 0 \Leftrightarrow false$ $s(x) = s(y) \Leftrightarrow x = y$

D_{nocc}: $nocc(e, empty, z) \Leftrightarrow z = 0$
$nocc(e, conc(x, Y), s(z)) \Leftrightarrow x = e \wedge nocc(e, Y, z)$
$nocc(e, conc(x, Y), z) \Leftrightarrow \neg x = e \wedge nocc(e, Y, z)$

D_{perm}: $perm(L, S) \Leftrightarrow (nocc(a, L, z) \Leftrightarrow nocc(a, S, z))\}$

Dperm is not closed to the structure of a program (i.e. there is not any explicit recursion). Following a transformational synthesis process for *Dperm* (e.g. applying universal instantiation on variables *L*, *S* and *z* in *Dperm*, we obtain *D*1 and then unfolding *D*1 w.r.t. *nocc*(*b; conc*(*v; V*); *s*(*k*)) and *nocc*(*b; conc*(*w;W*); *s*(*k*))) atoms using second axiom in *Dnocc*) we reach expressions such as *D*2:

D1 : $perm$ (*conc* (*v, V*); *conc* (*w,W*))
⇔ (*nocc* (*b, conc* (*v, V*), *s* (*k*))
⇔
nocc (*b, conc* (*w,W*), *s* (*k*)))

D2 : $perm$ (*conc* (*v, V*), *conc* (*w,W*))
⇔ ((*v* = *b* ∧ *nocc* (*b, V, k*))
⇔
(*w* = *b* ∧ *nocc* (*b,W, k*)))

Two questions arise at this point, (a) Is it possible to introduce recursive predicates in *D*2? and (b) How can we do it? It is dificult to achieve an "automatic answer" to these questions. Our method follows a constructive approach. A comparison based on the notion of similarity between *D*2 and *Dperm* is needed to decide about first question. Only if first question is answered affirmatively then a similarity-based folding rule is applied to *D* in order to answer second question.

Our work is explained in the following manner. Section 2 defines the form of our specifications and a non-constructive characterization of the folding rule is presented. Section 3 defines the concept of similarity. Basically, it represents an automatic method for deciding which subformulas produce recursion. In section 4, we describe a similarity based folding rule which preserves correctness, and finally, in section 5 we establish conclusions.

2 Preliminary Definitions

In this section, the syntax and semantics of our specifications and a non constructive definition of the folding rule are presented. The use of the folding rule is intended to introduce recursion in a specification.

DEFINITION 2.1 (SYNTAX OF A FORMULA) *A many sorted (typed) first order language is assumed to write our formulas. A formula* $Q_{\tau_1}x_1 ... Q_{\tau_n}x_n F$ *where* $Q_{\tau_i}x_i$ *is a universal or existential quantifier defined on a type* τ_i, x_i *is different from* x_j *for* $i \neq j$, *and F contains no quantifier, is said to be in prenex normal form. We consider that, when possible, all quantifiers in a formula are ordered following a lexicographic order defined on the names of their respective types.*

For example, $\forall_{Nat} a \forall_{Nat} z \forall_{Seq(Nat)}\ L \forall_{Seq(Nat)}\ S(nocc(a, L, z) \Leftrightarrow nocc(a, S, z))$is in prenex normal form where all quantifiers have been ordered following a lexicographic order defined on the names of their respective types.

In the following, we assume that all our formulas are in prenex normal form, this does not represent any restriction due to the existence of an effective procedure for transforming any first-order formula into an equivalent one in prenex normal form [12]. For legibility reasons, we omit τ subscripts when a type can be induced clearly in a formula and expressions such as $Q_\tau x ... Q_\tau z F$ can be collapsed into equivalent expressions $Q_\tau x, ..., z F$. For example, the formula $\forall_{Nat} a \forall_{Nat} z \forall_{Seq(Nat)}\ L \forall_{Seq(Nat)} S$ $(nocc\,(a, L, z) \Leftrightarrow nocc(a, S, z))$ can be collapse into the equivalent formula $\forall_{Nat} a, z \forall_{Seq(Nat)} L, S (nocc$ $(a, L, z) \Leftrightarrow nocc(a, S, z))$. In addition, when possible, universal quantifiers are omitted in the front of a formula.

DEFINITION 2.2 (SUBSTITUTION) *A (ground) variable substitution is the pair* (*v, t*) *where v is a variable and t is a (ground) term. A substitution* σ *is a set of variable substitutions. Let* $X = \{x_1, ..., x_n\}$ *and* $Y = \{y_1, ..., y_n\}$ *be two sets of variables where* x_i *is different from* x_j *and* y_i *is different from* y_j *for* $i \neq j$. *Let* $Q_X = \{Q_{\tau_{x_1}}, ..., Q_{\tau_{x_n}}\}$ *and* $Q_Y = \{Q_{\tau_{y_1}}, ..., Q_{\tau_{y_n}}\}$ *be two sets of quantifiers for variables in X and Y respectively. We say that* $\sigma = \{(x_1, y_1), ..., (x_n, y_n)\}$ *is a renaming substitution which agrees w.r.t. quantifications iff (a) X* ∩ *Y* = ∅ *and (b)* $Q_{\tau_{x_i}} = Q_{\tau_{y_i}}$ *with* $i = 1..n$.

DEFINITION 2.3 (SPECIFICATION) *An if-and-only-if axiom is a formula of the form* $r(x_1, ..., x_n) \Leftrightarrow R(y_1, ..., y_m)$ *(e.g. axioms for =, nocc and perm relation symbols in* $\mathcal{S}$*). The symbol* r *is called the defined symbol. The atom* $r(\ , ...;\ )$ *is called the left-hand side of the axiom and the (sub)formula R*($y_1, ..., y_m$) *is called the right-hand side of the axiom. A specification for a relation symbol* r *is the set* D_r *of all axioms with the same defined symbol. In the following, we use* $D_{r,n}$ *to identify the* n^{th} *axiom in* D_r.

