SYNTHESIS AND TRANSFORMATION OF LOGIC PROGRAMS USING UNFOLD/FOLD PROOFS

ALBERTO PETTOROSSI AND MAURIZIO PROIETTI

▷ We present a method for proving properties of definite logic programs. This method is called *unfold/fold proof method* because it is based on the unfold/fold transformation rules. Given a program P and two goals (that is, conjunctions of atoms) $F(\overline{X},\overline{Y})$ and $G(\overline{X},\overline{Z})$, where $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ are pairwise disjoint vectors of variables, the unfold/fold proof method can be used to show that the equivalence formula $\forall\overline{X}\,(\exists\overline{Y}\,F(\overline{X},\overline{Y}) \leftrightarrow \exists\overline{Z}\,G(\overline{X},\overline{Z}))$ holds in the least Herbrand model of P. Equivalence formulas of that form can be used to justify goal replacement steps, which allow us to transform logic programs by replacing old goals, such as $F(\overline{X},\overline{Y})$, by equivalent new goals, such as $G(\overline{X},\overline{Z})$. These goal replacements preserve the least Herbrand model semantics if we find *non-ascending* unfold/fold proofs of the corresponding equivalence formulas, that is, unfold/fold proofs which ensure suitable well-founded orderings between the successful SLD-derivations of $F(\overline{X},\overline{Y})$ and $G(\overline{X},\overline{Z})$, respectively.

We also present a method for *program synthesis from implicit definitions*. It can be used to derive a definite logic program for the predicate *newp* implicitly defined by an equivalence formula of the form $\forall\,\overline{X}(\exists\overline{Y}\,F(\overline{X},\overline{Y}) \leftrightarrow \exists\overline{Z}\,(H(\overline{X},\overline{Z}), newp(\overline{X},\overline{Z})))$, such that the predicates occurring in the goals $F(\overline{X},\overline{Y})$ and $H(\overline{X},\overline{Z})$ are defined in a given program P, and *newp* is a predicate symbol not occurring in P. The set of clauses defining *newp*, say *Eureka*, allows us to prove that the above equivalence formula holds in the least Herbrand model of $P \cup Eureka$ using an unfold/fold proof. Thus,

Last revised June 15, 2009. This paper is a revised version of "Synthesis of Programs from Unfold/Fold Proofs". In: Yves Deville (ed.) Logic Program Synthesis and Transformation, LoPSTr '93, Louvain-la-Neuve, Belgium, Springer-Verlag, Workshops in Computing Series, 1994, 141–158. This work has been partially supported by: 'Progetto Coordinato del CNR Programmazione Logica', 'Progetto Coordinato del CNR Verifica, Analisi e Trasformazione dei Programmi Logici' Finsiel S.p.A., Progetto Cofinanziato MURST, INTAS Project, and Programma Galileo.

Address correspondence to Alberto Pettorossi, DISP, Università di Roma Tor Vergata, Via di Tor Vergata, I-00133 Roma, Italy; Maurizio Proietti, IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy; E-mail: {adp,proietti}@iasi.cnr.it. URL: http://www.iasi.cnr.it/~{adp,proietti}.

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1994

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

the correctness of our synthesis method derives from the one of the unfold/fold proof method. We finally illustrate our synthesis method through some examples of program specialization, program synthesis, and program transformation, which can all be viewed as program syntheses from implicit definitions.

◁

1. INTRODUCTION

The unfold/fold transformation rules were originally introduced for deriving correct and efficient programs from initial program versions whose correctness could easily be verified [8, 24, 28]. These rules can also be used for other purposes, such as program analysis, synthesis, specialization, and verification. Indeed, for instance, in [19] we can find a method based on unfold/fold rules, for proving the equivalence of functional expressions. This method can also be adapted to the case of logic programs [4, 26] for proving equivalences of goals, that is, equivalences of conjunctions of atoms. In this paper, which builds upon [26], we formalize this method, called *unfold/fold proof method*, for the case of definite logic programs w.r.t. the least Herbrand model semantics. We also present a method for *program synthesis from implicit definitions* which is based on the unfold/fold proof method and can be used for the specialization, synthesis, and transformation of programs.

In all these areas our synthesis method is very effective and powerful. In particular, (i) it provides a uniform framework for program specialization w.r.t. input properties rather than input values, (ii) it allows for the change of data structure representations, which is otherwise done in the literature using ad hoc techniques, and finally, (iii) it allows for the derivation of more efficient logic programs by avoiding unnecessary nondeterminism.

Our synthesis method is related to the traditional methods for synthesizing logic programs (see, for instance [16] and also [11] for a survey) from initial specifications of the form: $\forall X\, (spec(X) \leftrightarrow newp(X))$, where *newp* is the predicate for which we want to synthesize a program and *spec* is any formula of the first order predicate calculus which provides the specification of the predicate *newp*. The unfold/fold rules can indeed be viewed as derivation rules in these synthesis methods.

Our synthesis method is also related to the *proofs-as-programs* method [1, 7, 12, 23] whereby the constructive proof of a property of the form $\forall X\, \exists Y\, spec(X, Y)$ can be used for synthesizing a program which, for any input X, computes an output Y such that $spec(X, Y)$ holds.

The main difference between our method and the ones we have mentioned above is that we allow for a more general form of specifications. In particular, in the method for program synthesis from implicit definitions we assume that given a program P, the specification of a new program to be synthesized for the predicate $newp(\overline{X}, \overline{Z})$ is provided by an equivalence formula of the form (†): $\forall\, \overline{X}(\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, (H(\overline{X}, \overline{Z}), newp(\overline{X}, \overline{Z})))$, where $F(\overline{X}, \overline{Y})$ and $H(\overline{X}, \overline{Z})$ contain predicates defined in P and *newp* is a predicate symbol not occurring in P. We say that *newp* is implicitly defined by that formula. Here and in what follows, the conjunction connective is denoted by comma "," and overlined variables or

overlined terms stand for vectors of variables or terms, respectively. Through our synthesis method which we describe below and whose correctness derives from the one of the unfold/fold proof method, we construct a set of new clauses, say *Eureka*, which constitute the definition of $newp(\overline{X}, \overline{Z})$. That set allows us to show via an unfold/fold proof that the above equivalence formula of the form (†) holds in the least Herbrand model of $P \cup Eureka$.

In Section 2 we list the unfold/fold rules for program transformation which we consider in this paper. In Section 3 we present the unfold/fold proof method for logic programs by showing how to use our transformation rules for proving that given a program P and two goals $F(\overline{X}, \overline{Y})$ and $G(\overline{X}, \overline{Z})$, where $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ are pairwise disjoint vectors of variables, the equivalence formula $\forall \overline{X}(\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X}, \overline{Z}))$ holds in the least Herbrand model of P. In Section 4 we give a sufficient condition which ensures that goal replacements based on proofs of equivalence formulas preserve total correctness w.r.t. the least Herbrand model semantics. This condition is useful for the mechanization of the method for program synthesis from implicit definitions which is presented in Section 5. In Section 5 we also indicate how that synthesis method can be used to specialize programs. In particular, we show how it can be used for deriving programs which avoid type checking when the input values are known to be of the required type. In Section 6 we apply the program synthesis method to the automatic improvement of data representations by performing the so called difference-list introduction. In Section 7 we apply our synthesis method for avoiding unnecessary nondeterminism and deriving efficient right recursive programs from inefficient left recursive ones. Finally, in Section 8 we compare our method to related work in the areas of program specialization, program synthesis, and program transformation.

2. THE PROGRAM TRANSFORMATION RULES

In this section we introduce the rules that we use for transforming programs and we state the conditions which ensure that they preserve the least Herbrand model semantics. These rules are similar to the ones presented in [28], with the exception of the rules for definition introduction and for folding, which are similar to the ones in [14, 22]. In contrast to [28], the definition introduction rule considered here may be used to introduce a new predicate by means of n clauses, with $n \geq 1$ (in [28] n is 1), and the folding rule may be used to replace n clauses, with $n \geq 1$, by a single clause (in [28] n is 1).

In this paper we consider definite programs and for the notions not explicitly introduced here we refer to [20]. We assume that a *goal* is a *conjunction* of n (≥ 0) atoms defined as follows: $goal\ ::=\ true \mid atom \mid goal\ ,\ goal$ where the conjunction operator "," is associative and it has *true* as neutral element (in [20] a goal is the negation of a conjunction of atoms). We will refer to *true* as the *empty conjunction*, or the *empty goal*.

A *clause* C is a formula of the form $H \leftarrow B$, where the *head* H is an atom denoted by $hd(C)$ and the *body* B is a goal denoted by $bd(C)$. The clause $H \leftarrow true$ may also be written as $H \leftarrow$. A *definite program* (or *program*, for short) is a finite *set* of clauses. Programs will also be denoted without the surrounding curly brackets for sets.

By $\overline{t}$ we denote a vector of terms of the form $(t_1, \ldots, t_k)$, for some $k \geq 0$. The vector $(t_1, \ldots, t_k)$ is also written without its enclosing round parentheses. Given a

vector $\bar{t}$ and a partition $(\bar{t}_1, \ldots, \bar{t}_k)$ of $\bar{t}$ into order-preserving subvectors of contiguous components, we will feel free to identify $\bar{t}$ with $(\bar{t}_1, \ldots, \bar{t}_k)$. Thus, for instance, $(a, b, c) = ((a, b), (c))$. By $G(\overline{X})$, where $\overline{X}$ is a vector of variables, we denote a goal whose variables are among those in $\overline{X}$, and by $G(\bar{t})$ we denote the goal obtained from the one denoted by $G(\overline{X})$ by replacing each variable in $\overline{X}$ by the corresponding term in $\bar{t}$.

We allow for the silent renaming of the variables occurring in a clause, that is, we allow for the replacement of a clause by one of its *variants*. Obviously, variable renamings preserve the least Herbrand model semantics (see below).

We assume that all our programs are written using symbols taken from a fixed language L which contains an infinite set of variable symbols and an infinite set of function and predicate symbols. The Herbrand universe associated with L is denoted by HU, and this universe is assumed to be the same for all programs derived by transformation from a given initial program.

We also adopt the following notation: (i) given a substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, $dom(\theta)$ denotes the set of variables $\{X_1, \ldots, X_n\}$ and $range(\theta)$ denotes the set of terms $\{t_1, \ldots, t_n\}$, and (ii) given a term t, $vars(t)$ denotes the set of variables occurring in t (a similar notation will also be used for variables occurring in vectors of terms, atoms, goals, and clauses).

We assume the existence of a set of *basic* predicates which denote primitive relations and for which no defining clauses are given in the programs. This set includes the equality predicate '$=$'. For instance, predicates which may be considered to be basic, are '$\leq$' and *plus*. With each basic predicate, say b, it is associated the set S_b of atoms of the form $b(\bar{t})$, where $\bar{t}$ is a vector of ground terms in HU, such that $b(\bar{t})$ is assumed to be true.

Given a definite program P, by $M(P)$ we denote the least model among all Herbrand models of P which: (i) have universe HU, and (ii) include S_b for every basic predicate b. For simplicity, we feel free to refer to $M(P)$ as the least Herbrand model of P. As a consequence, the properties of the basic predicates, such as associativity of *plus*, hold in $M(P)$ for every program P we consider.

As for the operational semantics of the programs, the following definition of an SLD-derivation (which is a simplified version of the one in [20]) is adequate for our purposes here.

Let C be a (possibly renamed) clause in a program P and (L, A, M) be a goal, where A is an atom called the *selected atom*. We say that the goal $(L, B, M)\theta$, where θ is a substitution, is *derived from* (L, A, M) *using* P iff one of the following two conditions holds:

1. (i) A is an atom with non-basic predicate, (ii) θ is an mgu of A and $hd(C)$, and (iii) B is $bd(C)$.

2. (i) A is of the form $b(\overline{u})$, where b is a basic predicate, (ii) $b(\overline{u})\theta \in S_b$, and B is *true* (since *true* is the neutral element w.r.t. the conjunction operator, in this case $(L, B, M)\theta$ is equal to $(L, M)\theta$).

An SLD-derivation of the goal G using the program P is a (finite or infinite) sequence of goals $G_0, G_1, \ldots$ such that G_0 is G and for $i = 0, 1, \ldots$, the goal G_{i+1} is derived from the goal G_i using P. An SLD-derivation is *successful* iff it is finite and its last goal is *true*.

Given a finite SLD-derivation Δ of the form: $G_0, \ldots, G_n$, we denote by $\lambda(\Delta)$ the

number of indexes i, with $0 \leq i \leq n-1$, such that the selected atom in G_i does not have a basic predicate.

The program transformation process can be viewed as the construction of a sequence of programs, called a *transformation sequence*, starting from a given initial program P_0. Let us assume that we have constructed the transformation sequence $\langle P_0, \ldots, P_k \rangle$. We may then perform a transformation step and construct the next program P_{k+1} in the sequence, by applying one of the rules R1–R6 listed below, collectively called *unfold/fold* rules.

R1. *Definition introduction.* From program P_k we derive by *definition introduction* the new program P_{k+1} by adding to P_k the following n (≥ 1) new clauses:

$newp(\overline{X}) \leftarrow Body_1, \ \ldots, \ newp(\overline{X}) \leftarrow Body_n$

such that: 1) *newp* is a new predicate symbol, that is, it does not occur in $\langle P_0, \ldots, P_k \rangle$, and 2) for $j = 1, \ldots, n$, all predicate symbols occurring in the goal $Body_j$ occur in the initial program P_0. We say that *newp* is the predicate *defined* by those n clauses, which constitute the *definition* of *newp*.

During the construction of the sequence $\langle P_0, \ldots, P_k \rangle$ of programs, we store in the set Def_k, for $k \geq 0$, all clauses, called *definition clauses*, which have been introduced by the definition introduction rule. Obviously, $Def_0 = \{\,\}$.

