TLP **4** (4): 495–537, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S147106840400198X Printed in the United Kingdom

Transformations of logic programs with goals as arguments

ALBERTO PETTOROSSI
Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy
(*e-mail:* alberto.pettorossi@uniroma2.it)

MAURIZIO PROIETTI
IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
(*e-mail:* proietti@iasi.rm.cnr.it)

Abstract

We consider a simple extension of logic programming where variables may range over goals and goals may be arguments of predicates. In this language we can write logic programs which use goals as data. We give practical evidence that, by exploiting this capability when transforming programs, we can improve program efficiency. We propose a set of program transformation rules which extend the familiar unfolding and folding rules and allow us to manipulate clauses with goals which occur as arguments of predicates. In order to prove the correctness of these transformation rules, we formally define the operational semantics of our extended logic programming language. This semantics is a simple variant of LD-resolution. When suitable conditions are satisfied this semantics agrees with LD-resolution and, thus, the programs written in our extended language can be run by ordinary Prolog systems. Our transformation rules are shown to preserve the operational semantics and termination.

KEYWORDS: program transformation, unfold/fold transformation rules, higher order logic programming, continuations

1 Introduction

Program transformation is a very powerful and widely recognized methodology for deriving programs from specifications. The *rules + strategies* approach to program transformation was advocated in the 1970s by Burstall and Darlington (1977) for developing first order functional programs. Since then Burstall and Darlington's approach has been followed in a variety of language paradigms, including logical languages (Tamaki and Sato 1984) and higher order functional languages (Sands 1996). The distinctive feature of the rules + strategies approach is that it allows us to separate the concern of proving the correctness of programs with respect to specifications from the concern of achieving computational efficiency. Indeed, the correctness of the derived programs is ensured by the use of semantics preserving transformation rules, whereas the computational efficiency is achieved through the use of suitable strategies which guide the application of the rules. The preservation of the semantics is proved once and for all, for some given sets of transformation rules, and if we restrict ourselves to suitable classes of programs, we can also guarantee the effectiveness of the strategies for improving efficiency.

In this paper we will argue through some examples, that a simple extension of logic programming may give extra power to the program transformation methodology based on rules and strategies. This extension consists in allowing the use of variables which range over goals, called *goal variables*, and the use of goals which are arguments of predicates, called *goal arguments*.

In the practice of logic programming the idea of having goal variables and goal arguments is not novel. The reader may look, for instance, at Sterling and Shapiro (1986) and Warren (1982). Goal variables and goal arguments can be used for expressing the meaning of logical connectives and for writing programs in a *continuation passing style* (Tarau and Boyer 1990; Wand 1980) as the following example shows.

Example 1
The following program $P1$:

$$F \vee G \leftarrow F$$
$$F \vee G \leftarrow G$$

expresses the meaning of the *or* connective. The following program $P2$:

$$p([\,], Cont) \leftarrow Cont$$
$$p([X|Xs], Cont) \leftarrow p(Xs, q(X, Cont))$$
$$q(0, Cont) \leftarrow Cont$$

uses the goal variable *Cont* which denotes a continuation. The goal $p(l, true)$ succeeds in $P2$ iff the list l consists of 0's only. □

Programs with goal variables and goal arguments, such as $P1$ and $P2$ in the above example, are not allowed by the usual first order syntax of Horn clauses, where variables cannot occur as atoms and predicate symbols are distinct from function symbols. Nevertheless, these programs can be run by ordinary Prolog systems whose operational semantics is based on LD-resolution, that is, SLD-resolution with the leftmost selection rule. For the concepts of *LD-resolution*, *LD-derivation*, and *LD-tree* the reader may refer to Apt (1997).

The extension of logic programming we consider in this paper, allows us to write programs which use goals as data. This extension turns out to be useful for performing program manipulations which are required during program transformation and are otherwise impossible. For instance, we will see that by using goal variables and goal arguments, we are able to perform goal rearrangements (also called *goal reorderings* in Bossi *et al.* (1996)) which are often required for folding, without affecting program termination and without increasing nondeterminism.