Definition 2.4 (Context) *A context* $\mathcal{C}$ *is a set of types and specifications for relation symbols. Types are constructed from function symbols appearing in* $\mathcal{C}$. $\mathcal{C}$ *is atomically complete if, for every ground atom* $r(t_1, ..., t_n)$, *either* $\mathcal{C} \vdash r(t_1, ..., t_n)$ *or* $\mathcal{C} \vdash \neg r$ $(t_1, ..., t_n)$. $\mathcal{C}$ *has isoinitial model M iff for every ground literal* l, $M \models l$ *iff* $\mathcal{C} \vdash l$. *Therefore, the meaning of a relation r in* $\mathcal{C}$ *is the set of all ground literals* l *defined on r such that* $\mathcal{C} \vdash l$.

Some authors have studied the problem of the existence of isoinitial models for theories in general [3] and some effective criteria have been proposed to construct consistent theories. Following [11], a context $\mathcal{C}$ admits an isoinitial model if and only if it is atomically complete. By hypothesis, we assume that our contexts are consistent in this way.

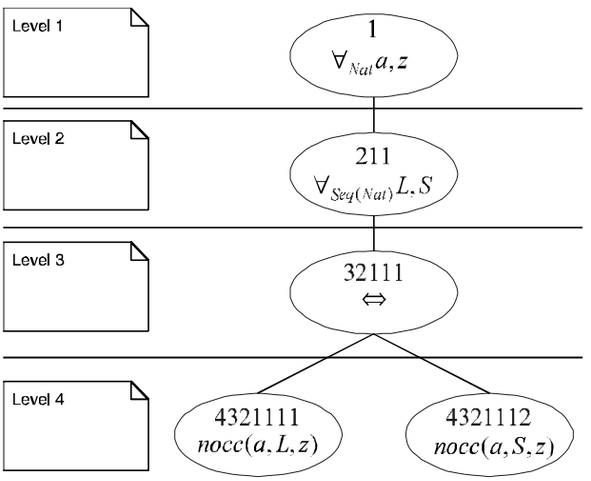

Fig. 1:

Definition 2.5 (Folding Rule) *A folding rule is a transformation rule intended for replacing a subformula by an atom. Let S be a formula and $r(x_j) \Leftrightarrow R_j(y_j)$ be an axiom for r in $\mathcal{C}$. We say that S_j is obtained from S folding with respect to $r(x_j) \Leftrightarrow R_j(y_j)$ iff $S_j = S\ |^R_{r(x_j)\theta_j}$ where (a) There exist subformula R in S and substitution θ such that $R \quad R_j(y_j)\theta_j$ and (b) $S\ |^R_{r(x_j)\theta_j}$ represents the textual replacement of R by $r(x_j)\theta_j$ in S.*

For example, the formula $S \equiv (nocc(b, V, k) \Leftrightarrow nocc(b, W, k)) \wedge v = b \wedge w = b$ can be folded with respect to D_{perm}. Applying the substitution $\theta_1 = \{(L, V), (S, W)$,-$(z, k)$,$(a, b)\}$ to the right-hand side of the axiom in D_{perm} we obtain the subformula $R \equiv nocc(b, V, k) \Leftrightarrow nocc(b, W, k)$. Finally, applying textual replacement, we obtain *perm* $(V, W) \wedge v = b \wedge w = b$.

3 Similarity

This section describes an automatic method to introduce recursion by a folding step. It is based on the notion of similarity. Basically, similarity represents a decidable relation between two parsed formulas.

DEFINITION 3.1 (PARSING TREE) *Let S be a formula in prenex normal form. We say that Parse(S) (graphical example in figure 1) is the parsing tree for S iff it is a tree representation of S where (a) each leaf node in Parse(S) represents a literal in S. (b) each non-leaf node in Parse(S) represents either a quantified set of variables $Q_\tau x, ..., z$ or a logical connective ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) in S and (c) each node in Parse(S) has unique identification by means of a number with format $lx \cdots xp$. The digit l represents the level where a node is located in Parse(S). The digit p decides if the node is located either at the left-hand side (p = 1) or at the right-hand side (p = 2) of its parent (if it exists). By default, nodes without brother nodes have p = 1. The digits $x \cdots x$ represent the identification of the parent node. The root node is an exception, it has not any parent therefore we consider a fixed identification for it equal to 1. In this way, a node identification determines univocally the position of a node in a parsing tree. We say that a preterminal node in Parse(S) is any non-leaf node in Parse(S) with at least one leaf node as child.*

Two formulas can be compared by the structure of their quantifiers and logical connectives. These measures are called similarity with respect to quantification and similarity with respect to logical connectives respectively. In the following definitions, we consider that S_1 and S_2 are two formulas in prenex normal form.

DEFINITION 3.2 (SIMILARITY FUNCTION) *We say that f is a similarity function from the node identification domain of Parse(S_1) to the node identification domain of Parse(S_2) iff each non-leaf $n_1 \in$ Parse(S_1) is mapped to a non-leaf node $n_2 = f(n_1) \in$ Parse(S_2) where quantifier/connective in n_1 coincides with quantifier/connective in n_2 and the level of n_2 is greater than or equal to the level of n_1.*

DEFINITION 3.3 (SIMILARITY W.R.T. QUANTIFICATION) *We say that S_2 is similar to S_1 w.r.t. quantification iff for each non-leaf node $n_1 \in$ Parse(S_1) containing the quantified set of variables $Q_\tau x_1, ..., x_n$ there exists a non-leaf node $n_2 = f(n_1) \in$ Parse(S_2) containing the quantified set of variables $Q_\tau y_1, ..., y_m$ such that (a) m $\geq$ n and (b) there exist two sequences of nodes, M_1 from Parse(S_1) and M_2 from Parse(S_2), with $M_1 = M_2^{-1}$ where M_1 contains n_1 and its predecessors (from bottom to up) and M_2 contains n_2 and its predecessors (from bottom to up). (M_2^{-1} is obtained by applying f^{-1}, when defined, to elements in M_2). If S_2 is similar to S_1 w.r.t. quantification then f induces a set of possible renaming substitutions for variables in S_1 (from variables in S_2) which agrees w.r.t. quantification. If $Q_\tau X$ is the set of quantified variables in n_1 and $Q_\tau Y$ is*

the set of quantified variables in $n_2 = f(n_1)$ *then f induces substitutions of the form* $\{(x_j, y_k)\}$ *with* $x_j \in X$ *and* $y_k \in Y$.