R2. *Unfolding.* Let C and D be clauses such that: 1) C is a clause in P_k of the form $H \leftarrow F, A, G$, where A is an atom with a non-basic predicate, 2) D is a variant of a clause, call it D', such that $vars(C) \cap vars(D') = \{\,\}$, and the atoms $hd(D')$ and A are unifiable with mgu θ. The *unfolding* of C w.r.t. A *using* D is the clause $(H \leftarrow F, bd(D'), G)\theta$.

Let $D_1, \ldots, D_n$, with $n \geq 0$, be the clauses in program P_k such that for $i = 1, \ldots, n$, there is a variant of D_i, say D'_i, whose head $hd(D'_i)$ is unifiable with A. Let $C_1, \ldots, C_n$ be the unfoldings of C w.r.t. A using $D_1, \ldots, D_n$, respectively. By *unfolding* C w.r.t. A in P_k we derive the program $P_{k+1} = (P_k - \{C\}) \cup \{C_1, \ldots, C_n\}$. The atom A is said to be the *selected atom* for unfolding.

For $i = 1, \ldots, n$, we say that clause C_i is derived from C and we write $C \Rightarrow C_i$.

Notice that the unfolding of a clause C amounts to the removal of C from P_k if $n = 0$. Sometimes in the literature this particular case is treated as an extra rule called *clause removal* or *clause deletion* rule.

R3. *Folding.* Let $C_1, \ldots, C_n$, with $n \geq 1$, be clauses in program P_k. Let $D_1, \ldots, D_n$ be the clauses in Def_k which constitute the definition of a predicate, say *newp*. Let D_i be of the form $newp(\overline{X}) \leftarrow Body_i$, for $i = 1, \ldots, n$. Suppose that there exists a substitution θ such that for $i = 1, \ldots, n$, the following two conditions hold:

1. C_i is of the form $H \leftarrow F, Body_i\theta, G$, and
2. for every variable V occurring in $Body_i$ and not in $\overline{X}$, we have that: (i) $V\theta$ is a variable which does not occur in (H, F, G), and (ii) for any variable Y occurring in $Body_i$ and different from V, the variable $V\theta$ does not occur in $Y\theta$.

Let C be the clause $H \leftarrow F, newp(\overline{X})\theta, G$. By *folding* $C_1, \ldots, C_n$ we derive the new program $P_{k+1} = (P_k - \{C_1, \ldots, C_n\}) \cup \{C\}$.

For $i=1,\ldots,n$, we say that clause C is derived from C_i and we write $C_i \Rightarrow C$.

R4. *Goal replacement.* Let C be a clause in P_k of the form $H \leftarrow L, F(\overline{X},\overline{Y}), M$ and let $G(\overline{X},\overline{Z})$ be a goal. Let us assume that: (i) $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ are pairwise disjoint vectors of variables, (ii) $vars(H,L,M) \cap vars(\overline{Y},\overline{Z}) = \{\,\}$, (iii) the predicates occurring in $F(\overline{X},\overline{Y})$ and the predicates occurring in $G(\overline{X},\overline{Z})$ occur in P_0, and (iv) $M(P_0) \models \forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X},\overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X},\overline{Z}))$.

Let D be the clause $H \leftarrow L, G(\overline{X},\overline{Z}), M$. By *goal replacement* we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{D\}$.

We say that clause D is derived from C and we write $C \Rightarrow D$.

Notice that rule R4 is a *self-inverse*, in the sense that if P_{k+1} can be derived from P_k by goal replacement, then a program P_{k+2} equal to P_k can be derived from P_{k+1} by goal replacement. Obviously, for the goal replacement from P_{k+1} to P_{k+2} we use the fact that $M(P_0) \models \forall \overline{X}\, (\exists \overline{Z}\, G(\overline{X},\overline{Z}) \leftrightarrow \exists \overline{Y}\, F(\overline{X},\overline{Y}))$ holds. Thus, if $C \Rightarrow D$ holds by rule R4, then $D \Rightarrow C$ holds by rule R4.

R5. *Generalization + equality introduction.* Let C be a clause in program P_k of the form $(H \leftarrow Body)\{X/t\}$, such that the variable X does not occur in t. By generalization + equality introduction we derive from C the clause D: $H \leftarrow X=t, Body$ and we get the program P_{k+1} by replacing C by D in P_k.

We say that clause D is derived from C and we write $C \Rightarrow D$.

R6. *Simplification of equality.* Let C be a clause in program P_k of the form: $H \leftarrow X=t, Body$, where X does not occur in t. By simplification of equality we derive from C the clause D: $(H \leftarrow Body)\{X/t\}$, and we get the program P_{k+1} by replacing C by D in P_k.

We say that clause D is derived from C and we write $C \Rightarrow D$.

Rule R6 is the inverse of rule R5 in the sense that, if P_{k+1} can be derived from P_k by rule R5, then a program P_{k+2} equal to P_k can be derived from P_{k+1} by rule R6. Thus, if $C \Rightarrow D$ holds by rule R5, then $D \Rightarrow C$ holds by rule R6. Analogously, rule R5 is the inverse of rule R6.

We stipulate that the $\Rightarrow$ relation is closed w.r.t. variable renaming, that is, if $C \Rightarrow D$ holds for two clauses C and D, then $C' \Rightarrow D'$ holds for any variant C' and D' of C and D, respectively.

A *derivation path* from clause C_0 to clause C_n is a sequence $C_0,\ldots,C_n$ of clauses, with $n \geq 0$, such that for $i=0,\ldots,n-1$, $C_i \Rightarrow C_{i+1}$. A derivation path from C_0 to C_n is also written as $C_0 \Rightarrow \ldots \Rightarrow C_n$. There exists a derivation path from C_0 to C_n iff $C_0 \Rightarrow^* C_n$, where as usual, $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

The transformation rules R1–R6 preserve the least Herbrand model semantics as specified by the following Definition 2.1 and Theorem 2.1.

Definition 2.1. [Non-ascending goal replacement] Let $\langle P_0,\ldots,P_k\rangle$ be a transformation sequence. Given a goal H such that $\overline{W}$ is the vector of the variables occurring in H and $M(P_0) \models \exists \overline{W}\, H$, we define $\mu(H) = min\{\lambda(\Delta) \mid \Delta$ is a successful SLD-derivation of H using $P_0\}$. Given two goals $F(\overline{X},\overline{Y})$ and $G(\overline{X},\overline{Z})$ such that $M(P_0) \models \forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X},\overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X},\overline{Z}))$ we say that the replacement of $F(\overline{X},\overline{Y})$ by $G(\overline{X},\overline{Z})$ in the body of a clause in P_k is *non-ascending* iff for each vector $\overline{t}$ of ground terms such that $M(P_0) \models \exists \overline{Y}\, F(\overline{t},\overline{Y})$ we have that $\mu(F(\overline{t},\overline{Y})) \geq \mu(G(\overline{t},\overline{Z}))$.

Theorem 2.1. [Total correctness of a transformation sequence] *Let* $\langle P_0, \ldots, P_k \rangle$ *be a transformation sequence constructed by using the transformation rules* R1–R6 *with the following restrictions:*

(α) *the folding rule is applied in program* P_h*, with* $0 < h < k$*, to clauses* $C_1, \ldots, C_m$ *with head predicate* p *using clauses* $D_1, \ldots, D_m$ *only if*

- *either* p *occurs in* P_0
- *or for every* $i \in \{1, \ldots, m\}$*, there exist two clauses, say* A *in* P_j *and* B *in* P_{j+1}*, for some* j*, with* $0 \leq j < h$*, such that* $A \Rightarrow B \Rightarrow^* C_i$ *where* B *is derived from* A *by unfolding, and*

(β) *all goal replacements are non-ascending.*

Then $M(P_0 \cup Def_k) = M(P_k)$.

Proof. It is an extension of the correctness results reported in [14, 29]. The total correctness of the rules R1–R6 presented above is proved in [29] (see Theorem 3.7), with the restriction that the folding rule is allowed only in the case where the number of folded clauses is 1. In particular, our notion of non-ascending goal replacement is subsumed by the notion of goal replacement *consistent with a weight-tuple measure* used in the proof of Theorem 3.7 in [29]. The correctness of our more general folding rule R3 by which we derive a new clause by folding n (≥ 1) clauses at a time, is proved by Theorem 1 of [14]. However, in [14, 29] the correctness of each rule is not proved in isolation. On the contrary, in those papers the total correctness of an entire transformation sequence is proved when some suitable conditions on the set of rules and on the order of their application are satisfied. (These conditions are similar to our restrictions (α) and (β).) Thus, our total correctness theorem is not a straightforward consequence of the above mentioned results. Nevertheless, the proofs in [29] can be extended to the case where we use the more general folding rule R3. We do not present this extension here because the amount of technical machinery is rather large and the differences from the proof in [29] are of minor importance. □

3. THE UNFOLD/FOLD PROOF METHOD

In this section we present the unfold/fold proof method following the approach described in [19, 26]. This method can be used to prove that an equivalence formula, say *Equiv*, of the form $\forall \overline{X} \, (\exists \overline{Y} \, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z} \, G(\overline{X}, \overline{Z}))$ holds in the least Herbrand model of a given program P. The soundness of the method (see Theorem 3.1) relies on the correctness of the transformation rules R1–R6 w.r.t. the least Herbrand model semantics (see Theorem 2.1).

Definition 3.1. [Unfold/fold proof] Let P be a program and *Equiv* be the formula $\forall \overline{X} \, (\exists \overline{Y} \, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z} \, G(\overline{X}, \overline{Z}))$, where $\overline{X}, \overline{Y}$, and $\overline{Z}$ are pairwise disjoint vectors of variables and $F(\overline{X}, \overline{Y})$ and $G(\overline{X}, \overline{Z})$ are two given goals. An *unfold/fold proof* of *Equiv* using P consists of two *totally correct* transformation sequences $T_1 : \langle P, P \cup \{C_1\}, \ldots, P \cup S_1 \rangle$ and $T_2 : \langle P, P \cup \{C_2\}, \ldots, P \cup S_2 \rangle$ (see Fig. 3.1) such that:

$P \cup$ $\boxed{C_1 : new1(\overline{X}) \leftarrow F(\overline{X},\overline{Y})}$ $\qquad$ $\boxed{C_2 : new2(\overline{X}) \leftarrow G(\overline{X},\overline{Y})}$ $\cup P$

$(\text{R2}+\text{R3}+\text{R4}+\text{R5}+\text{R6})^*$ $\downarrow$ $\qquad$ $\downarrow$ $(\text{R2}+\text{R3}+\text{R4}+\text{R5}+\text{R6})^*$

$\{new1/new2\}$

$P \cup \boxed{S_1} \;=\; \boxed{S_2} \cup P$

FIGURE 3.1. Unfold/fold proof of $\forall \overline{X}\,(\exists \overline{Y}\,F(\overline{X},\overline{Y}) \leftrightarrow \exists \overline{Z}\,G(\overline{X},\overline{Z}))$ using program P.

(i) the sequences T_1 and T_2 are constructed by first adding, using rule R1, to program P the following definition clauses C_1 and C_2, respectively:

$C_1 :\; new1(\overline{X}) \leftarrow F(\overline{X},\overline{Y})$

$C_2 :\; new2(\overline{X}) \leftarrow G(\overline{X},\overline{Z})$

and then applying a sequence of transformation rules, each of which is taken from the set $\{\text{R2},\text{R3},\text{R4},\text{R5},\text{R6}\}$,

(ii) for $i = 1,2$, and for each clause D derived during the construction of T_i we have that $C_i \Rightarrow^* D$, and

(iii) S_2 can be obtained from S_1 by substituting the predicate symbol *new*2 for *new*1. $\square$

Notice that, by the folding rule R3 in T_1 every folding step is performed using clause C_1 only, and analogously, in T_2 every folding step is performed using C_2 only.

The following theorem shows that the unfold/fold proof method is *sound* w.r.t. the least Herbrand model semantics.

Theorem 3.1. [Soundness of the unfold/fold proof method] *If there exists an unfold/fold proof of the formula* $\forall \overline{X}\,(\exists \overline{Y}\,F(\overline{X},\overline{Y}) \leftrightarrow \exists \overline{Z}\,G(\overline{X},\overline{Z}))$ *using* P*, then we have that:* $M(P) \models \forall \overline{X}\,(\exists \overline{Y}\,F(\overline{X},\overline{Y}) \leftrightarrow \exists \overline{Z}\,G(\overline{X},\overline{Z}))$.

Proof. Without loss of generality, let us assume that each of the vectors $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ in the formula $\forall \overline{X}\,(\exists \overline{Y}\,F(\overline{X},\overline{Y}) \leftrightarrow \exists \overline{Z}\,G(\overline{X},\overline{Z}))$ consists of one variable only. Suppose that the unfold/fold proof of $\forall X\,(\exists Y\,F(X,Y) \leftrightarrow \exists Z\,G(X,Z))$ consists of the transformation sequences T_1 and T_2 constructed as indicated in Definition 3.1. By our assumptions of Section 2 we have that the Herbrand universe HU is the same for all programs in the transformation sequences T_1 and T_2. Since S_2 is equal to S_1 modulo the substitution of *new*2 for *new*1, we have that for every term $t \in HU$:

$$M(P \cup S_1) \models new1(t) \quad \text{iff} \quad M(P \cup S_2) \models new2(t).$$

In $P \cup \{C_1\}$ the predicate *new*1 is defined by clause C_1 only, and in $P \cup \{C_2\}$ the predicate *new*2 is defined by clause C_2 only. Thus, we have that the following two properties hold for every term $t \in HU$:

$$M(P \cup \{C_1\}) \models new1(t) \quad \text{iff} \quad M(P \cup \{C_1\}) \models \exists Y\,F(t,Y)$$

$$M(P \cup \{C_2\}) \models new2(t) \quad \text{iff} \quad M(P \cup \{C_2\}) \models \exists Z\,G(t,Z).$$

From the assumption that T_1 and T_2 are totally correct (see Definition 3.1), it follows that $M(P\cup S_1)=M(P\cup\{C_1\})$ and $M(P\cup S_2)=M(P\cup\{C_2\})$. Thus, we have that for every term $t\in HU$:

$$M(P\cup\{C_1\})\models\exists Y\,F(t,Y)\quad\text{iff}\quad M(P\cup\{C_2\})\models\exists Z\,G(t,Z).$$

Now, since the predicate symbols occurring in $F(t,Y)$ and $G(t,Z)$ do not depend on $new1$ and $new2$ (because $new1$ and $new2$ are new predicate symbols), we can replace both $M(P\cup\{C_1\})$ and $M(P\cup\{C_2\})$ by $M(P)$ and we conclude that for every term $t\in HU$:

$$M(P)\models\exists Y\,F(t,Y)\quad\text{iff}\quad M(P)\models\exists Z\,G(t,Z).$$

Finally, by observing that HU is the universe of $M(P)$ we get:

$$M(P)\models\forall X\,(\exists Y\,F(X,Y)\leftrightarrow\exists Z\,G(X,Z)).\quad\Box$$

The following example shows an application of the unfold/fold proof method.