Goal rearrangement is a long standing issue in logic program transformation. Indeed, although the unfold/fold transformation rules by Tamaki and Sato (1984) preserve the least Herbrand model, they may require goal rearrangements and thus, they may not preserve the operational semantics based on LD-resolution. Moreover, goal rearrangements may increase nondeterminism by requiring that predicate calls have to be evaluated before their arguments are sufficiently instantiated, and in many Prolog systems, insufficiently instantiated calls of built-in predicates may cause errors at run-time. In Bossi and Cocco (1994) it has been proved that by ruling out goal rearrangements, if some suitable conditions hold, then the unfolding, folding, and goal replacement transformation rules preserve the operational semantics of logic programs based on LD-resolution and, in particular, these rules preserve *universal termination*, i.e. the finiteness of all LD-derivations (Apt 1997; Vasak and Potter 1986). Unfortunately, if we forbid goal rearrangements, many useful program transformations are no longer possible.

In this paper, we show through some examples that in our simple extension of logic programming we can restrict goal rearrangements to leftward moves of goal equalities. We also show that these moves preserve universal termination and do not increase nondeterminism, and thus, the deterioration of performance of the derived program is avoided.

The following simple example illustrates the essential idea of our technique which is based on the use of goal equalities. More complex examples will be presented in Sections 2 and 7.

Example 2

Suppose that during program transformation we are required to fold a clause of the form:

1. $p(X) \leftarrow a(X),\ b(X),\ c(X)$

by using a clause of the form:

2. $q(X) \leftarrow a(X),\ c(X)$

We can avoid a leftward move of the atom $c(X)$ by introducing, instead, an *equality between a goal variable and a goal*, thereby transforming clause 1 into the following clause:

3. $p(X) \leftarrow a(X),\ G{=}c(X),\ b(X),\ G$

Now we introduce the following predicate q' which takes the goal variable G as an argument:

4. $q'(X,G) \leftarrow a(X),\ G{=}c(X)$

Then we fold clause 3 using clause 4, thereby getting the clause:

5. $p(X) \leftarrow q'(X,G),\ b(X),\ G$

At this point we may continue the program transformation process by transforming clause 4, which defines the predicate q', instead of clause 2, which defines the predicate q. For instance, we may want to unfold clause 4 w.r.t. the goal $c(X)$ occurring as an argument of the equality predicate. □

As this example indicates, during program transformation we need to have at our disposal some transformation rules which can be used when goals occur as arguments. Indeed, in this paper:

- we introduce transformation rules for our logic language which allows goals as arguments,
- we show through some examples that the use of these rules makes it possible to improve efficiency without performing goal rearrangements which increase nondeterminism, and
- we prove that, under suitable conditions, our transformation rules are correct in the sense that they preserve the operational semantics of our logic language and, in particular, they preserve universal termination.

To show our correctness result, we first define the operational semantics of our logic language with goal arguments and goal variables. This semantics will be given in terms of ordinary LD-resolution, except for the following two important cases which we now examine.

The first case occurs when, during the construction of an LD-derivation, we generate a goal which has an occurrence of an unbound goal variable in the leftmost position. In this case, we say that the LD-derivation gets stuck. This treatment of unbound goal variables is in accordance with that of most Prolog systems which halt with error when trying to evaluate a call consisting of an unbound variable.

The second case occurs when we evaluate a goal equality of the form: $g_1 = g_2$. In this case we stipulate that $g_1 = g_2$ succeeds iff g_1 *is a goal variable which does not occur in* g_2 and it gets stuck otherwise. (In particular, for any goal g the evaluation of the equality $g = g$ gets stuck.) This somewhat restricted rule for the evaluation of goal equalities is required for the correctness of our transformation rules, as the following example shows.