In figure 5, S_2 is similar to S_1 w.r.t. quantification:

$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n : [x_1, x_2, \ldots, x_n]$ and $Q_1 y_1 Q_2 y_2 \ldots Q_n y_n : [y_1, y_2, \ldots, y_n]$

and some examples of substitutions induced by f are:

$\{(a, b), (z, k), (L, V), (S, W)\}$
$\{(a, k), (z, v), (L, W), (S, V)\}$ ...

Definition 3.4 (Similarity w.r.t. Logical Connectives) *We say that* S_2 *is in-depth similar to* S_1 *iff for each non-leaf node* $n_1 \in$ *Parse*(S_1) *containing a logical connective there exists a non-leaf node* $n_2 = f(n_1) \in$ *Parse*(S_2) *and there exist two sequences of nodes,* M_1 *from Parse*(S_1) *and* M_2 *from Parse*(S_2), *with* $M_1 = M_2^{-1}$ *where* M_1 *contains* n_1 *and its predecessors (from bottom to up) and* M_2 *contains* n_2 *and its predecessors (from bottom to up). We say that* S_2 *is in-breadth similar to* S_1 *iff for each level* $l > 1$ *of Parse*(S_1) *with* $N_{1,l} = \{lx_1p_1, \ldots, lx_kp_k\}$ *as the set of all nodes in* l *containing logical connectives, there exists a set of nodes in Parse*(S_2), *possibly from several levels, say* $l_1, \ldots, l_j$, *of the form* $N_{2,\{l_1,\ldots,l_j\}} = \{\pi_1 f(x_1) p_1 \varsigma_1, \ldots, \pi_k f(x_k)\ p_k \varsigma_k\}$ *where* π_i *and* ς_i $(i = 1..k)$ *are (sub)sequences of numbers. If the node with identification* 1 *(level* l = 1*) of Parse*(S_1) *contains a logical connective then there exists a node identification in Parse*(S_2) *of the form* $\pi f(1) \varsigma$ *in Parse*(S_2) *where* π *and* ς *are (sub)sequences of numbers. We say that* S_2 *is similar to S*1 *w.r.t. logical connectives iff* S_2 *is in-depth similar and in-breadth similar to* S_1.

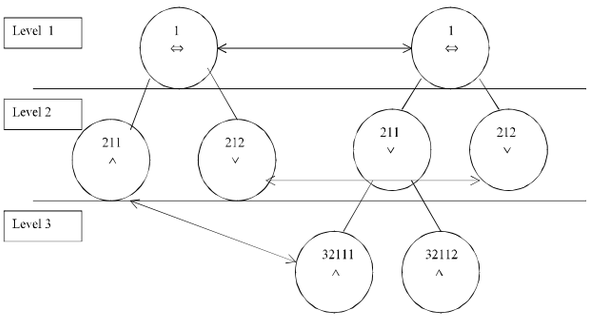

Fig. 2:

For example, in figure 2 we show an example of similarity with respect to logical connectives between the formulas S_1 and S_2 (for legibility reasons, each r_i represents a ground literal):

$$S_1: \quad (r_1 \wedge r_2) \Leftrightarrow (r_3 \vee r_4)$$
$$S_2: \quad ((r_5 \wedge r_6) \vee (r_7 \wedge r_8)) \Leftrightarrow (r_9 \vee r_{10}))$$

.

$f(1) = 1$ $\quad \mathbf{M_1} = \{\mathbf{1}\}$
$M_2 = \{1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{1}\}$

$f(211) = 32111$ $\quad \mathbf{M_1} = \{\mathbf{211}, \mathbf{1}\}$
$M_2 = \{32111, 211, 1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{211}. \mathbf{1}\}$

$f(212) = 212$ $\quad \mathbf{M_1} = \{\mathbf{212}, \mathbf{1}\}$
$M_2 = \{212, 1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{212}, \mathbf{1}\}$

In general, if *f* exists then it may not be unique. For example, the node $n_{211} \in$ *Parse*(S_1) can also be mapped to the $n_{32112} \in$ *Parse*(S_2) obtaining in this way another *f*.

In-breadth similarity (in relation to the definition 3.4, bold numbers have been used for x_jp_j in $N_{1,l}$ and for $f(x_j)p_j$ in $N_{2,\{l_1,\ldots,l_j\}}$):

$l = 2$ (level 2), $x_1 = 1$, $p_1 = 1$,
$x_2 = 1$, $p_2 = 2$, $N_{1,2} = \{2\mathbf{11}, 2\mathbf{12}\}$
$f(1) = 1$, $\pi_1 = 32$, $\varsigma_1 = 1$, π_2
$= 2$, $\varsigma_2 = \emptyset$, $N_{2,\{2,3\}} = \{32\mathbf{111}, 2\mathbf{12}\}$

In figure 3 we show an example of non-similarity (non in-depth similarity) with respect to logical connectives between the formulas S_3 and S_4.

$$S_3: \quad (r_1 \wedge r_2) \Leftrightarrow (r_3 \vee r_4)$$
$$S_4: \quad ((r_5 \wedge r_6) \vee (r_7 \wedge r_8)) \Leftrightarrow r_9$$

Non in-depth similarity:

$f(1) = 1 \qquad \mathbf{M_1} = \{\mathbf{1}\}$
$M_2 = \{1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{1}\}$

$f(212) = 211 \qquad \mathbf{M_1} = \{\mathbf{212}, \mathbf{1}\}$
$M_2 = \{211, 1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{212}, \mathbf{1}\}$

$f(211) = 32111 \quad \mathbf{M_1} = \{\mathbf{211}, \mathbf{1}\}$
$M_2 = \{32111, 211, 1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{211}, \mathbf{212}, \mathbf{1}\}$