Example 3.1. [Functionality of Fibonacci] We give the unfold/fold proof of the functionality of the Fibonacci predicate. We recall that a predicate $p(\overline{X},\overline{Y})$ is said to be *functional* w.r.t. $\overline{X}$ in a program P iff we have that:

for all vectors $\overline{t},\overline{u}$, and $\overline{v}$ of ground terms, $M(P)\models(p(\overline{t},\overline{u}),p(\overline{t},\overline{v}))\rightarrow\overline{u}=\overline{v}$

which is equivalent to:

$M(P)\models\forall\overline{X},\overline{Y},\overline{Z}\,((p(\overline{X},\overline{Y}),p(\overline{X},\overline{Z}))\leftrightarrow(p(\overline{X},\overline{Y}),\overline{Y}=\overline{Z})).$

Let us consider the following program *Fib* for the computation of the Fibonacci numbers:

$fib(0,s(0))\leftarrow$
$fib(s(0),s(0))\leftarrow$
$fib(s(s(N)),F2)\leftarrow fib(s(N),F1),\ fib(N,F),\ plus(F1,F,F2)$

where the predicate $plus(X,Y,Z)$ is a basic predicate which is assumed to be functional w.r.t. X and Y, that is, the following formula holds in $M(Fib)$ (recall that the least Herbrand model of a program includes the true ground facts about all basic predicates):

$\forall X,Y,Z1,Z2\,((plus(X,Y,Z1),\ plus(X,Y,Z2))\leftrightarrow(plus(X,Y,Z1),\ Z1=Z2)).$

We want to prove that the predicate $fib(N,F)$ is functional w.r.t. N, that is, the following equivalence formula holds in $M(Fib)$:

*Equiv*1: $\forall N,F1,F2\,((fib(N,F1),\ fib(N,F2))\leftrightarrow(fib(N,F1),\ F1=F2)).$

We apply the unfold/fold proof method and we introduce the following two clauses:

C_1: $new1(N,F1,F2)\leftarrow fib(N,F1),\ fib(N,F2)$
C_2: $new2(N,F1,F2)\leftarrow fib(N,F1),\ F1=F2$

By applying the transformation rules, clauses C_1 and C_2 can be transformed into the two sets of clauses S_1 and S_2, respectively, shown in Fig. 3.2. Those two sets are equal modulo predicate renaming. Therefore, *Equiv*1 holds in $M(Fib)$.

Notice that the functionality of *plus*, which is used as a lemma in the above unfold/fold proof, can also be proved by the unfold/fold method, in case the predicate *plus* is considered to be a non-basic predicate and its defining clauses are given in the program. □

The reader should notice that the unfold/fold proof method cannot be used for proving that a formula φ is a logical consequence of a program P (i.e. φ is true in *all* models of P) because in general the unfold/fold rules do not preserve all models of P but only the least Herbrand model.

An important issue is how to find unfold/fold proofs in a mechanical way. Obviously, the existence of unfold/fold proofs is undecidable, in general. As usual in the field of automated theorem proving, we may cope with this limitation by (i) suitably restricting the class of programs we consider, and/or (ii) adopting strategies which may help us construct the two transformation sequences T_1 and T_2 required by our proof method. We will not further discuss here this mechanization issue. However, in Section 5 we will propose a strategy for finding the so called *Eureka* sets of clauses, which when successful, allows us to automatically construct the transformation sequences T_1 and T_2.

In the following Section 4, we will further study the correctness of the unfold/fold proof method w.r.t. the goal replacement rule, and in later sections we will illustrate some applications of this method to the areas of program synthesis and program transformation.

4. UNFOLD/FOLD PROOFS AND GOAL REPLACEMENT

In the previous section we have seen that proofs of equivalence formulas are needed to apply the goal replacement rule. However, these proofs are *not* sufficient for ensuring the total correctness of a transformation sequence when it includes a goal replacement step, as the following example shows.

Example 4.1. Let us consider the program

$P: \quad p \leftarrow q, \qquad q \leftarrow$

We have that $M(P) \models p \leftrightarrow q$. By replacing q by p in $p \leftarrow q$ we get:

$Q: \quad p \leftarrow p, \qquad q \leftarrow$

and $M(Q) = \{q\} \neq \{p, q\} = M(P)$. □

As stated by Theorem 2.1, a sufficient condition for the total correctness of a transformation sequence is that goal replacements are performed only if they are non-ascending. In what follows we provide a sufficient condition for ensuring that a goal replacement is non-ascending. Our condition relies on the construction of a suitable unfold/fold proof of the equivalence formula which justifies the goal replacement.

Definition 4.1. [Non-ascending unfold/fold proof] Let P be a program and *Equiv* be the formula $\forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X}, \overline{Z}))$, where $\overline{X}, \overline{Y}$, and $\overline{Z}$ are pairwise disjoint vectors of variables and $F(\overline{X}, \overline{Y})$ and $G(\overline{X}, \overline{Z})$ are two given goals.

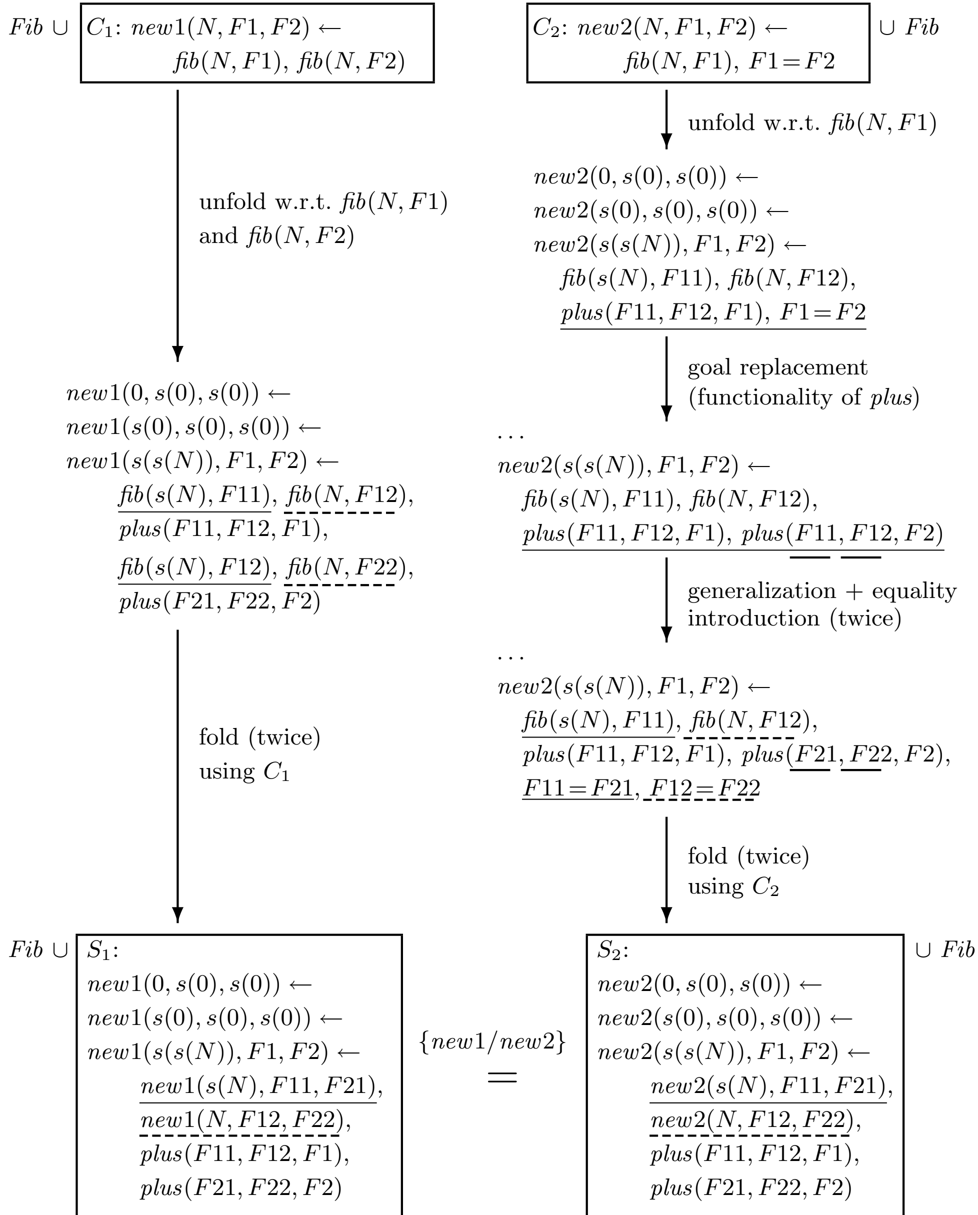

FIGURE 3.2. Unfold/fold proof of the functionality of the Fibonacci predicate *fib* assuming the functionality of the predicate *plus*.

A *non-ascending* unfold/fold proof of *Equiv* consists of two transformation sequences T_1 : $\langle P, P \cup \{C_1\}, \ldots, P \cup S_1\rangle$ and T_2 : $\langle P, P \cup \{C_2\}, \ldots, P \cup S_2\rangle$ satisfying the properties at Points (i)–(iii) of Definition 3.1 and also satisfying the following properties:

(iv) Each transformation sequence includes at least one unfolding step.

(v) In each transformation sequence if a folding step is not the last one, then it is followed by folding steps only. (Recall that, by our folding rule R3, in T_1 every folding step is performed using clause C_1 only, and analogously, in T_2 every folding step is performed using C_2 only.)

(vi) In each transformation sequence each application of the goal replacement rule which replaces goal G_1 by goal G_2 is restricted to one of the following two cases: (1) in G_1 and G_2 there are basic predicates only; (2) G_1 is of the form $(H1, H2)$ and G_2 is of the form $(H2, H1)$ (that is, the goal replacement consists in rearranging the order of the goals). In these two cases the equivalence formulas which justify the goal replacements hold in the least Herbrand model of every program.

(vii) For each derivation path $R_1: C_1 \Rightarrow \ldots \Rightarrow L$, where L is a clause in S_1 there exists a derivation path $R_2: C_2 \Rightarrow \ldots \Rightarrow M$, where M is a clause in S_2 such that: (1) M can be obtained from L by replacing every occurrence of *new*1 by *new*2, and (2) the number of clauses obtained by unfolding steps and occurring in R_1, is not less than the number of clauses obtained by unfolding steps and occurring in R_2.

An example of a non-ascending unfold/fold proof is given by the proof of the functionality of the Fibonacci predicate in Example 3.1. By the following theorem we have that goal replacement steps justified by non-ascending unfold/fold proofs (see Definition 4.1) are non-ascending (see Definition 2.1) and thus, by Theorem 2.1, they are totally correct w.r.t. the least Herbrand model semantics.

Theorem 4.1. Let P_0 be a program and let $F(\overline{X}, \overline{Y})$ and $G(\overline{X}, \overline{Z})$ be goals. If there exists a non-ascending unfold/fold proof of $\forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X}, \overline{Z}))$ using P_0, then for each transformation sequence $\langle P_0, \ldots, P_k\rangle$ the replacement of $F(\overline{X}, \overline{Y})$ by $G(\overline{X}, \overline{Z})$ in the body of a clause in P_k is a non-ascending goal replacement.

PROOF. See Appendix.

5. A METHOD FOR PROGRAM SYNTHESIS FROM IMPLICIT DEFINITIONS AND ITS APPLICATION TO PROGRAM SPECIALIZATION

In this section we present a method for the synthesis of programs from implicit definitions, and we see an example of its use for program specialization.

We assume that given a program P, the set of clauses to be synthesized for a new predicate, say *newp*, not occurring in P, is specified by a closed formula

*Equiv*2 of the form: $\forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, (H(\overline{X}, \overline{Z}), newp(\overline{X}, \overline{Z})))$, where: (i) the predicates occurring in the goals $F(\overline{X}, \overline{Y})$ and $H(\overline{X}, \overline{Z})$ are defined in P and (ii) $\overline{X}, \overline{Y}$ and $\overline{Z}$ are pairwise disjoint vectors of variables. The formula *Equiv*2 is said to be an *implicit definition* of *newp*.

Recall that a variable in $(\overline{U}, \overline{V})$ need not occur in a goal denoted by $G(\overline{U}, \overline{V})$ (see Section 2). For instance, the atom $p(X)$ is among the goals denoted by $G(X, Y)$. Analogously, we stipulate that $newp(\overline{X}, \overline{Z})$ denotes an atom with predicate *newp* all of whose arguments are variables taken from the vector $(\overline{X}, \overline{Z})$.