Example 3
Let us consider the program $Q1$:

1. $h \leftarrow p(q)$
2. $p(G) \leftarrow G = q$
3. $q \leftarrow s$

where h, p, q, and s are predicate symbols and G is a goal variable. If we unfold the goal argument q in clause 1 using clause 3, we get the clause:

4. $h \leftarrow p(s)$

and we have the new program $Q2$ made out of clauses 2, 3, and 4. By using ordinary LD-resolution and unification, the goal h succeeds in the original program $Q1$, while it fails in the derived program $Q2$, because s does not unify with q. □

This example shows that the set of successes is *not* preserved by unfolding w.r.t. a goal argument. Similar incorrectness problems also arise with other transformation rules, such as folding and goal replacement. These problems come from the fact that operationally equivalent goals (such as q and s in the above example) are not syntactically equal.

In contrast, if we consider our restricted rule for the evaluation of goal equalities, the LD-derivation which starts from the goal h and uses the program $Q1$, gets stuck when the goal $q = q$ is selected. Also the LD-derivation which starts from the goal h and uses the derived program $Q2$, gets stuck when the goal $s = q$ is selected. Thus, the unfolding w.r.t. the argument q has preserved the operational semantics based on LD-resolution with our restricted rule for evaluating goal equalities.

In this paper we will consider two forms of correctness for our program transformations: weak correctness and strong correctness. Suppose that we have transformed a program P_1 into a program P_2 by applying our transformation rules. We say that this transformation is *weakly correct* iff, for any ordinary goal, that is, a goal without occurrences of goal variables and goal arguments, the following two properties hold: (i) if P_1 universally terminates, then P_2 universally terminates, and (ii) if both P_1 and P_2 universally terminate, then they compute the same set of *most general* answer substitutions. The transformation from P_1 to P_2 is *strongly correct* iff (i) it is weakly correct, and (ii) for any ordinary goal, if P_2 universally terminates, then P_1 universally terminates.

Thus, when a transformation is weakly correct, the transformed program may be more defined than the original program in the sense that there may be some goals which have no semantic value in the original program (that is, either their evaluation does not terminate or it gets stuck), whereas they have a semantic value in the transformed program (that is, their evaluation terminates and it does not get stuck).

This paper is organized as follows. In Section 2 we present an introductory example to motivate the language extension we will propose in this paper, and the transformation rules for this extended language. In Section 3 we give the definition of the syntax of our extended logic language with goal variables and goal arguments. In Section 4 we introduce the operational semantics of our extended language.

In Sections 5 and 6 we present the transformation rules and the conditions under which these rules are either weakly correct or strongly correct. For this purpose it is crucial that we assume that: (i) the evaluation of any goal variable gets stuck if that variable is unbound, and (ii) the evaluation of goal equalities is done according to the restricted rule we mentioned above. We will also show that, if a goal does not get stuck in a program, and we transform this program by using our rules, then the given goal does not get stuck in the transformed program. In this case, as it happens in the examples given in this paper, our operational semantics agrees with LD-resolution, and we can execute our transformed program by using ordinary Prolog systems.

In Section 7 we give some more examples of program transformation using our extended logic language and our transformation rules. We also give practical evidence that these transformations improve program efficiency. In Section 8 we make some final remarks and we compare our results with related work.

2 A motivating example

In order to present an example which motivates the introduction of goal variables and goal arguments, we begin by recalling a well-known program transformation strategy, called *tupling strategy* (Pettorossi and Proietti 1994). Given a program where some predicate calls require common subcomputations (detected by a suitable program analysis), the tupling strategy is realized by the following three steps.

The Tupling Strategy

(*Step* A) We introduce a new predicate defined by a clause, say T, whose body is the conjunction of the predicate calls with common subcomputations.

(*Step* B) We derive a program for the newly defined predicate which avoids redundant common subcomputations. This step can be divided into the following three substeps: (B.1) first, we unfold clause T, (B.2) then, we apply the goal replacement rule to avoid redundant goals, and (B.3) finally, we fold using clause T.