$f(211) = 32112 \quad \mathbf{M_1} = \{\mathbf{211}, \mathbf{1}\}$
$M_2 = \{32112, 211, 1\}$
$\mathbf{M_2^{-1}} = \{\mathbf{211}, \mathbf{212}, \mathbf{1}\}$

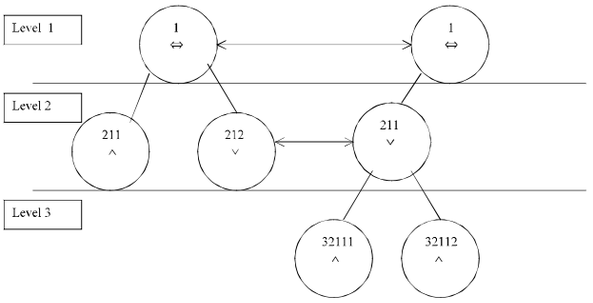

Fig. 3:

There is not any *f* such that $M_1 = M_2^{-1}$ for the node $n_{211} \in Parse(S_3)$.

In figure 4 we show another example of non-similarity (non in-breadth similarity) with respect to logical connectives between the formulas S_5 and S_6:

$S_5: \quad (r_1 \wedge r_2) \Leftrightarrow (r_3 \vee r_4)$
$S_6: \quad (r_5 \wedge r_6) \vee (r_7 \Leftrightarrow ((r_8 \vee r_9) \wedge (r_{10} \wedge r_{11})))$

In-depth similarity:

$f(1) = 211, \qquad \mathbf{M_1} = \{\mathbf{1}\}$
$M_2 = \{211\},$
$\mathbf{M_2^{-1}} = \{\mathbf{1}\}$

$f(211) = 4321222, \mathbf{M_1} = \{\mathbf{211}, \mathbf{1}\}$
$M_2 = \{4321222, 32122, 212, 1\},$
$\mathbf{M_2^{-1}} = \{\mathbf{211}, \mathbf{1}\}$

$f(212) = 4321221, \mathbf{M_1} = \{\mathbf{212}, \mathbf{1}\}$
$M_2 = \{4321221, 32122, 212, 1\},$
$\mathbf{M_2^{-1}} = \{\mathbf{212}, \mathbf{1}\}$

Non in-breadth similarity:

$l = 2$ (level 2), $x_1 = 1,\ p_1 = 1,\ x_2 = 1,\ p_2 = 2,\ N_{1,2} = \{2\mathbf{11}, 2\mathbf{12}\}$

$f(1) = 212,\ \pi_1 = 43,\ \varsigma_1 = 1,\ \pi_2 = 43,\ \varsigma_2 = 2,$
$N_{2,\{4\}} = \{43\mathbf{212}2\mathbf{1}, 43\mathbf{212}2\mathbf{2}\}$

DEFINITION 3.5 (SIMILARITY) *Let S_2 be similar to S_1 w.r.t. quantification and logical connectives by a function f. Let L be the set of all literals in S_1. Let N Lea f_1 be the set of all preterminal nodes in Parse(S_1). Let N Lea f_2 be the set of nodes $n_2 \in$ Parse(S_2) with $n_2 = f(n_1)$ and $n_1 \in NLeaf_1$. Let Lea f_2 be the set of leaf nodes in subtrees of Parse(S_2) with root node $n_2 \in NLeaf_2$. We say that S_2 is similar to S_1 iff there exist a $SLeaf_2 \subseteq Leaf_2$, with K as the set of literals in nodes of $SLeaf_2$, and a substitution σ induced by f such that $L\sigma = K$.*

For example, in figure 5, we show the similarity between $S_1 \equiv nocc(a, L, z) \Leftrightarrow nocc(a, S, z)$ and $S2 \equiv (v = b \wedge nocc(b, V, k)) \Leftrightarrow (w = b \wedge nocc(b, W, k))$

Similarity w.r.t to quantification:

$f(1) = 1 \quad f(211) = 211$

Similarity w.r.t. logical connectives (In-depth similarity):

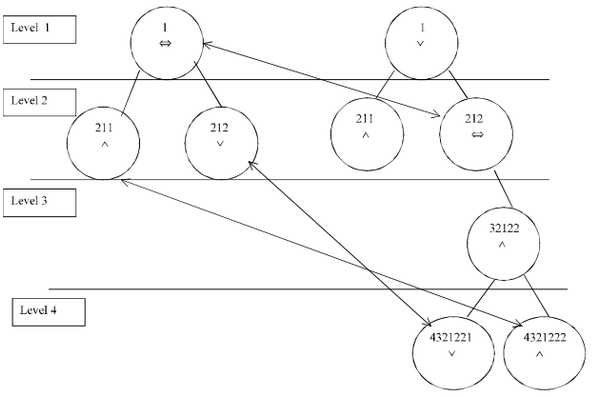

Fig. 4:

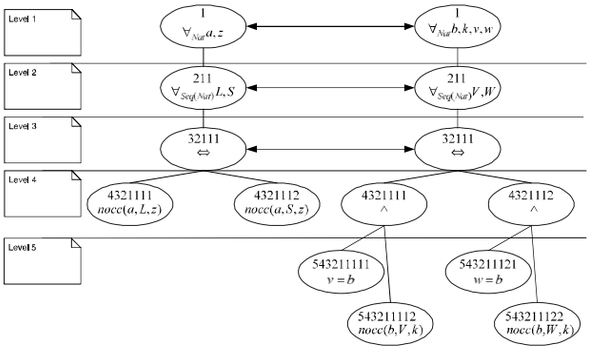

Fig. 5:

$f(32111) = 32111 \quad \mathbf{M_1} = \mathbf{M_2^{-1}} = \{\mathbf{32111}, \mathbf{211}, \mathbf{1}\}$

Similarity w.r.t. logical connectives
(In-breadth similarity):

$l = 3 \quad f(211) = 211 \quad N_{1,3}$
$= \{3\mathbf{2111}\} \quad N_{2,\{3\}} = \{3\mathbf{2111}\}$