Thus, the following formulas are examples of implicit definitions of a predicate *newp*:

$\forall X\, (\exists Y\, f(X, Y) \leftrightarrow \exists Z\, (h_1(X), h_2(Z), newp(X, Z)))$
$\forall X\, (\exists Y\, f(X, Y) \leftrightarrow \exists Z\, (h(X), newp(X, Z)))$
$\forall X\, (\exists Y\, f(X, Y) \leftrightarrow \exists Z\, (h(Z), newp(X, Z)))$
$\forall X\, (\exists Y\, f(X) \leftrightarrow (h(X), newp(X)))$

The method for program synthesis we present here, has the objective of generating a set of clauses, say *Eureka*, which provide a definition of *newp*, such that $M(P \cup Eureka) \models Equiv2$. The reader who is familiar with the abduction theory, may realize that the task of generating the set *Eureka* can be viewed as an instance of an *abduction problem* [17], where *Equiv*2 is the *observed formula* and *Eureka* is a set of *abductive explanations* to be added to P for justifying the formula *Equiv*2.

We will now present our synthesis method by looking, at the same time, at its application to a *program specialization* problem which can be stated as follows.

Given a program P, a predicate $p(X)$ defined in P, and a set I of input values, the problem of specializing $p(X)$ w.r.t. I is the problem of generating a set *Eureka* of clauses defining a new predicate $spec_p(X)$ such that $p(X)$ is equivalent to $spec_p(X)$ for all X in I.

We assume that I is specified by a predicate defined in P, say $input(X)$, such that X belongs to I iff $input(X)$ holds in the least Herbrand model of P. Thus, the problem of specializing $p(X)$ w.r.t. I is the problem of synthesizing a set *Eureka* of clauses defining $spec_p(X)$, such that

$$M(P \cup Eureka) \models \forall X\, ((input(X),\ p(X)) \leftrightarrow (input(X),\ spec_p(X)))$$

Obviously, as a trivial solution of this problem we may choose $spec_p(X)$ to be $p(X)$ itself. However, this trivial solution is not of interest to us. In Examples 5.1, 5.2, and 5.3 we will show that our synthesis method from implicit definitions is powerful enough to produce a non-trivial solution different from $p(X)$.

Example 5.1. [Specializing List Concatenation with Type Checks] Let us consider the following program *LConcat* for concatenating lists:

$concat([\,], Ys, Ys) \leftarrow list(Ys)$
$concat([X|Xs], Ys, [X|Zs]) \leftarrow list(Xs),\ list(Ys),\ list(Zs),\ concat(Xs, Ys, Zs)$
$list([\,]) \leftarrow$
$list([X|Xs]) \leftarrow list(Xs)$

Similarly to [13], we would like to specialize our predicate $concat(Xs, Ys, Zs)$ w.r.t. the set of triples (Xs, Ys, Zs) in the Herbrand universe such that the conjunction $list(Xs),\ list(Ys),\ list(Zs)$ holds. Thus, we would like to introduce a new predicate, say $conc(Xs, Ys, Zs)$, and generate a set *Eureka* of clauses such that:

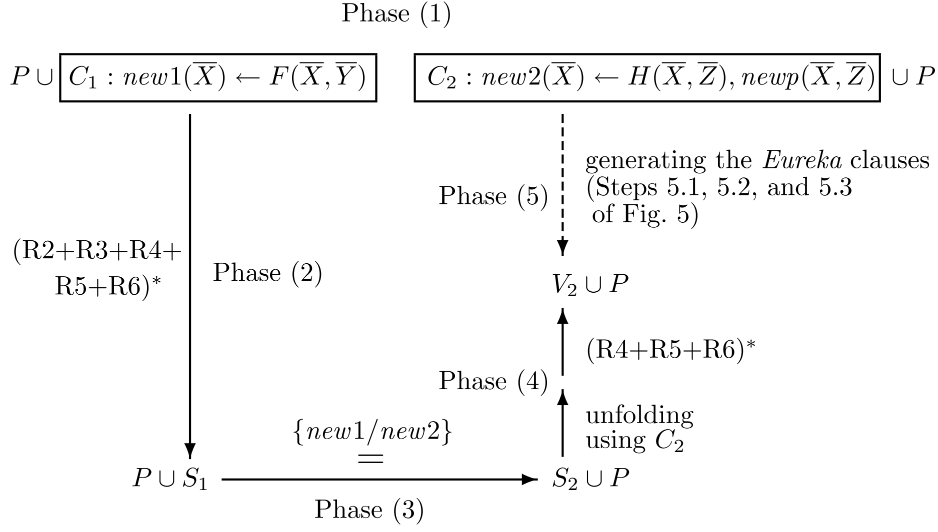

FIGURE 5.1. Five phase synthesis method from implicit definitions using unfold/fold proofs.

$M(LConcat \cup Eureka) \models \forall Xs, Ys, Zs$
$(list(Xs),\ list(Ys),\ list(Zs),\ concat(Xs, Ys, Zs)$
$\leftrightarrow list(Xs),\ list(Ys),\ list(Zs),\ conc(Xs, Ys, Zs))$

The *Eureka* clauses defining the predicate *conc* should perform list concatenation of Xs and Ys to produce Zs, without checking that the values of these variables are lists. □

This example will be continued below (see Examples 5.2 and 5.3). Let us now present our method for program synthesis from implicit definitions with reference to a given program P and a formula *Equiv*2 of the form: $\forall \overline{X}\,(\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\,(H(\overline{X}, \overline{Z}), newp(\overline{X}, \overline{Z})))$, where $F(\overline{X}, \overline{Y})$ and $H(\overline{X}, \overline{Z})$ are goals, and $\overline{X}, \overline{Y}$ and $\overline{Z}$ are pairwise disjoint vectors of variables. It consists of the following five phases (see also Fig. 5.1).

Phase (1). We introduce the following two clauses:

C_1: $new1(\overline{X}) \leftarrow F(\overline{X}, \overline{Y})$

C_2: $new2(\overline{X}) \leftarrow H(\overline{X}, \overline{Z}),\ newp(\overline{X}, \overline{Z})$

where $new1$ and $new2$ are predicate symbols not occurring in P and for $i = 1, 2$, all the universally quantified variables of *Equiv*2 occur in the head of C_i.

Phase (2). We construct a totally correct transformation sequence $\langle P,\ P \cup \{C_1\}, \ldots,\ P \cup S_1\rangle$ by first adding (using rule R1) clause C_1 to program P and then applying a sequence of transformation rules, each of which is

taken from the set $\{R2, R3, R4, R5, R6\}$. We assume that each clause in S_1 is derived, in zero or more steps, from clause C_1, that is, for each clause D not in P, derived during the construction of the transformation sequence, we have that $C_1 \Rightarrow^* D$.

Phase (3). We get a set S_2 of clauses from the set S_1 by replacing every occurrence of the predicate symbol *new*1 by *new*2.

Phase (4). We eliminate the occurrences of *new*2 from the bodies of the clauses in S_2 by performing some unfolding steps using clause C_2. We then apply zero or more times the rules R4, R5, and R6, thereby getting a new set of clauses, say V_2.

Phase (5). We generate a set *Eureka* of clauses which allows us to construct a transformation sequence from $P \cup \mathit{Eureka} \cup \{C_2\}$ to $P \cup \mathit{Eureka} \cup V_2$ by applying rules R2, R4, R5, and R6. This final phase is further detailed below.

During Phases (4) and (5) we restrict the application of rule R4 to the cases indicated at Point (vi) of Definition 4.1, and during Phase (5) we perform at least one unfolding step.

Notice that, when at the end of Phase (5) we have derived the set *Eureka*, there is a transformation sequence $T_1 : \langle P \cup \mathit{Eureka}, P \cup \mathit{Eureka} \cup \{C_1\}, \ldots, P \cup \mathit{Eureka} \cup S_1\rangle$ and also a transformation sequence $T_2 : \langle P \cup \mathit{Eureka}, P \cup \mathit{Eureka} \cup \{C_2\}, \ldots, P \cup \mathit{Eureka} \cup S_2\rangle$. This is due to the fact that, so to speak, we can reverse the transformation steps from S_2 to V_2. Indeed, (i) the unfolding steps using C_2 can be reversed by folding steps using C_2, (ii) the goal replacement rule R4 is a self-inverse (see the definition of R4 in Section 2), and (iii) the generalization + equality introduction rule R5 is the inverse of the simplification of equality rule R6, and vice versa.

The total correctness of the transformation sequence T_1 easily follows from the total correctness of the transformation sequence constructed in Phase (2), and the total correctness of the transformation sequence T_2 follows from that fact that its construction complies with the restrictions (α) and (β) of Theorem 3.1. Thus, T_1 and T_2 constitute an unfold/fold proof of the equivalence formula *Equiv*2 using $P \cup \mathit{Eureka}$ and the following theorem follows from Theorem 3.1.

Theorem 5.1. [Soundness of the synthesis method] *Let P be a program, $F(\overline{X}, \overline{Y})$ and $H(\overline{X}, \overline{Z})$ be goals and newp be a predicate symbol not occurring in P. Let Eureka be the set of clauses defining newp derived by the synthesis method from implicit definitions starting from the formula $\forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, (H(\overline{X}, \overline{Z}), newp(\overline{X}, \overline{Z})))$. Then $M(P \cup \mathit{Eureka}) \models \forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, (H(\overline{X}, \overline{Z}), newp(\overline{X}, \overline{Z})))$.*

Now we illustrate the five phase synthesis method from implicit definitions that we have described, by continuing our Example 5.1.

Example 5.2. [Specializing List Concatenation with Type Checks, Continued] The set *Eureka* of clauses for $conc(Xs, Ys, Zs)$, which is the specialized version of $concat(Xs, Ys, Zs)$, is generated according to the following five phases.

Phase (1). We introduce the two clauses:

C_1: $new1(Xs, Ys, Zs) \leftarrow list(Xs),\ list(Ys),\ list(Zs),\ concat(Xs, Ys, Zs)$
C_2: $new2(Xs, Ys, Zs) \leftarrow list(Xs),\ list(Ys),\ list(Zs),\ conc(Xs, Ys, Zs)$

Phase (2). We first unfold clause C_1 w.r.t. $concat(Xs, Ys, Zs)$, we then perform some unfolding steps w.r.t. *list* atoms, we delete duplicate atoms (this is an instance of goal replacement), and we finally perform one folding step. Thus, we get the following set of clauses:

S_1: $new1([\,], Ys, Ys) \leftarrow list(Ys)$
$new1([X|Xs], Ys, [X|Zs]) \leftarrow new1(Xs, Ys, Zs)$

Phase (3). We replace the predicate symbol *new*1 by *new*2 and we get the following set of clauses:

S_2: $new2([\,], Ys, Ys) \leftarrow list(Ys)$
$new2([X|Xs], Ys, [X|Zs]) \leftarrow new2(Xs, Ys, Zs)$

Phase (4). We perform an unfolding step using clause C_2 and we get the set V_2 made out of the following two clauses:

D_1: $new2([\,], Ys, Ys) \leftarrow list(Ys)$
D_2: $new2([X|Xs], Ys, [X|Zs]) \leftarrow list(Xs),\ list(Ys),\ list(Zs),\ conc(Xs, Ys, Zs)$

Phase (5). The generation of the set *Eureka* of clauses which allows us to derive the program $LConcat \cup Eureka \cup V_2$ from the program $LConcat \cup Eureka \cup \{C_2\}$ is shown in Example 5.3. □

Phase (5) of our synthesis method is the most complex phase of all, and indeed, no algorithm exists for the generation of the set *Eureka* in all cases. We will now present a strategy which is successful in our specialization problem (see Example 5.3 below) and also in many other cases (see, for instance, Examples 6.1 and 7.1).

Our strategy for Phase (5) consists of the following three steps (see Fig. 5.2).

Step 5.1. (*Instantiation*) Let us assume that V_2 is the set $\{D_i \mid i = 1, \ldots, n\}$, where for $i = 1, \ldots, n$, the clause D_i is of the form: $new2(\bar{t}_i) \leftarrow Body_i$.

In this step we construct a *multiset* $S_E = \{E_i \mid i = 1, \ldots, n\}$ of instances of clause C_2 such that the heads of the clauses in S_E are equal (modulo variable renaming) to the heads of the clauses in V_2.

This construction is performed by applying to C_2, for $i = 1, \ldots, n$, the substitution $\{\overline{X}/\bar{t}_i\}$ (possibly with identity bindings), thereby obtaining:

E_i: $new2(\bar{t}_i) \leftarrow H(\bar{t}_i, \overline{Z}),\ newp(\bar{t}_i, \overline{Z})$

where we assume that the following two constraints are satisfied: (1) the variables affected by non-identity bindings occur in $newp(\overline{X}, \overline{Z})$, and (2) $vars(\bar{t}_i) \cap vars(\overline{Z}) = \{\,\}$.

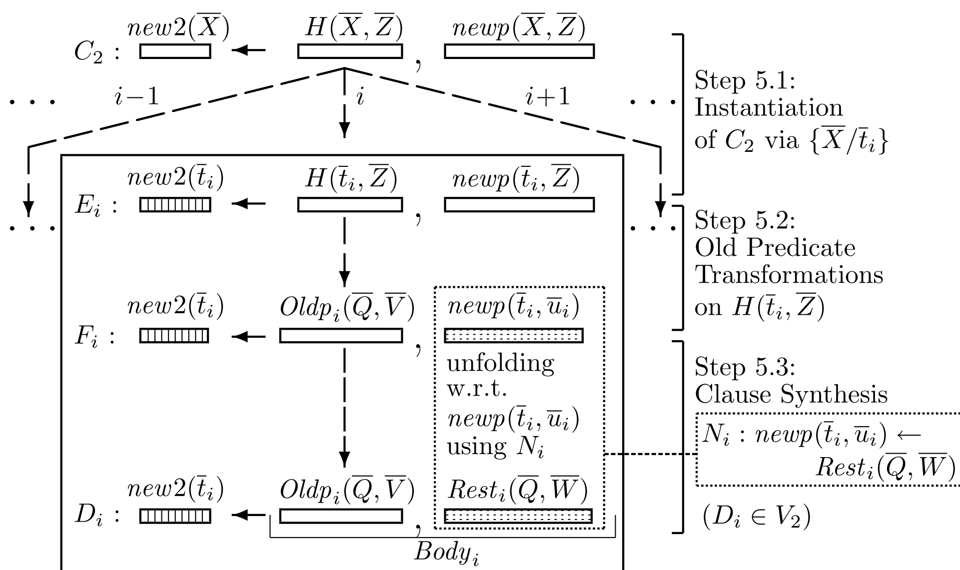

FIGURE 5.2. Steps 5.1, 5.2, and 5.3 of the strategy for the synthesis of the set *Eureka* of clauses.