(*Step* C) By suitable folding steps using clause T, we express the predicates which are inefficiently computed by the initial program, in terms of the predicate introduced at Step (A).

A difficulty encountered when applying the tupling strategy is that, in order to apply the folding rule as indicated at Steps (B) and (C), it is often necessary to rearrange the atoms in the body of the clauses and, as already discussed in the Introduction, these rearrangements may affect program termination or increase nondeterminism.

The following example shows that this difficulty in the application of the tupling strategy can be overcome by introducing goal variables and goal arguments.

Example 4
Let us consider the following program *Deepest*:

1. $deepest(l(N), N) \leftarrow$
2. $deepest(t(L,R), X) \leftarrow depth(L, DL),\ depth(R, DR),\ DL \geqslant DR,$
   $deepest(L, X)$
3. $deepest(t(L,R), X) \leftarrow depth(L, DL),\ depth(R, DR),\ DL \leqslant DR,$
   $deepest(R, X)$
4. $depth(l(N), 1) \leftarrow$
5. $depth(t(L,R), D) \leftarrow depth(L, DL),\ depth(R, DR),\ max(DL, DR, M),$
   $plus(M, 1, D)$

where $deepest(T, X)$ holds iff T is a binary tree and X is the label of one of the deepest leaves of T. The two calls $depth(L, DL)$ and $deepest(L, X)$ in clause 2 may generate common redundant calls of the *depth* predicate. Indeed, both $depth(t(L1, R1), N)$ and $deepest(t(L1, R1), X)$ generate two calls of the form $depth(L1, DL)$ and $depth(R1, DR)$. In accordance with the tupling strategy, we transform the given program as follows.

(*Step* A) We introduce the following new predicate:

6. $dd(T, D, X) \leftarrow depth(T, D),\ deepest(T, X)$

(*Step* B.1) We apply a few times the unfolding rule, and we derive:

7. $dd(l(N), 1, N) \leftarrow$

8. $dd(t(L,R),D,X) \leftarrow depth(L,DL),\ depth(R,DR),$
$\quad max(DL,DR,M),\ plus(M,1,D),$
$\quad depth(L,DL1),\ depth(R,DR1),$
$\quad DL1 \geqslant DR1,\ deepest(L,X)$
9. $dd(t(L,R),D,X) \leftarrow depth(L,DL),\ depth(R,DR),$
$\quad max(DL,DR,M),\ plus(M,1,D),$
$\quad depth(L,DL1),\ depth(R,DR1),$
$\quad DL1 \leqslant DR1,\ deepest(R,X)$

(*Step* B.2) Since *depth* is *functional* with respect to its first argument, by applying the goal replacement rule we delete the atoms $depth(L,DL1)$ and $depth(R,DR1)$, in clauses 8 and 9, and we replace the occurrences of $DL1$ and $DR1$ by DL and DR, respectively, thereby getting the following clauses 10 and 11:

10. $dd(t(L,R),D,X) \leftarrow depth(L,DL),\ depth(R,DR),\ max(DL,DR,M),$
$\quad plus(M,1,D),\ DL \geqslant DR,\ deepest(L,X)$
11. $dd(t(L,R),D,X) \leftarrow depth(L,DL),\ depth(R,DR),\ max(DL,DR,M),$
$\quad plus(M,1,D),\ DL \leqslant DR,\ deepest(R,X)$

(*Step* B.3) To fold clause 10 using clause 6, we move $deepest(L,X)$ immediately to the right of $depth(L,DL)$. Similarly, in the body of clause 11 we move $deepest(R,X)$ immediately to the right of $depth(R,DR)$. Then, by folding we derive:

12. $dd(t(L,R),D,X) \leftarrow dd(L,DL,X),\ depth(R,DR),\ max(DL,DR,M),$
$\quad plus(M,1,D),\ DL \geqslant DR$
13. $dd(t(L,R),D,X) \leftarrow depth(L,DL),\ dd(R,DR,X),\ max(DL,DR,M),$
$\quad plus(M,1,D),\ DL \leqslant DR$