$NLeaf_1 = \{32111\} \quad NLeaf_2 = \{32111\}$

$Leaf_2 = \{543211111, \quad 543211112,$
$543211121, \quad 543211122\}$

$SLeaf_2 = \{543211112, \quad 543211122\}$

Substitution σ induced by f:

$\sigma = \{(a,b),(z,k),(L,V),(S,W)\}$
$L = \{nocc(a,L,z), nocc(a,S,z)\},$
$K = \{nocc(b,V,k), nocc(b,W,k)\}, \quad L\sigma = K$

3.1 Algorithmic Justification for Similarity

The constructive nature of definitions 3.3, 3.4 and 3.5 can be justified in an algorithmic way. Different searching algorithms can be proposed for the construction of the similarity function *f*. We propose a construction following an incremental style. First, *f* is constructed in order to decide only about similarity w.r.t. quantifiers. Then, we search for a substitution induced by (this incomplete) *f*. Finally, we search for a remaining part of *f* which decides about similarity w.r.t. logical connectives. Our searching algorithm follows a generate-and-test strategy. It is possible to explore the complete search space due to the finite number of quantifiers and logical connectives in a formula.

Initially, a sequence of non-leaf node identifications is constructed by traversing *Parse*(S_1) in a breadth-first way. For example, for *Parse*(S_1) in figure 5 we obtain Seq_{S_1}:

$$Seq_{S_1} = \{1, 211, 32111\}$$

From this sequence, the subset of nodes containing quantified set of variables is selected. Then, a generate and test strategy is su±ce to construct (an incomplete due to the incremental construction) *f* which decides about similarity w.r.t. quantifiers. The generate-part generates a tentative *f* for each node in this subset. Hence, each node containing a quantified set of variables of the form $Q_\tau X$ in *Parse*(S_1) is bounded to a node containing a quantified set of variables of the form $Q_\tau Y$. The test part decides about conditions (a) and (b) in definition 3.3. If it is not possible to construct an *f* in these terms then we conclude that there is not any similarity w.r.t. quantification and then there is not any similarity *f*. For example, for *Parse*(*S*1) in figure 5, the subset of nodes containing quantified set of variables is equal to $\{1, 211\}$.

$f(1) = 1 \qquad Q_\tau = \forall_{Nat}$
$X = \{a,z\} \quad Y = \{b,k,v,w\}$
$f(211) = 211 \quad Q_\tau = \forall_{Seq(Nat)}$
$X = \{L,S\} \quad Y = \{V,W\}$

The set of all tentative substitutions σ_p induced by (our incomplete) *f* is computed in the following manner. For each pair of quantified set of variables $Q_\tau X$ and $Q_\tau Y$ by *f*, the set of all possible substitutions is calculated by means of a cartesian product.

Then, the set of all tentative substitutions σ_p induced by *f* is calculated by the cartesian product of these substitutions. The calculation of σ_p is a terminating problem due to the finite number of variables in a formula. For example, in our example (fig. 5):

For the pair $\forall_{Nat} a, z$ and $\forall_{Nat} b, k, v, w$:

$$\begin{aligned}\sigma_{\forall_{Nat}} \;=\; & \{\{(a,b),(z,k)\} \\ & \quad \{(a,b),(z,v)\} \\ & \quad\quad \{(a,b),(z,w)\} \\ & \quad \{(a,k),(z,b)\} \\ & \quad\quad \{(a,k),(z,v)\} \\ & \quad\quad\quad \{(a,k),(z,w)\} \\ & \quad \{(a,v),(z,b)\} \\ & \quad\quad \{(a,v),(z,k)\} \\ & \quad\quad\quad \{(a,v),(z,w)\} \\ & \quad \{(a,w),(z,b)\} \\ & \quad\quad \{(a,w),(z,k)\} \\ & \quad\quad\quad \{(a,w),(z,v)\}\}\end{aligned}$$

For the pair $\forall_{Seq(Nat)} L, S$ and $\forall_{Seq(Nat)} L, W$:

$$\begin{aligned}\sigma_{\forall_{Seq(Nat)}} \;=\; & \{\{(L,V),(S,W)\} \\ & \{\{(L,W),(S,L)\}\}\end{aligned}$$

$\sigma_p = \sigma_{\forall_{Nat}} \times \sigma_{\forall_{Seq(Nat)}}$:

$$\begin{aligned}\sigma_p \;=\; & \{\{(a,b),(z,k),(L,V),(S,W)\} \\ & \quad \{(a,b),(z,k),(L,W),(S,L)\} \quad ...\end{aligned}$$

Then, we select the sets of literals *L* from *Parse*(S_1) and *K* from *Parse*(S_2). Then σ is any substitution in σ_p such that $L\sigma = K$. A generate and test strategy is suffice to explore the σ_p search space. Only if *K* does not exist or there is not any σ such that $L\sigma = K$ then we conclude that there is not similarity w.r.t. quantification and then there is not any similarity *f*.

In a similar way, we construct the remaining part of *f* which is intended to decide about similarity w.r.t. logical connectives. A generate and test strategy is suffice to explore the search space. The generate-part generates a tentative (remaining part of) *f*. A breadth-first search is suffice to construct tentative *f*'s. The test-part decides about in-depth and in-breadth similarities induced by each tentative *f*. The search space for the remaining part of *f* is finite due to the finite number of logical connectives in a formula. In our example, the remaining part of *f* is only determined by the selection *f* (32111) = 32111. Finally, for our example (fig. 5):

$$\begin{aligned}f \;=\; & \{f(1) = 1 \\ & \; f(211) = 211 \\ & \; f(32111) = 32111\}\end{aligned}$$

Only if the remaining part of *f* can not be constructed then we conclude that there is not similarity w.r.t. logical connectives and then there is not any similarity *f*.

4 Similarity-based Folding Rule

In this section, a similarity-based folding rule is defined. Basically, it is a constructive definition of the folding rule in definition 2.5.