This first step is motivated by the fact that in order to derive the clauses of V_2 from C_2, we should eventually derive clauses whose heads are equal to the heads of the clauses in V_2. This instantiation may provide a guidance for further transformation steps. Notice that, unlike [8], instantiation is not among the transformation rules we have considered. However, we show below that the form of instantiation we require here, may be viewed as an unfolding step using the clauses in the set *Eureka*.

Step 5.2. (*Old Predicate Transformations*) For $i = 1, \ldots, n$, starting from clause E_i, we repeatedly apply transformation rules taken from {R2, R4, R5, R6} (with the restrictions mentioned at Point (vi) of Definition 4.1) whose only effect on the atom with predicate *newp* may be an instantiation. Moreover, unfolding steps are allowed only if they produce exactly one clause, whose head is equal (modulo variable renaming) to the one of E_i. Notice that by complying with these restrictions, the body of each derived clause contains exactly one occurrence of the predicate *newp*. We stop this transformation process which started from E_i, when we get to a clause of the form:

F_i: $new2(\bar{t}_i) \leftarrow Oldp_i(\overline{Q},\overline{V}),\ newp(\bar{t}_i,\overline{u}_i)$

where: (i) $\overline{Q}$ is the vector of the variables occurring in $newp(\bar{t}_i,\overline{u}_i)$, (ii) $\overline{V}$ is the vector of the variables of F_i not occurring in $\overline{Q}$, and (iii) D_i is equal to a clause of the form: $new2(\bar{t}_i) \leftarrow Oldp_i(\overline{Q},\overline{V}),\ Rest_i(\overline{Q},\overline{W})$, where $\overline{W}$ is a vector of variables which do not belong to $vars(\overline{Q},\overline{V})$. By suitable variable renamings, we may also assume that $vars(\overline{Z}) \cap vars(\overline{Q},\overline{V},\overline{W}) = \{\,\}$.

Step 5.3. (*Clause Synthesis*) For each clause F_i derived at Step 5.2 we consider the

clause:
N_i: $newp(\overline{t}_i, \overline{u}_i) \leftarrow Rest_i(\overline{Q}, \overline{W})$
and we take *Eureka* to be the set $\{N_i \mid i = 1, \ldots, n\}$.

Now we can prove that there exists a transformation sequence from $P \cup \{N_i \mid i = 1, \ldots, n\} \cup \{C_2\}$ to $P \cup \{N_i \mid i = 1, \ldots, n\} \cup V_2$. The proof which we now give, is based on the fact that $\overline{Z}$, $\overline{Q}$, $\overline{V}$, and $\overline{W}$ are pairwise disjoint vectors of variables.
(i) We unfold C_2 w.r.t. $newp(\overline{X}, \overline{Z})$ using the clauses $\{N_i \mid i = 1, \ldots, n\}$ synthesized at Step 5.3, and we derive a set of clauses of the form $\{new2(\overline{t}_i) \leftarrow H(\overline{t}_i, \overline{u}_i),\ Rest_i(\overline{Q}, \overline{W}) \mid i = 1, \ldots, n\}$ (whose heads are equal to the ones of the clauses in the multiset S_E derived at Step 5.1).
(ii) We apply the generalization + equality introduction rule and we derive a set of clauses of the form $\{new2(\overline{t}_i) \leftarrow H(\overline{t}_i, \overline{Z}),\ \overline{Z} = \overline{u}_i,\ Rest_i(\overline{Q}, \overline{W}) \mid i = 1, \ldots, n\}$.
(iii) We perform the old predicate transformations corresponding to those performed at Step 5.2, and we derive the set of clauses $\{new2(\overline{t}_i) \leftarrow Oldp_i(\overline{Q}, \overline{V}), \overline{u}_i = \overline{u}_i, Rest_i(\overline{Q}, \overline{W}) \mid i = 1, \ldots, n\}$. Notice that for $i = 1, \ldots, n$, these transformations determine the instantiation of $\overline{Z}$ to $\overline{u}_i$, because $newp(\overline{t}_i, \overline{Z})$ in E_i becomes $newp(\overline{t}_i, \overline{u}_i)$ in F_i.
(iv) We get V_2 by simplification of equality.

In the following example we illustrate the Steps 5.1, 5.2, and 5.3 for performing Phase 5 of Example 5.2.

Example 5.3. [Specializing List Concatenation with Type Checks, Continued]
Step 5.1. By instantiation, from C_2 of Example 5.2, we get:

E_1: $new2([\,], Ys, Ys) \leftarrow list([\,]),\ list(Ys),\ list(Ys),\ conc([\,], Ys, Ys)$
E_2: $new2([X|Xs], Ys, [X|Zs]) \leftarrow list([X|Xs]),\ list(Ys),\ list([X|Zs]),$
$\quad conc([X|Xs], Ys, [X|Zs])$

The heads of clauses E_1 and E_2 are equal to those of clauses D_1 and D_2 in V_2 of Example 5.2, respectively.

Step 5.2. By unfolding clauses E_1 and E_2 w.r.t. *list* atoms and deleting duplicate atoms we get the two clauses:

F_1: $new2([\,], Ys, Ys) \leftarrow list(Ys),\ conc([\,], Ys, Ys)$
F_2: $new2([X|Xs], Ys, [X|Zs]) \leftarrow list(Xs),\ list(Ys),\ list(Zs),$
$\quad conc([X|Xs], Ys, [X|Zs])$

With reference to Fig. 5.2 we have that (see also clauses D_1 and D_2 of Phase (4) of Example 5.2):

$Oldp_1(Ys)$ is $list(Ys)$,
$Rest_1(Ys)$ is the empty goal *true*,
$Oldp_2(X, Xs, Ys, Zs)$ is $(list(Xs),\ list(Ys),\ list(Zs))$,
$Rest_2(X, Xs, Ys, Zs)$ is $conc(Xs, Ys, Zs)$.

Step 5.3. The set *Eureka* consists the following two clauses:

N_1: $conc([\,], Ys, Ys) \leftarrow$
N_2: $conc([X|Xs], Ys, [X|Zs]) \leftarrow conc(Xs, Ys, Zs)$

When we use the clauses for *conc*, instead of those for *concat*, no list type check is performed. Indeed, those checks are not necessary if we know in advance that the arguments of *concat* are lists. □

Before closing the section let us remark that our specialization technique based on the synthesis method from implicit definitions is an extension of *partial evaluation* [21]. Indeed, partial evaluation corresponds to the case where program specialization is applied using a predicate $input(X)$ of the form: $X = t$, for some (possibly non-ground) term t.

6. SYNTHESIS OF PROGRAMS THAT USE DIFFERENCE-LISTS

Difference-lists are data structures which are sometimes used, instead of lists, for implementing algorithms that manipulate sequences of elements. The advantage of using difference-lists is that the concatenation Z of two sequences X and Y, represented as difference-lists, can be performed in constant time, while it takes linear time (w.r.t. the length of X) if we use the standard predicate for list concatenation, which in this section we denote by the basic predicate $app(X, Y, Z)$.

A difference-list can be thought of as a pair of lists, denoted by $L\backslash R$, such that there exists a third list Y for which $app(Y, R, L)$ holds [9]. In that case we say that Y is *represented* by the difference-list $L\backslash R$. Obviously, a single list can be represented by many difference-lists.

Programs that use lists are often simpler to write and understand than the equivalent ones which make use of difference-lists. Thus, one may be interested in providing techniques for transforming in an automatic way programs which use lists, into programs which use difference-lists. Several such techniques have been proposed in the literature [15, 31].

We will show that by applying our program synthesis method we can automatically perform the transformation which introduces difference-lists. Our method is very general and it can be used also to perform other changes of data representations.

The problem of transforming programs which use difference-lists, instead of lists, can be formulated as follows. Let $p(X, Y)$ be a predicate defined in a program P where Y is a list. We want to synthesize a new predicate, say $\mathit{diff_p}(X, L\backslash R)$, where $L\backslash R$ is a difference-list, together with an additional set of clauses, say *Eureka*, defining $\mathit{diff_p}$. We also want $\mathit{diff_p}(X, L\backslash R)$ to be equivalent to $p(X, Y)$ when $L\backslash R$ is a difference-list representing Y.

Thus, our program transformation problem reduces to the problem of looking for a set *Eureka* of clauses such that:

$$M(P \cup \mathit{Eureka}) \models \forall X, Y\, (p(X, Y) \leftrightarrow \exists L, R\, (app(Y, R, L), \mathit{diff_p}(X, L\backslash R)))$$

where, as already mentioned, $app(Y, R, L)$ holds iff $L\backslash R$ is a difference-list representing Y.

In the following example which we take from [27, page 297], we show how our synthesis method may derive a program which uses difference-lists from an initial program which uses lists.

Example 6.1. [Implementing Queues by Difference-lists] Let us consider the following program *Queue* defining a predicate $queue(S)$ which holds iff S is a sequence of *enqueue* and *dequeue* operations represented as a list of terms of the form $enqueue(X)$ and $dequeue(X)$, respectively.

$queue(S) \leftarrow q(S, [\,])$
$q([enqueue(X)|Xs], Q) \leftarrow app(Q, [X], Q1),\ q(Xs, Q1)$
$q([dequeue(X)|Xs], Q) \leftarrow app([X], Q1, Q),\ q(Xs, Q1)$
$q([\,], Q) \leftarrow Q = [\,]$

Queues of elements are represented as lists. The second argument of the predicate q is a queue which is initially empty (represented as $[\,]$), and it is updated according to the sequence of $enqueue(X)$ and $dequeue(X)$ operations specified by the value of the first argument of q. The $enqueue(X)$ and $dequeue(X)$ operations are implemented by means of list concatenations using $app(Q, [X], Q1)$ and $app([X], Q1, Q)$, respectively, that is, elements enter a queue from the 'right end' and exit a queue from the 'left end'. Since the evaluation of $app(Q, [X], Q1)$ is expensive, we would like to represent the lists Q and $Q1$, which occur in the second argument of q, as difference-lists. Thus, we look for a predicate $diff_q(S, L \backslash R)$ defined by a set *Eureka* of clauses such that:

$$Equiv3: \quad M(Queue \cup Eureka) \models \forall S, Q\, (q(S, Q) \leftrightarrow \exists L, R\, (app(Q, R, L),\ diff_q(S, L \backslash R)))$$

The predicates *app*, '=', and '≠' are considered to be basic predicates (this hypothesis allows us to apply Theorem 5.1 to prove the soundness of our synthesis).

The synthesis of *Eureka* can be performed by routine application of our five phase synthesis method as follows.

Phase (1). We introduce two clauses:

C_1: $new1(S, Q) \leftarrow q(S, Q)$

C_2: $new2(S, Q) \leftarrow app(Q, R, L),\ diff_q(S, L \backslash R)$

Phase (2). We unfold clause C_2 w.r.t. $q(S, Q)$. We then perform some folding steps and we get the following set of clauses:

S_1: $new1([enqueue(X)|Xs], Q) \leftarrow app(Q, [X], Q1),\ new1(Xs, Q1)$
$new1([dequeue(X)|Xs], Q) \leftarrow app([X], Q1, Q),\ new1(Xs, Q1)$
$new1([\,], Q) \leftarrow Q = [\,]$

As one may expect by looking at clause C_1, the clauses for $new1$ are equal to those for q where each occurrence of q has been replaced by $new1$.

Phase (3). By replacing the predicate symbol $new1$ by $new2$ we get the following set of clauses:

S_2: $new2([enqueue(X)|Xs], Q) \leftarrow app(Q, [X], Q1),\ new2(Xs, Q1)$
$new2([dequeue(X)|Xs], Q) \leftarrow app([X], Q1, Q),\ new2(Xs, Q1)$
$new2([\,], Q) \leftarrow Q = [\,]$

Phase (4). By unfolding using clause C_2, by performing goal replacement steps which are justified by properties of the basic predicates (see below), and finally, by applying the generalization + equality introduction rule, we get the set V_2 consisting of the following three clauses:

D_1: $new2([enqueue(X)|Xs], Q) \leftarrow app(Q, R, L),\ R = [X|R1],\ \mathit{diff_q}(Xs, L\backslash R1)$

D_2: $new2([dequeue(X)|Xs], Q) \leftarrow app(Q, R, L),\ L = [X|L1],\ R \neq [X|L1],\ \mathit{diff_q}(Xs, L1\backslash R)$

D_3: $new2([\,], Q) \leftarrow app(Q, R, L),\ R = L$

We have used the following three simple properties of *app* for deriving D_1, D_2, and D_3, respectively:

$\forall Q, X, R1, L\ (\exists Q1\, (app(Q, [X], Q1),\ app(Q1, R1, L)) \leftrightarrow app(Q, [X|R1], L))$

$\forall Q, X, R, L1\ (\exists Q1\, (app([X], Q1, Q),\ app(Q1, R, L1)) \leftrightarrow (app(Q, R, [X|L1]),\ R \neq [X|L1]))$

$\forall Q\ (Q = [\,] \leftrightarrow \exists L, R\ (app(Q, L, R),\ R = L))$

Phase (5). The set *Eureka* can now be synthesized according to the following three steps:

Step 5.1 (*Instantiation*) By instantiation and variable renaming from C_2 we get:

E_1: $new2([enqueue(X)|Xs], Q) \leftarrow app(Q, R, L),\ \mathit{diff_q}([enqueue(X)|Xs], L\backslash R)$
E_2: $new2([dequeue(X)|Xs], Q) \leftarrow app(Q, R, L),\ \mathit{diff_q}([dequeue(X)|Xs], L\backslash R)$
E_3: $new2([\,], Q) \leftarrow app(Q, R, L),\ \mathit{diff_q}([\,], L\backslash R)$

Step 5.2 (*Old Predicate Transformations*) No transformation is applied because, with reference to Fig. 5.2, we have that for $i = 1, 2, 3$, F_i is E_i, and
$Oldp_1(X, Xs, L, R, Q)$ is $app(Q, R, L)$,
$Rest_1(X, Xs, L, R, R1)$ is $(R = [X|R1],\ \mathit{diff_q}(Xs, L\backslash R1))$,
$Oldp_2(X, Xs, L, R, Q)$ is $app(Q, R, L)$,
$Rest_2(X, Xs, L, R, L1)$ is $(L = [X|L1],\ R \neq [X|L1],\ \mathit{diff_q}(Xs, L1\backslash R))$,
$Oldp_3(L, R, Q)$ is $app(Q, R, L)$,
$Rest_3(L, R)$ is $R = L$.