DEFINITION 4.1 (EVALUATION RULE) *Let S*$(l_1, l_2, ..., l_p, l_{p+1}, ..., l_n)$ *be a formula in the language of the context* $\mathcal{C}$ *constructed from literals* $l_1, l_2, ..., l_p, l_{p+1}, ..., l_n$. *We say that* $S_{eval}(\{l_1, l_2, ..., l_p\})$ *is obtained from S evaluating the set of literals* $\{l_1, l_2, ..., l_p\}$ *if and only if* $S_{eval}(\{l_1, l_2, ..., l_p\})$ *is of the following form:*

$$\begin{aligned}& S_{eval}(\{l_1, l_2, ..., l_p\}) \equiv \\ & \quad (S(true, true, ..., true, l_{p+1}, ..., l_n) \\ & \qquad \wedge \quad l_1 \wedge l_2 \wedge ... \wedge l_p) \qquad \vee \\ & \quad (S(false, true, ..., true, l_{p+1}, ..., l_n) \\ & \qquad \wedge \quad \neg l_1 \wedge l_2 \wedge ... \wedge l_p) \qquad \vee \\ & \quad (S(true, false, ..., true, l_{p+1}, ..., l_n) \\ & \qquad \wedge \quad l_1 \wedge \neg l_2 \wedge ... \wedge l_p) \qquad \vee \\ & \quad ... \\ & \quad (S(false, false..., false, l_{p+1}, ..., l_n) \\ & \qquad \wedge \quad \neg l_1 \wedge \neg l_2 \wedge ... \wedge \neg l_p) \quad \vee\end{aligned}$$

For example, let $S \equiv (v = b \wedge nocc(b, V, k)) \Leftrightarrow (w = b \wedge nocc(b, W, k))$ be a formula in the language of $\mathcal{S}$. Let $l_1 \equiv v = b$ and $l_3 \equiv w = b$ be two literals in *S*.

Then

$$\begin{aligned}& S_{eval}(\{l_1, l_3\}) \equiv \\ & \quad (true \wedge nocc(b, V, k) \Leftrightarrow true \wedge nocc(b, W, k)) \\ & \qquad \wedge \quad v = b \wedge w = b \qquad \vee \\ & \quad (false \wedge nocc(b, V, k) \Leftrightarrow true \wedge nocc(b, W, k)) \\ & \qquad \wedge \quad \neg v = b \wedge w = b \qquad \vee \\ & \quad (true \wedge nocc(b, V, k) \Leftrightarrow false \wedge nocc(b, W, k)) \\ & \qquad \wedge \quad v = b \wedge \neg w = b \qquad \vee \\ & \quad (false \wedge nocc(b, V, k) \Leftrightarrow false \wedge nocc(b, W, k)) \\ & \qquad \wedge \quad \neg v = b \wedge \neg w = b\end{aligned}$$

THEOREM 4.1 (CORRECTNESS OF THE EVALUATION RULE) *Let $S(l_1, ..., l_p, ..., l_n)$ be a formula in the language of $\mathcal{C}$ constructed from literals $l_1, ..., l_p, ..., l_n$.*

Let $S_{eval}(\{l_1, ..., l_p\})$ be the formula obtained from S evaluating the set of literals $\{l_1, ..., l_p\}$.

Let M be an isoinitial model for $\mathcal{C}$. Then $M \models S \Leftrightarrow S_{eval}(\{l_1, ..., l_p\})$

Proof 4.1 *Proof of $M \models S \Rightarrow S_{eval}(\{l_1, ..., l_p\})$. The evaluation rule (definition 4.1) constructs $S_{eval}(\{l_1, ..., l_p\})$ by means of 2^p mutually exclusive disjunctions representing all possible evaluation cases for $l_1, ..., l_p$ literals in S. Suppose (by absurdum) that M is model for a ground instance of S and it is not model for the respective ground instance of $S_{eval}(\{l_1, ..., l_p\})$. By hypothesis, $\mathcal{C}$ is atomically complete (definition 2.4) and then there exists a proof in $\mathcal{C}$ for the ground instance of S but there is not a proof in $\mathcal{C}$ for the ground instance of $S_{eval}(\{l_1, ..., l_p\})$. Hence, we conclude that $S_{eval}(\{l_1, ..., l_p\})$ does not consider all possible evaluations for $l_1, ..., l_p$ literals in S.*

Proof of $M \models S_{eval}(\{l_1, ..., l_p\}) \Rightarrow S$. If M is model of a ground instance of $S_{eval}(\{l_1, ..., l_p\})$ then M is model of only one instance of their disjunctions and then, by construction, it is a model of the respective ground instance of S.

Definition 4.2 (Rewrite Rules) *In order to simplify specifications, we consider a set of rewrite rules of the form $\{l \longrightarrow r\}$ in presence of negations and false and true propositions. A formula $S(true, false, l_{p+1}, ..., l_n)$ constructed from literals $l_{p+1}, ..., l_n$ and propositions true, false is transformed into the formula S_{rew} by application of rewrite rules repeatedly.*

$(1)\ \neg true \rightarrow false, \quad (2)\ \neg false \rightarrow true,$

$(3)\ true \vee F \rightarrow true$

$(4)\ false \vee F \rightarrow F, \quad (5)\ true \wedge F \rightarrow F,$

$(6)\ false \wedge F \rightarrow false$

$(7)\ false \Rightarrow F \rightarrow true, \quad (8)\ true \Rightarrow F \rightarrow F,$

$(9)\ false \Leftrightarrow F \rightarrow \neg F$

$(10)\ F \Rightarrow true \rightarrow true, \quad (11)\ F \Rightarrow false \rightarrow \neg F,$

$(12)\ true \Leftrightarrow F \rightarrow F$

$(13)\ \neg\neg F \rightarrow F, \quad (14)\ \neg(F \Rightarrow G) \rightarrow F \wedge \neg G,$

$(15)\ \neg(F \wedge G) \rightarrow \neg F \vee \neg G$

$(16)\ \neg(F \vee G) \rightarrow \neg F \wedge \neg G,$

$(17)\ \neg(F \Leftrightarrow G) \rightarrow \neg(F \Rightarrow G) \vee \neg(G \Rightarrow F)$

A formula S(true; false; $l_{p+1}, ..., l_n$) constructed from literals $l_{p+1}, ..., l_n$ and propositions true, false is transformed into the formula Srew by application of rewrite rules repeatedly.

For example, let $S = ((false \wedge nocc(b, V, k)) \Leftrightarrow (true \wedge nocc(b, W, k)))$ be a formula. $S_{rew} = \neg nocc(b, W, k)$ represents the simplified form of *S* obtained after the application of (6), (5) and (9) rewrite rules.

DEFINITION 4.3 (SIMILARITY-BASED FOLDING RULE) *Let S be a formula in the language of $\mathcal{C}$. Let r be a relation in $\mathcal{C}$. Let $r(x_j) \Leftrightarrow R_j(y_j)$ be an axiom in D_r. We say that S_j is obtained from S folding by similarity with respect to $r(x_j) \Leftrightarrow R_j(y_j)$ iff $S_j \equiv (S_{eval}(K - K_j))\ _{rew}|^{R_j(y_j)\theta_j}_{r(x_j)\theta_j}$ where*

1. *The variables appearing only in $R_j(y_j)$ but not in $r(x_j)$ do not appear in S.*
2. *K is the set of all literals in S.*
3. *S is similar to $R_j(y_j)$, where Lj is the set of all literals in $R_j(y_j)$ and K_j is a (sub)set of literals of S and θ_j is the substitution such that $L_j\theta_j = K_j$.*
4. *$(S_{eval}(K - K_j))$rew is obtained from S evaluating (definition 4.1) literals not in K_j and then applying rewrite rules (definition 4.2) repeatedly.*

For example, let $S \equiv (v = b \wedge nocc(b, V, k)) \Leftrightarrow (w = b \wedge nocc(b, W, k))$ be a formula in the language of $\mathcal{S}$. Let $perm(L, S) \Leftrightarrow (nocc(a, L, z) \Leftrightarrow nocc(a, S, z))$ be the axiom in D_{perm}. $K = \{v = b, nocc(b, V, k), w = b, nocc(b, W, k)\}$ is the set of all literals in *S*. *S* is similar to $nocc(a, L, z) \Leftrightarrow nocc(a, S, z)$ (definition 3.5). Let $L_1 = \{nocc(a, L, z), nocc(a, S, z)\}$be the set of literals in the right-hand side of the axiom in D_{perm} and let $K_1 = \{nocc\ (b, V, k), nocc(b, W, k)\}$ be the set of literals in S such that $L_1\theta_1 = K_1$ with $\theta_1 = \{(L, V), (S, W), (z, k), (a, b)\}$. Then, let

$$S_{eval}(K - K_1) \equiv$$
$$(true \wedge nocc(b,V,k) \Leftrightarrow true \wedge nocc(b,W,k))$$
$$\wedge \quad v = b \wedge w = b \quad \vee$$
$$(false \wedge nocc(b,V,k) \Leftrightarrow true \wedge nocc(b,W,k))$$
$$\wedge \quad \neg v = b \wedge w = b \quad \vee$$
$$(true \wedge nocc(b,V,k) \Leftrightarrow false \wedge nocc(b,W,k))$$
$$\wedge \quad v = b \wedge \neg w = b \quad \vee$$
$$(false \wedge nocc(b,V,k) \Leftrightarrow false \wedge nocc(b,W,k)))$$
$$\wedge \quad \neg v = b \wedge \neg w = b$$

be the formula obtained from S evaluating literals not in K_1 (i.e. $v = b$ and $w = b$). Applying (repeatedly) rewrite rules:

$$S_{eval}(K - K_1))_{rew} \equiv (nocc(b,V,k) \Leftrightarrow nocc(b,W,k))$$
$$\wedge \quad v = b \wedge w = b \quad \vee$$
$$\neg nocc(b,W,k)$$
$$\wedge \quad \neg v = b \wedge w = b \quad \vee$$
$$\neg nocc(b,V,k)$$
$$\wedge \quad v = b \wedge \neg w = b \quad \vee$$
$$\neg v = b \wedge \neg w = b$$

Considering $(nocc(a,L,z) \Leftrightarrow nocc(a,S,z))\theta_1 \equiv nocc(b,V,k) \Leftrightarrow nocc(b,W,k)$ and $perm(L,S)\theta_1 \equiv perm(V,W)$ then

$$(S_{eval}(K - K_1))_{rew} \,|^{nocc(b,V,k) \Leftrightarrow nocc(b,W,k)}_{perm(V,W)} \equiv$$
$$\mathbf{perm(V,W)} \wedge v = b \wedge w = b \vee$$
$$\neg nocc(b,W,k) \wedge \neg v = b \wedge w = b \vee$$
$$\neg nocc(b,V,k) \wedge v = b \wedge \neg w = b \vee$$
$$\neg v = b \wedge \neg w = b$$

is obtained from S folding by similarity with respect to $D_{perm,1}$.

Finally and reconsidering specification D_2 in section 1 (introduction) we obtain:

$$D_2: \quad perm(conc(v,V), conc(w,W)) \Leftrightarrow$$
$$(\mathbf{perm(V,W)} \wedge v = b \wedge w = b \vee$$
$$\neg nocc(b,W,k) \wedge \neg v = b \wedge w = b \vee$$
$$\neg nocc(b,V,k) \wedge v = b \wedge \neg w = b \vee$$
$$\neg v = b \wedge \neg w = b)$$

THEOREM 4.2 (CORRECTNESS OF THE SIMILARITY-BASED FOLDING RULE) *Let S be a formula in the language of $\mathcal{C}$. Let r be the relation in $\mathcal{C}$. Let $r(x_j) \Leftrightarrow R(y_j)$ be an axiom in D_r. Let S_j be the formula obtained from S folding by similarity with respect to $r(x_j) \Leftrightarrow R(y_j)$ (definition 4.3). Then*

$$M \models S \Leftrightarrow S_j \text{ where } S_j \equiv (S_{eval}(K - K_j))_{rew} \,|^{R_j(y_j)\theta_j}_{r(x_j)\theta_j}$$