Step 5.3 (*Clause Synthesis*) The set *Eureka* consists of the clauses:

$\mathit{diff_q}([enqueue(X)|Xs], L\backslash R) \leftarrow R = [X|R1],\ \mathit{diff_q}(Xs, L\backslash R1)$
$\mathit{diff_q}([dequeue(X)|Xs], L\backslash R) \leftarrow L = [X|L1],\ R \neq [X|L1],\ \mathit{diff_q}(Xs, L1\backslash R)$
$\mathit{diff_q}([\,], L\backslash R) \leftarrow R = L$

By simplification of the equalities, we get:

$\mathit{diff_q}([enqueue(X)|Xs], L\backslash[X|R1]) \leftarrow \mathit{diff_q}(Xs, L\backslash R1)$
$\mathit{diff_q}([dequeue(X)|Xs], [X|L1]\backslash R) \leftarrow R \neq [X|L1],\ \mathit{diff_q}(Xs, L1\backslash R)$
$\mathit{diff_q}([\,], L\backslash L) \leftarrow$

Notice that, by *Equiv*3, for $Q = [\,]$ the following property holds:

$M(Queue \cup Eureka) \models \forall S\ (q(S, [\,]) \leftrightarrow \exists L\ \mathit{diff_q}(S, L\backslash L))$

Thus, we can express the predicate *queue* in terms of *diff_q* as follows:

$queue(S) \leftarrow \mathit{diff_q}(S, L\backslash L)$

This clause, together with the clauses of the set *Eureka* (see Step 5.3), is analogous to the program given in [27], except for the inequality occurring in the body of one of our final clauses. As the reader may verify, that inequality is necessary for establishing *Equiv*3. □

7. A TRANSFORMATION STRATEGY FOR AVOIDING UNNECESSARY NONDETERMINISM

In this section we present, through an example, a new transformation strategy for improving program efficiency by avoiding unnecessary nondeterminism. For some steps of our strategy we make use of the method of program synthesis from implicit definitions described in Section 4.

A well known technique for avoiding nondeterminism is based on *clause fusion* [10]. It consists in replacing two clauses of the form:

$H \leftarrow I, F$
$H \leftarrow I, G$

by the clauses

$H \leftarrow I, B$
$B \leftarrow F$
$B \leftarrow G$

where B is an atom with a new predicate symbol, say *newp*. Thus, clause fusion can be viewed as a definition step, for introducing the predicate *newp*, followed by a folding step. By this transformation we factorize the goal I which is common to the two initial clauses for H and we avoid the repeated evaluation of this atom in case of backtracking.

However, computations which are common to several clauses are not always apparent in the syntactic structure of the clauses. In fact, we may have clauses of the form:

$H \leftarrow Body1$
$H \leftarrow Body2$

where *Body*1 and *Body*2 do not syntactically include any common subgoal and yet during their evaluation, they produce redundant computations. Indeed, this is the case when there exist three goals, say I, F, and G, such that the following equivalence formulas hold in the least Herbrand model of the program at hand:

*Equiv*4: $\forall \overline{X_1}\, (\exists \overline{Y_1}\, Body1 \leftrightarrow \exists \overline{Z_1}\, (I, F))$

*Equiv*5: $\forall \overline{X_2}\, (\exists \overline{Y_2}\, Body2 \leftrightarrow \exists \overline{Z_2}\, (I, G))$

In *Equiv*4, $\overline{X_1}$ denotes the vector of the variables in $vars(H) \cap vars(Body1)$, $\overline{Y_1}$ denotes the vector of variables occurring in *Body*1 and not occurring in $\overline{X_1}$, and $\overline{Z_1}$ denotes the vector of variables occurring in (I, F) and not in $\overline{X_1}$. Similarly in *Equiv*5, $\overline{X_2}$ denotes the vector of the variables in $vars(H) \cap vars(Body2)$, $\overline{Y_2}$ denotes the vector of variables occurring in *Body*2 and not occurring in $\overline{X_2}$, and $\overline{Z_2}$ denotes the vector of variables occurring in (I, G) and not in $\overline{X_2}$.

In this case, in order to avoid unnecessary nondeterminism, we may replace *Body*1 by (I, F) and *Body*2 by (I, G) and then we apply clause fusion as described above. For these transformations to be totally correct transformations it is required that these two goal replacements are non-ascending (see Theorem 2.1).

The proposal of an automatic method for finding such goals I, F, and G, is beyond the scope of the present paper. However, we may use a strategy which will be applied in Example 7.1 below. The first step of this strategy is a preliminary analysis of the resolution steps starting from the goal H, and this analysis may suggest us a suitable choice for the goal I. Then, in order to construct the goals

F and G such that *Equiv*4 and *Equiv*5 hold for the chosen goal I, we may apply our synthesis method from implicit definitions. For instance, in order to construct F we may introduce a new predicate, say f, which is implicitly defined by the fact that the following equivalence formula holds in the least Herbrand model of the program at hand:

*Equiv*6: $\forall \overline{X}_1\, (\exists \overline{Y}_1\, Body1 \leftrightarrow \exists \overline{Z}_1\, (I, f(\overline{X}_1, \overline{Z}_1)))$

where $\overline{Z}_1$ is the vector of the variables occurring in I and not in $\overline{X}_1$.

If we are able to construct a set *EurekaF* of clauses defining f, by using our synthesis method, then we may stipulate that the goal F is $f(\overline{X}_1, \overline{Z}_1)$. Analogously, we can apply our synthesis method for constructing the goal G, thereby deriving the set *EurekaG* of clauses which define a new predicate, say g.

Our strategy will then continue by transforming the sets *EurekaF* and *EurekaG* of clauses defining f and g, because they, in turn, may still contain some unnecessary nondeterminism.

The reader should notice that the addition to the program at hand of the clauses *EurekaF* and *EurekaG* may not be possible by rule R1 which does not allow for the introduction of recursive definitions. However, we can use Theorem 2.1 for showing the total correctness of a suitable transformation sequence constructed by applying our strategy for avoiding nondeterminism. Indeed, given the initial program $P_0^+ = P_0 \cup EurekaF \cup EurekaG$, Theorem 2.1 ensures that the transformation sequence from program $P_0^+ \cup \{H \leftarrow Body1\} \cup \{H \leftarrow Body2\}$ to the new program $P_0^+ \cup \{H \leftarrow I,\, F\} \cup \{H \leftarrow I,\, G\}$ is totally correct if it is due to non-ascending goal replacements.

In the following example we verify that the suitable goal replacements are non-ascending by constructing non-ascending unfold/fold proofs and using Theorem 4.1.

Example 7.1. [Reachability in a graph] Let us consider the following program P_0 which defines the reachability relation in a directed graph:

1. $reach(X) \leftarrow initial(X)$
2. $reach(X) \leftarrow reach(Y),\ edge(Y, X)$

together with some suitable clauses defining the *initial* and *edge* predicates, which we do not show here.

Program P_0 may perform some redundant computations when evaluating the goal $reach(X)$, where X is an unbound variable, according to the standard left-to-right, depth-first Prolog strategy. Indeed, the computed answer substitutions for the goal $reach(X)$ are obtained by evaluating goals of the form $(initial(X_0), edge(X_0, X_1), edge(X_1, X_2), \ldots, edge(X_n, X))$ for increasing values of n. However, the partial results obtained during the evaluation of 'shorter' goals, are not taken into account for the evaluation of 'longer' goals, and equal subgoals may be repeatedly evaluated along different branches of the SLD-tree (for whose definition we refer to [20]).

In order to discover redundant computations of this kind, we may symbolically generate and examine the set of the SLD-derivations starting from a given goal. (Symbolic evaluation is a standard analysis technique used in various program transformation methods [6, 8, 30].) In particular, by constructing a finite upper portion of the SLD-tree starting from the goal of interest, we may find equal subgoals which have to be evaluated along distinct branches of that SLD-tree, and thus, they produce redundant computations. The description of general analysis techniques which can be used for this search, is beyond the scope of this paper.

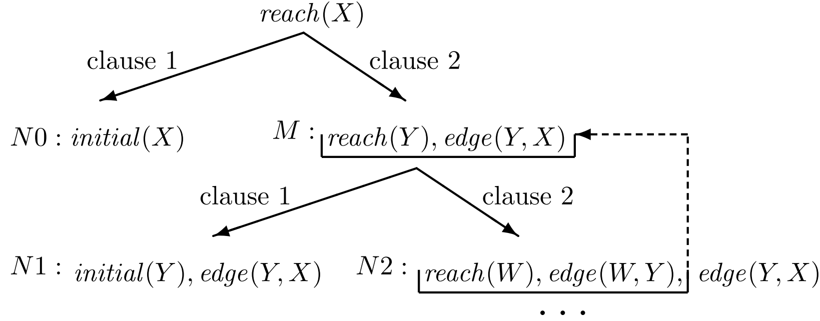

FIGURE 7.1. An upper portion of the SLD-tree for $\mathit{reach}(X)$.

In our case, in order to avoid unnecessary computations and reduce nondeterminism, we may apply the strategy we have described earlier in this section. It consists in looking for some goals I, F, and G, such that clauses 1 and 2 can be rewritten as:

$1'$. $\mathit{reach}(X) \leftarrow I,\, F$
$2'$. $\mathit{reach}(X) \leftarrow I,\, G$

We start off by considering the upper portion of the SLD-tree with root goal $\mathit{reach}(X)$ depicted in Fig. 7.1.

By analyzing that SLD-tree, we discover that for all successful SLD-derivations starting from $\mathit{reach}(X)$ we will get to a goal of the from $\mathit{initial}(Z), E$ where E is a goal whose definition may depend on the SLD-derivation. In particular, (i) the SLD-derivation which uses clause 1 in the first step, contains an occurrence of the goal $\mathit{initial}(Z)$ with $Z = X$ (see goal $N0$), and (ii) all successful SLD-derivations which use clause 2 in the first step, contain a goal of the form: $(\mathit{initial}(Z),\ G(Z, X))$ where Z is a variable distinct from X and $G(Z, X)$ is a goal whose definition depends on the SLD-derivation. Property of Point (ii) derives from the fact that a leftmost subgoal of the goal $N2$ is an instance of the goal in M (see the dashed arrow in Fig. 7.1).

As a result of this analysis, we may conclude that (i) the body of clause 1 is equivalent to the goal $(\mathit{initial}(Z), Z\!=\!X)$ and (ii) the body of clause 2 is equivalent to a goal of the form $(\mathit{initial}(Z),\ G(Z, X))$ for a suitable goal $G(Z, X)$.

Thus, in clauses $1'$ and $2'$ we can choose the goal I to be $\mathit{initial}(Z)$, the goal F to be $Z\!=\!X$, and the goal G to be $g(Z, X)$, where $g(Z, X)$ is a new predicate implicitly defined by:

$$\mathit{Equiv}7\colon\quad M(P_0 \cup \mathit{EurekaG}) \models \forall X\, (\exists Y\, (\mathit{reach}(Y),\ \mathit{edge}(Y, X)) \leftrightarrow \exists Z\, (\mathit{initial}(Z),\ g(Z, X)))$$

The suitable set $\mathit{EurekaG}$ of clauses defining $g(Z, X)$ can be generated by using, as we will indicate below, our five phase synthesis method.

When we have the set $\mathit{EurekaG}$, the transformation continues by adding this set of clauses to P_0 and replacing clauses 1 and 2 (by using the generalization + equality introduction and goal replacement rules, respectively) by the following two clauses:

1*. $reach(X) \leftarrow initial(Z),\ Z = X$
2*. $reach(X) \leftarrow initial(Z),\ g(Z, X)$

Here are the five phase synthesis method for constructing the set *EurekaG*.

Phase (1). We introduce two clauses:

C_1: $new1(X) \leftarrow reach(Y),\ edge(Y, X)$
C_2: $new2(X) \leftarrow initial(Z),\ g(Z, X)$

Phase (2). By unfolding clause C_1 w.r.t. $reach(Y)$ we get the set R_1 consisting of the following two clauses:

R_1: $new1(X) \leftarrow initial(Y),\ edge(Y, X)$
$new1(X) \leftarrow initial(Y1),\ edge(Y1, Y),\ edge(Y, X)$

and then by folding we derive the set S_1 of clauses:

S_1: $new1(X) \leftarrow initial(Y),\ edge(Y, X)$
$new1(X) \leftarrow new1(Y),\ edge(Y, X)$

Phase (3). By replacing in S_1 the predicate symbol $new1$ by $new2$ we get the following set S_2 of clauses:

S_2: $new2(X) \leftarrow initial(Y), edge(Y, X)$
$new2(X) \leftarrow new2(Y), edge(Y, X)$

Phase (4). By unfolding the second clause in S_2 using clause C_2 and by variable renaming, we get the set V_2 consisting of the two clauses:

D_1: $new2(X) \leftarrow initial(Z),\ edge(Z, X)$
D_2: $new2(X) \leftarrow initial(Z),\ g(Z, Y),\ edge(Y, X)$

Phase (5). The set *EurekaG* can now be synthesized according to the following three steps.