Proof 4.2 *The similarity between S and $R_j(y_j)$ implies the existence of a substitution θ_j such that $L_j\theta_j = K_j$ where L_j represents the set of all literals in $R_j(y_j)$ and K_j (say $K_j = \{l_{p+1}, ..., l_n\}$) represents a (sub)set of literals in S (definition 4.3). The formula $S_{eval}(K - K_j)$ represents the evaluation of S with respect to literals not in K_j and by theorem 4.1, this formula is equivalent to S. This formula is composed by F_i disjunctions ($i = 1..2^p$). Applying rewrite rules on each disjunction, we obtain a formula of the form:*

$$(S_{eval}(K - K_j))_{rew} \equiv (F_1(l_{p+1}, ..., l_n)$$
$$\wedge \quad l_1 \wedge l_2 \wedge ... \wedge l_p) \quad \vee$$
$$(F_2(l_{p+1}, ..., l_n)$$
$$\wedge \quad \neg l_1 \wedge l_2 \wedge ... \wedge l_p) \quad \vee$$
$$(F_3(l_{p+1}, ..., l_n)$$
$$...$$
$$\wedge \quad l_1 \wedge \neg l_2 \wedge ... \wedge l_p) \quad \vee$$
$$(F_{2^p}(l_{p+1}, ..., l_n)$$
$$\wedge \quad \neg l_1 \wedge ... \wedge \neg l_p)$$

As rewrite rules preserve semantics, then $M \models S \Leftrightarrow (S_{eval}(K - K_j))_{rew}$. Considering the existence of $k \in \{1..2^p\}$ with $F_k(l_{p+1}, ..., l_n) = R_j(y_j)\theta_j$ then

$$M \models r(x_j) \Leftrightarrow R(y_j)$$
$$M \models r(x_j)\theta_j \Leftrightarrow R(y_j)\theta_j$$
$$M \models S \Leftrightarrow (S_{eval}(K - K_j))_{rew} \,|^{R_j(y_j)\theta_j}_{r(x_j)\theta_j}$$

5 Conclusions

The objective of applying transformations is to filter out a new version of the specification where recursion may be introduced by a folding step. Several (nonconstructive)

versions of the folding rule have been proposed mainly in the context of clausal (and restricted) specifications (e.g. logic programs [18] and [9]). We do not restrict the form of the specifications. Hence, it is possible to apply folding rule on general specifications in a flexible manner. On the other hand, constructive versions for this rule are needed if we are interested in the construction of automatic synthesizers. In this way, we propose a new folding rule which decides *how* to introduce recursive predicates in a specifications automatically which contrast with prior approaches. Our method is based on finding similarities between formulas represented as parsing trees and it constitutes an automatic assistance to the complex task of deriving recursive specifications from non recursive ones. At this point, an important problem remains to be solved. The "eureka" about *when* to apply folding rule is di±cult to establish in an automatic way [8]. The use of our proposal is intended to be integrated in a more general method which decides when apply such transformation (e.g. [10]). We think that our work is a little contribution towards the construction of automatic synthesizers.

References

[1] C. Aravindan and P. M. Dung. On the Correctness of Unfold/Fold Transformations of Normal and Extended Logic Programs. *(The Journal of Logic Programming)* 201-217 , 1995.

[2] A. Avellone, M. Ferrari and P. Miglioli. Synthesis of Programs in Abstract Data Types. 8*th* In *(Proceedings of the International Workshop on Logic Program Synthesis and Transformation)*. LNCS 1559, Springer, 1999, pages 81-100.

[3] A. Bertoni, G. Mauri and P. Miglioli. On the Power of Model Theory in Specifying Abstract Data Types and in capturing their Recursiveness. *(Fundamenta Informaticae)*, VI(2):27-170, 1983.

[4] A. Bundy, A. Smaill and G. Wiggins. The Synthesis of Logic Programs from Inductive Proofs. In *(Proceedings of Esprit Symposium on Computational Logic)*. Springer-Verlag, pages 135-149, 1990.

[5] R. M. Burstall y J. Darlington. A Transformational System for Developing Recursive Programs. *(Journal of the ACM)* 24(1):44-67, 1977.

[6] Y. Deville and K. K. Lau. Logic Program Synthesis. *(J. Logic Programming)* 19,20:321-350, 1994.

[7] R. G. Dromey. Systematic Program Development. *(IEEE Transaction of Software Engineering)*. 14(1):12-29, 1988.

[8] P. Flener. Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers, Massachusetts, 1995.

[9] P. A. Gardner and J. C. Shepherdson. Unfold/Fold Transformations of Logic Programs. MIT Press, pages 565-583, 1991.

[10] F. J. Galán and J. M. Cañete. Synthesis of Constructive Specifications. In *(Proceedings of the I Int. Workshop on Programming and Languages)*. Ed. F. Orejas. Almagro, Spain, 2001.

[11] K. K. Lau and M. Ornaghi. On Specification Frameworks and Deductive Synthesis of Logic Programs. In *(Proceedings of LOPSTR'94 and META'94)*. Springer-Verlag, 1994.

[12] E. Mendelson. Introduction to Mathematical Logic. Ed. Wadsworth & Brooks/Cole Advanced books & Software, Third edition, 1987.

[13] H. A. Partsch. Specification and Transformation of Programs: A Formal Approach to Software Development. Springer-Verlag 1990.

[14] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *(J. Logic Programming)* 19, 20: 261-320, 1994.

[15] M. Proietti and A. Pettorossi. An Abstract Strategy for Transforming Logic Programs. *(Fundamenta Informaticae)* 18:267-286, 1993.

[16] D. R. Smith. KIDS: A Semiautomatic program development system. *(IEEE Transaction of Software Engineering)* 16:1024-1043, 1990.

[17] D. Stuart Robertson, J. Agust. Pragmatics in the Synthesis of Logic Programs. In *(Proceedings of the* 8*th Int. Workshop on Logic Program Synthesis and Transformation)*. LNCS 1559, Springer, pages 41-60, 1999.

[18] Tamaki, H. and Sato, T. Unfold/Fold Transformation of Logic Programs. *Proceedings of the Second International Conference on Logic Programming*, Uppsala, Sweden, 1984, pp. 127-138.