Step 5.1. (*Instantiation*) By instantiation from C_2 we get two copies of the same clause:

E_1: $new2(X) \leftarrow initial(Z),\ g(Z, X)$
E_2: $new2(X) \leftarrow initial(Z),\ g(Z, X)$

Step 5.2. (*Old Predicate Transformations*) We do not apply any transformation rule to E_1 or E_2 because the conditions stated in Step 5.2 of Section 5 for stopping the transformation process are already satisfied. Indeed, with reference to Fig. 5.2 we have that:

F_1 is E_1, F_2 is E_2, and
$Oldp_1(Z, X)$ is $initial(Z)$, $Rest_1(Z, X)$ is $edge(Z, X)$,
$Oldp_2(Z, X)$ is $initial(Z)$, $Rest_2(Z, Y, X)$ is $(g(Z, Y),\ edge(Y, X))$.

Step 5.3. (*Clause Synthesis*) The set *EurekaG* consists of the following clauses:

3. $g(Z, X) \leftarrow edge(Z, X)$
4. $g(Z, X) \leftarrow g(Z, Y), edge(Y, X)$

Thus, the derived program version P_1 is:

1*. $reach(X) \leftarrow initial(Z),\ Z = X$
2*. $reach(X) \leftarrow initial(Z),\ g(Z, X)$
3. $g(Z, X) \leftarrow edge(Z, X)$
4. $g(Z, X) \leftarrow g(Z, Y),\ edge(Y, X)$

together with the clauses defining the predicates *initial* and *edge*. By Theorem 2.1 we have that $M(P_0 \cup EurekaG) = M(P_1 \cup EurekaG)$ because it is the case that the replacement of the body of clause 2 by $(initial(Z),\ g(Z, X))$ is a non-ascending goal replacement. This property can be established by providing a non-ascending unfold/fold proof of *Equiv*7 using $P_0 \cup EurekaG$ and by applying Theorem 4.1. We now show the two transformation sequences T_1 and T_2 which constitute that proof and then we show that the proof is non-ascending.

The initial program of T_1 is $P_0 \cup EurekaG$. By definition introduction we derive $P_0 \cup EurekaG \cup \{C_1\}$, then by unfolding C_1 w.r.t. $reach(Y)$ we get $P_0 \cup EurekaG \cup R_1$, and finally, by folding we get $P_0 \cup EurekaG \cup S_1$.

The initial program of T_2 is $P_0 \cup EurekaG$. By definition introduction we derive $P_0 \cup EurekaG \cup \{C_2\}$, then by unfolding C_2 w.r.t. $g(Z, X)$ we get $P_0 \cup EurekaG \cup \{D_1, D_2\}$, and finally, by folding clause D_2 using C_2 we get $P_0 \cup EurekaG \cup S_2$.

The sequences T_1 and T_2 constitute a non-ascending proof because: (i) every derivation path from C_1 to a clause in S_1 contains precisely one unfolding step, and (ii) for each clause, say E, in S_2 there is a derivation path in T_2 from C_2 to E which contains precisely one unfolding step (for the notion of a non-ascending proof given in Definition 4.1 the number of applications of the other transformation rules is not significant).

Now we may continue our derivation from program P_1 and we may apply the clause fusion technique. Thus, we replace clauses 1* and 2* by the following three clauses (by performing a definition introduction and a folding step):

5. $reach(X) \leftarrow initial(Z),\ b(Z, X)$
6. $b(Z, X) \leftarrow Z = X$
7. $b(Z, X) \leftarrow g(Z, X)$

The current program version, call it P_2, consists of clauses 3, 4, 5, 6, and 7, together with the clauses for *edge* and *initial*. Now our strategy continues by transforming the clauses for the predicate g. We consider an upper portion of the SLD-tree with root-goal $g(Z, X)$ and we perform an analysis similar to the one described above for $reach(X)$. By this analysis we get that the body of clause 4 is equivalent to a goal of the form $(edge(Z, V),\ Q(V, X))$. Thus, we may apply our synthesis method by introducing the new predicate q implicitly defined by the equivalence formula:

*Equiv*8: $M(P_2 \cup EurekaQ) \models$
$$\forall X, Z\ (\exists Y\ (g(Z, Y),\ edge(Y, X) \leftrightarrow \exists V\ (edge(Z, V),\ q(V, X)))$$

where *EurekaQ* is the set of clauses which should be generated.

By one more application of our five phase method which we do not present here, we derive the following *EurekaQ* clauses for the predicate q:

8. $q(Z,X) \leftarrow edge(Z,X)$
9. $q(Z,X) \leftarrow q(Z,Y),\ edge(Y,X)$

Thus, the clauses defining q are equal to those defining g and we may replace q by g in *Equiv*8, and we get:

*Equiv*9: $M(P_2) \models \forall X, Z\, (\exists Y\, (g(Z,Y),\ edge(Y,X) \leftrightarrow \exists V\, (edge(Z,V),\ g(V,X)))$

By a goal replacement justified by *Equiv*9, from clause 4 we derive:

10. $g(Z,X) \leftarrow edge(Z,V),\ g(V,X)$

We may easily verify that also this goal replacement step is non-ascending (by constructing a non-ascending unfold/fold proof of *Equiv*9) and therefore $M(P_2) = M(P_3)$, where P_3 is the program obtained from P_2 by replacing clause 4 by clause 10. Now we may apply the clause fusion technique to clauses 3 and 10. In order to do so, we apply rule R5 to clause 3 and we get:

3′. $g(Z,X) \leftarrow edge(Z,V),\ V = X$

and then we fold clauses 3′ and 10 by using clauses 6 and 7. We thus derive:

11. $g(Z,X) \leftarrow edge(Z,V),\ b(V,X)$

The current program version P_4 is:

5. $reach(X) \leftarrow initial(Z),\ b(Z,X)$
6. $b(Z,X) \leftarrow Z = X$
7. $b(Z,X) \leftarrow g(Z,X)$
11. $g(Z,X) \leftarrow edge(Z,V),\ b(V,X)$

together with the clauses 8 and 9 and the clauses for *edge* and *initial*.

By some final transformation steps by which (i) we unfold the equality in the body of clause 6 (by rule R6), (ii) we unfold clause 7 w.r.t. the predicate g, and (iii) we discard clauses 8 and 9 because the predicate q is not needed, we get the following final program P_5 (apart from the clauses for *edge* and *initial*):

5. $reach(X) \leftarrow initial(Z),\ b(Z,X)$
12. $b(X,X) \leftarrow$
13. $b(Z,X) \leftarrow edge(Z,V),\ b(V,X)$

This program is right-recursive and it computes the set of reachable vertices from the given initial ones in a *forward-chaining* fashion. This means that the evaluation of the goal $reach(X)$ is done by program P_5 in a more deterministic way w.r.t. the initial program P_0 which, instead, is left-recursive and evaluates the goal $reach(X)$ in a *backward-chaining* fashion. □

A similar transformation of left-recursive programs into right-recursive programs was presented in [5] where, however, the transformation of P_0 into P_5 is presented in one ‘big step’ as a schema-based transformation, which is validated by an ad-hoc inductive proof. We believe that our approach based on transformation rules and strategies, is much more flexible than the schema-based approach. Indeed, it is possible to use our approach to perform program derivations analogous to the one we have presented here, even if the initial program is not an instance of a known schema.

8. RELATED WORK AND CONCLUSIONS

The use of unfold/fold rules for the verification of program properties has been often suggested since the early days of program transformation [8]. In this paper we have formalized a method based on unfold/fold transformations, called the unfold/fold proof method, which can be used for proving properties of logic programs w.r.t. the least Herbrand model semantics. Since the unfold/fold transformations may be designed to preserve many different semantics (see, for instance, [24]), one may extend our method to prove properties w.r.t. those semantics as well.

We have provided some conditions which ensure that, when a property is used as a lemma to perform program transformations, these transformations are indeed totally correct, that is, they preserve the least Herbrand model semantics. These conditions rely on the existence of a *non-ascending* unfold/fold proof of the property of interest, and thus, since they refer to finite objects, they have a more constructive nature w.r.t. other techniques, such as those based on *consistency with weight tuple measures* [29] or *non-increasingness* [2], which rely on the verification of properties of possibly infinite sets of SLD-derivations.

We have also presented a method for synthesizing programs from unfold/fold proofs of program properties. Although our method makes use of unfold/fold transformations, we feel that it falls into the category of *synthesis methods* because the initial specification is not in Horn clause form (that is, it is not a logic program), but it is assumed to be a more general formula of the form:

$$\forall \overline{X}\, (\exists \overline{Y}\; F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\; (H(\overline{X}, \overline{Z}),\; newp(\overline{X}, \overline{Z}))) \tag{8.1}$$

where $F(\overline{X}, \overline{Y})$ and $H(\overline{X}, \overline{Z})$ are conjunctions of atoms and *newp* is the predicate for which we would like to synthesize a program. Specifications of the form (8.1) can be considered to be an implicit definition of the new predicate *newp*, and thus, we say that our proposed technique is a *synthesis method from implicit definitions*.

A very large number of synthesis methods have been proposed in the literature. All these methods may vary because of: (i) the form of the initial specification, (ii) the rules used for deriving programs from specifications, and (iii) the language used for the synthesized programs (see, for instance, [7, 11, 12, 16] for references in the case of logic programs).

Our synthesis method is related to the methods for logic program synthesis (see, for instance, [16]) where the initial specification is an equivalence formula of the first order predicate calculus and one is allowed to use derivation rules similar to the unfold/fold rules. These methods, called *deductive synthesis* methods in the survey paper [11], allow for initial specifications of the form:

$$\forall X\, (spec(X) \leftrightarrow newp(X))$$

where *newp* is the predicate for which we want to synthesize a program and *spec* is any formula of the first order predicate calculus. Thus, no implicit definitions like those provided by formulas of the form (8.1) above are allowed.

Our synthesis method can also be viewed as a technique for the extraction of a program from an unfold/fold proof. Thus, the basic idea of our method is also related to the *proofs-as-programs* approach [1, 7, 12, 23] whereby the constructive proof of a property can be used for synthesizing a program which satisfies that property. However, between our approach and the proofs-as-programs approach, there are many differences. Among them we recall the differences due to: (i) the

derivation rules considered (in particular, *constructive type theory* is used in [1, 7], untyped first-order logic is used in [23], and *extended execution* is used in [12]), (ii) the languages in which the synthesized programs are written (indeed, the authors of [1, 23] consider applicative languages), and (iii) the form of the specifications. With reference to this last difference, one should notice that the synthesis methods based on the proofs-as-programs approach are used for synthesizing programs from specifications of the form:

$$\forall X\, \exists Y\, spec(X, Y)$$

Thus, in the case of functional programming, this means that a synthesized program corresponds to a total function f such that $\forall X\, spec(X, f(X))$, or equivalently, $\forall X\ \forall Y\ (Y = f(X) \rightarrow spec(X, Y))$. This specification is less general than the implicit definitions considered in this paper. The same holds in the case of logic programming.

The synthesis method we propose also extends the standard techniques which are currently available in the framework of unfold/fold program transformation. An informal argument to support this claim can be given as follows.

By using the unfold/fold rules as defined in [28], a new predicate, say *newp*, can be introduced in terms of already available predicates only in an explicit way, in the sense that one may add to the current program P a clause C of the form:

$$newp(X) \leftarrow F(X, Y)$$

where F is a conjunction of atoms whose predicates occur in P. Thus, in the least Herbrand model of $P \cup \{C\}$ the new predicate *newp* is specified by the formula:

$$\forall X\, (\exists Y\, F(X, Y) \leftrightarrow newp(X))$$

which is a less general formula than the ones we have considered in this paper.

Implicit definitions are also considered in [18], where some modifications of the unfolding and folding rules are introduced to deal with generalized definitions of the form:

$$(H(X),\ newp(X)) \leftarrow F(X, Y)$$

for some goal $H(X)$ and $F(X, Y)$. In our method we do not need to introduce any modified rule. Moreover, in [18] the form of allowed derivations is very restricted, while in our case, the unfold/fold proofs may be of any general form.

The reader may also verify that the program specialization and difference-list transformation examples we have presented cannot be derived in a natural way by using the unfold/fold transformations of [28]. Some modified versions of the rules should be used instead, like, for instance, the unfold/fold rules with *constraints* introduced by [3] for specializing logic programs, or the *inverse definition* and the *data structure mapping* introduced by [31].

Our last example on the avoidance of nondeterminism shows that the unfold/fold transformation technique enhanced with our synthesis method, is able to derive programs for which other methods require the off-line proof of some insightful lemmas (like the schema-based equivalence in [5]). By using our synthesis method, in fact, we produce equivalences which may be used as lemmas during the program derivation itself. Thus, the synthesis method we propose may also be useful to enhance

other unfold/fold-based techniques for avoiding nondeterminism which do not use lemmas (like, for instance, [25]).

We would like to stress the point that the program specialization method presented here as an application of our synthesis method, is strictly more general than the usual partial evaluation methods [21]. Indeed, we are able to specialize our initial program w.r.t. a set of input values which can be described by *any* predicate, while in [21] the set of input values can only be a set of instances of a given tuple of terms.

More formally, by using our method one can solve program specialization problems specified by the formula:

$$\forall X\,((\mathit{input}(X),\, p(X)) \leftrightarrow (\mathit{input}(X),\, \mathit{spec_p}(X)))$$

where $\mathit{input}(X)$ is any predicate and $\mathit{spec_p}(X)$ is the specialized version of $p(X)$ which satisfies $\mathit{input}(X)$ for each X. The methods based on [21] can only solve problems specified by:

$$\forall Y\,(p(t(Y)) \leftrightarrow \mathit{spec_p}(t(Y)))$$

which can be viewed as a special case of the above specification when $X = t(Y)$ and $\mathit{input}(t(Y))$ is true.

A final remark concerns the mechanization of our synthesis method. As it is the case for general purpose synthesis and transformation techniques, suitable strategies need to be devised for dealing with particular classes of program specifications and ensuring the derivation of efficient programs. If one uses our approach various strategies, such as the ones described in [24], are available and one can indeed apply them for guiding the application of the unfold/fold transformation rules.

9. ACKNOWLEDGEMENTS

We would like to thank all members of the community of the Logic Program Synthesis and Transformation (LoPSTr) Workshops and the referees who stimulated us with suggestions and comments. Particular thanks go to A. Bossi, N. Cocco, D. De Schreye, L. Fribourg, and M. Leuschel for illuminating conversations. The occasion of a special issue of the Journal of Logic Programming edited by A. Bossi and Y. Deville, encouraged us to improve the preliminary version of this paper presented at LoPSTr '93. S. Renault implemented a system for unfold/fold transformations which helped us in the program derivations presented in this paper.

REFERENCES

1. J. L. Bates and R. L. Constable. Proofs as programs. *ACM Toplas*, 7(1):113–136, 1985.
2. A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In *Proceedings ALP '94*, Lecture Notes in Computer Science 850, pages 269–286. Springer-Verlag, 1994.
3. A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

4. D. Boulanger and M. Bruynooghe. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, 15:495–521, 1993.
5. D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(1):115–134, 1991.
6. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.
7. A. Bundy, A. Smaill, and G. Wiggins. The synthesis of logic programs from inductive proofs. In J. W. Lloyd, editor, *Computational Logic, Symposium Proceedings, Brussels, November 1990*, pages 135–149, Berlin, 1990. Springer-Verlag.
8. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
9. K. L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proceedings Information Processing '77*, pages 939–944. North-Holland, 1977.
10. S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–229, 1988.
11. Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19, 20:321–350, 1994.
12. L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D. H. D. Warren and P. Szeredi, editors, *Proceedings Seventh International Conference on Logic Programming, Jerusalem, Israel, June 18-20, 1990*, pages 685–699. The MIT Press, 1990.
13. J. P. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In *Proceedings of LoPSTr'92, Manchester, U.K.*, pages 151–167. Springer-Verlag, 1993.
14. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Proceedings Sixth International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, Lecture Notes in Computer Science 844, pages 340–354. Springer-Verlag, 1994.
15. Å. Hansson and S.-Å. Tärnlund. Program transformation by data structure mapping. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 117–122. Academic Press, 1982.
16. C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.
17. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1992.
18. T. Kanamori and M. Maeji. Derivation of logic programs from implicit definition. Technical Report TR-178, ICOT, Tokyo (Japan), 1986.
19. L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.
20. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
21. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
22. M. J. Maher. Correctness of a logic program transformation system. IBM Research Report RC 13496, T. J. Watson Research Center, 1987.

23. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Toplas*, 2:90–121, 1980.

24. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

25. A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 414–427. ACM Press, 1997.

26. M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, *Logic Program Synthesis and Transformation, Proceedings of LoPSTr '93, Louvain-la-Neuve, Belgium*, Workshops in Computing, pages 141–158. Springer-Verlag, 1994.

27. L. S. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1994. Second Edition.

28. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

29. H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

30. V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

31. J. Zhang. An automatic d–list transformation algorithm for Prolog programs. Technical report, Department of Computer Science, University College of Swansea, U.K., 1987.

10. APPENDIX

Proof of Theorem 4.1
Since restriction (α) of Theorem 2.1 is implied by Points (iv) and (v) of Definition 4.1, and restriction (β) of that theorem is implied by Point (vi) of that definition, we have that by Theorem 2.1 the transformation sequences T_1 and T_2 of Definition 4.1 are totally correct. Thus, a non-ascending unfold/fold proof is a particular unfold/fold proof. By Theorem 3.1, we have that $M(P_0) \models \forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X}, \overline{Z}))$.

We have to prove that, for each vector $\bar{t}$ of terms in HU such that $M(P_0) \models \exists \overline{Y}\, F(\bar{t}, \overline{Y})$, we have that: $\mu(F(\bar{t}, \overline{Y})) \geq \mu(G(\bar{t}, \overline{Z}))$. (Notice that by the completeness of SLD-resolution and by the equivalence $M(P_0) \models \forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X}, \overline{Z}))$, for each $\bar{t}$ such that $M(P_0) \models \exists \overline{Y}\, F(\bar{t}, \overline{Y})$, there exist successful SLD-derivations using P_0 of both $F(\bar{t}, \overline{Y})$ and $G(\bar{t}, \overline{Z})$ and thus, $\mu(F(\bar{t}, \overline{Y}))$ and $\mu(G(\bar{t}, \overline{Z}))$ are both defined.)

For reasons of simplicity we present the proof of this fact in the case where no basic predicates occur either in the SLD-derivations of $F(\bar{t}, \overline{Y})$, or in the ones of $G(\bar{t}, \overline{Z})$, or in the unfold/fold proof of $\forall \overline{X}\, (\exists \overline{Y}\, F(\overline{X}, \overline{Y}) \leftrightarrow \exists \overline{Z}\, G(\overline{X}, \overline{Z}))$ (recall also that we assume that '=' is a basic predicate). In particular, rules R4, R5, and R6 are not applied in this unfold/fold proof. The extension of our proof to the general case where also basic predicates may occur, is straightforward, because the function μ does not depend on the SLD-derivation steps which are performed by selecting basic predicates.

Let M_F be the set of vectors $\bar{t}$ of ground terms in HU such that $M(P_0) \models \exists \overline{Y}\, F(\bar{t}, \overline{Y})$. Let us consider the ordering $>$ on the set M_F defined as follows: given any two vectors $\bar{t}$ and $\bar{u}$ of ground terms in M_F we have $\bar{t} > \bar{u}$ iff $\mu(F(\bar{t}, \overline{Y})) > \mu(F(\bar{u}, \overline{Y}))$. The $>$ ordering on M_F is well-founded.

Let us now prove, by complete induction w.r.t. the $>$ ordering, that for each $\bar{t} \in M_F$, $\mu(F(\bar{t}, \overline{Y})) \geq \mu(G(\bar{t}, \overline{Z}))$.

Given any $\bar{t} \in M_F$, we assume by induction hypothesis that for each $\bar{u} \in M_F$, if $\bar{t} > \bar{u}$ then $\mu(F(\bar{u}, \overline{Y})) \geq \mu(G(\bar{u}, \overline{Z}))$ and we have to show that $\mu(F(\bar{t}, \overline{Y})) \geq \mu(G(\bar{t}, \overline{Z}))$.

Let ρ be $\mu(F(\bar{t}, \overline{Y}))$, that is, the length of the shortest successful SLD-derivation of $F(\bar{t}, \overline{Y})$ using P_0 (recall that no basic predicates occur in that SLD-derivation). Let the transformation sequences T_1 and T_2, the clauses C_1 and C_2, and the sets S_1 and S_2 of clauses be defined as in the Definition 4.1 of a non-ascending unfold/fold proof. We can construct:
(i) a successful SLD-derivation of $F(\bar{t}, \overline{Y})$ using P_0 of length ρ of the form: $F_0, F_1, \ldots, F_m, \ldots, \mathit{true}$ (recall that in this paper a goal is a conjunction of atoms and by a successful SLD-derivation we mean a derivation whose last goal is the empty conjunction *true*) and
(ii) a derivation path R_1 (taken from the transformation sequence T_1) from C_1 to a clause L in S_1 of the form:

$R_1:\; E_0 \Rightarrow E_1 \Rightarrow \ldots \Rightarrow E_m \Rightarrow E_{m+1} \Rightarrow \ldots \Rightarrow E_{m+k}$

such that the following conditions hold:
(a.1) $E_0 = C_1 = \mathit{new}1(\overline{X}) \leftarrow F(\overline{X}, \overline{Y})$
(b.1) $E_{m+k} = L$
(c.1) for $i = 1, \ldots, m$, with $m \geq 1$, E_i is derived from E_{i-1} by unfolding, and
(d.1) for $i = m+1, \ldots, m+k$, with $k \geq 0$, E_i is derived from E_{i-1} by folding using C_1.
We also have that for $i = 0, \ldots, m$, the goal F_i is $bd(E_i\theta_i)$, where θ_i is the mgu of $\mathit{new}1(\bar{t})$ and $hd(E_i)$ (in particular, $\theta_0 = \{\overline{X}/\bar{t}\}$ and $F_0 = F(\bar{t}, \overline{Y})$).

By definition of a non-ascending unfold/fold proof there exist a clause M in S_2 and a derivation path R_2 (taken from the transformation sequence T_2) of the form:

$R_2:\; Q_0 \Rightarrow Q_1 \Rightarrow \ldots \Rightarrow Q_n \Rightarrow Q_{n+1} \Rightarrow \ldots \Rightarrow Q_{n+k}$

where:
(a.2) $Q_0 = C_2 = \mathit{new}2(\overline{X}) \leftarrow G(\overline{X}, \overline{Z})$
(b.2) $Q_{n+k} = M$ and M is derived from L by substituting $\mathit{new}2$ for $\mathit{new}1$,
(c.2) for $i = 1, \ldots, n$, with $n \geq 1$, Q_i is derived from Q_{i-1} by unfolding, and
(d.2) for $i = n+1, \ldots, n+k$, Q_i is derived from Q_{i-1} by folding using C_2.

Notice that in R_1 there are as many folding steps as in R_2 because: (1) the number of occurrences of $\mathit{new}1$ in the body of L is the number of folding steps performed in R_1, (2) by Point (b.2) above, and (3) the number of occurrences of $\mathit{new}2$ in the body of M is the number of folding steps performed in R_2. Notice also that $m \geq n$ because of Point (vii) of Definition 4.1.

Let us now consider the sequence of clauses: $Q_0\eta_0, Q_1\eta_1, \ldots, Q_n\eta_n$, where $Q_0, Q_1, \ldots, Q_n$ are the clauses occurring the initial part of the derivation path R_2 and, for $i = 0, \ldots, n$, the substitution η_i is the mgu of $\mathit{new}2(\bar{t})$ and $hd(Q_i)$. We have that $\eta_0 = \theta_0$. We also have that $\eta_n = \theta_m$ because the arguments of $hd(E_m)$ are equal to the ones of $hd(Q_n)$, and this is the case because: (i) $hd(E_m) = hd(L)$ and $hd(Q_n) = hd(M)$ (recall that folding steps only are performed to derive L from E_m and M from Q_n), and (ii) M is derived from L by substituting $\mathit{new}2$ for $\mathit{new}1$.

By construction, Q_n can be obtained from E_m by first folding k times using E_0 (which is C_1), then replacing the occurrences of $new1$ by $new2$, and finally unfolding k times using Q_0 (which is C_2). Thus, $Q_n\eta_n$ can be obtained from $E_m\theta_m$ by replacing k instances, say $F(\overline{t1},\overline{Y1}),\ldots,F(\overline{tk},\overline{Yk})$, of $F(\overline{X},\overline{Y})$ by the corresponding k instances $G(\overline{t1},\overline{Z1})$, $\ldots$, $G(\overline{tk},\overline{Zk})$ of $G(\overline{X},\overline{Z})$. By the definition of the folding rule and, in particular, as a consequence of Condition 2 of R3, we have that for $i=1,\ldots,k$, $\overline{Yi}$ and $\overline{Zi}$ are vectors of variables and each variable in $\overline{Yi}$ does not occur in the clause $E_m\theta_m$ outside the instance $F(\overline{ti},\overline{Yi})$, and, analogously, each variable in $\overline{Zi}$ does not occur in the clause $Q_n\eta_n$ outside the instance $G(\overline{ti},\overline{Zi})$.

Without loss of generality, we may assume that $\overline{t1},\ldots,\overline{tk}$ are vectors of ground terms (if they are not, we may replace them by some other vectors of ground terms without changing the length of the shortest successful SLD-derivation of $bd(E_m\theta_m)$ using P_0). We have that $\bar{t}>\overline{t1},\ldots,\bar{t}>\overline{tk}$ and therefore, by inductive hypothesis, we have that $\mu(F(\overline{t1},\overline{Y1}))\geq\mu(G(\overline{t1},\overline{Z1})),\ldots,\mu(F(\overline{tk},\overline{Yk}))\geq\mu(G(\overline{tk},\overline{Zk}))$.

Now, the function μ satisfies the following property: if A and B are goals such that $M(P_0)\models\exists\overline{U}\,(A,B)$, where $\overline{U}=vars((A,B))$, and $vars(A)\cap vars(B)=\{\,\}$, then $\mu((A,B))=\mu(A)+\mu(B)$.

Since for $i=1,\ldots,k$, $F(\overline{ti},\overline{Yi})$ does not share any variable with other goals in $bd(E_m\theta_m)$, and $G(\overline{ti},\overline{Zi})$ does not share any variable with other goals in $bd(Q_n\eta_n)$, we have that $\mu(bd(E_m\theta_m))\geq\mu(bd(Q_n\eta_n))$ (recall that a successful SLD-derivation of $bd(Q_n\eta_n)$ using P_0 exists because: (1) $M(P_0)\models\exists\overline{V}\,bd(E_m\theta_m)\leftrightarrow\exists\overline{W}\,bd(Q_n\eta_n)$, where $\overline{V}$ and $\overline{W}$ are the variables of $bd(E_m\theta_m)$ and $bd(Q_n\eta_n)$, respectively, (2) by hypothesis, a successful SLD-derivation of $bd(E_m\theta_m)$ (which is F_m) using P_0 exists, and (3) SLD-resolution is complete).

The following sequence of goals is a successful SLD-derivation of $G(\bar{t},\overline{Z})$ using P_0: $bd(Q_0\eta_0),bd(Q_1\eta_1),\ldots,bd(Q_n\eta_n),\ldots,true$, where $bd(Q_0\eta_0)=G(\bar{t},\overline{Z})$ and the SLD-derivation $bd(Q_n\eta_n),\ldots,true$ is the shortest successful SLD-derivation of $bd(Q_n\eta_n)$ using P_0. Since $m\geq n$ we have that:

$\mu(F(\bar{t},\overline{Y}))=m+\mu(bd(E_m\theta_m))\geq n+\mu(bd(Q_n\eta_n))\geq\mu(G(\bar{t},\overline{Z}))$. $\square$