(*Step* C) Finally, we fold clauses 2 and 3 using clause 6, so that to evaluate the predicates *depth* and *deepest* we use the predicate *dd*, instead. Also for these folding steps we have to suitably rearrange the order of the atoms. By folding, we derive the following program *Deepest*1:

1. $deepest(l(N),N) \leftarrow$
14. $deepest(t(L,R),D,X) \leftarrow dd(L,DL,X),\ depth(R,DR),\ DL \geqslant DR$
15. $deepest(t(L,R),D,X) \leftarrow depth(L,DL),\ dd(R,DR,X),\ DL \leqslant DR$
7. $dd(l(N),1,N) \leftarrow$
12. $dd(t(L,R),D,X) \leftarrow dd(L,DL,X),\ depth(R,DR),\ max(DL,DR,M),$
$\quad plus(M,1,D),\ DL \geqslant DR$
13. $dd(t(L,R),D,X) \leftarrow depth(L,DL),\ dd(R,DR,X),\ max(DL,DR,M),$
$\quad plus(M,1,D),\ DL \leqslant DR$

To evaluate a goal of the form $deepest(t,X)$, where t is a ground tree and X is a variable, we may construct an LD-derivation using the program *Deepest*1 which does not generate redundant calls of *depth*. This LD-derivation performs only one traversal of the tree t and has linear length with respect to the size of t. However, this LD-derivation is constructed in a nondeterministic way, and if the corresponding LD-tree is traversed in a depth-first manner, like most Prolog systems do, the program exhibits an inefficient generate-and-test behaviour. Thus, in practice, the tupling strategy may diminish program efficiency.

The main reason of this decrease of efficiency is that, in order to fold clause 10, we had to move the atom $deepest(L, X)$ to a position to the left of $DL \geqslant DR$, and this move forces the evaluation of calls of $deepest(L, X)$ even when $DL \geqslant DR$ fails. (Notice that the move of $deepest(R, X)$ to the left of $DL \leqslant DR$ is harmless because $DL \leqslant DR$ is evaluated after the failure of $DL \geqslant DR$ and, thus, $DL \leqslant DR$ never fails.) □

In the following example, we present an alternative program derivation which starts from the same initial program *Deepest*. In this alternative derivation we will use our extended logic language which will be formally defined in the following Section 3. As already mentioned in the Introduction, when writing programs in our extended language, we may use: (i) the goal equality predicate =, (ii) goal variables occurring at top level in the body of a clause, and (iii) the disjunction predicate $\vee$. This alternative program derivation avoids harmful goal rearrangements and produces an efficient program without redundant subcomputations.

Example 5

Let us consider the program *Deepest* listed at the beginning of Example 4 consisting of clauses 1–5. By using disjunction in the body of a clause, clauses 2 and 3 can be rewritten as follows:

16. $deepest(t(L,R),X) \leftarrow depth(L,DL),\ depth(R,DR),$
$((DL \geqslant DR,\ deepest(L,X)) \vee (DL \leqslant DR,\ deepest(R,X)))$

After this initial transformation step the derived program, call it *DeepestOr*, consists of clauses 1, 4, 5, and 16.

Now we consider an extension of the tupling strategy which makes use of the transformation rules for logic programs with goal arguments and goal variables. These rules will be formally presented in Section 5. We proceed as follows.

(*Step* A) We introduce the following new predicate g which takes a goal variable G as an argument:

17. $g(T,D,X,G) \leftarrow depth(T,D),\ G = deepest(T,X)$

Notice also that in clause 17 the goal $deepest(T,X)$ occurs as an argument of the equality predicate.

(*Step* B) We derive a set of clauses for the newly defined predicate g as follows.

(*Step* B.1) We unfold clause 17 w.r.t. $depth(T,D)$ and we derive:

18. $g(l(N),1,X,G) \leftarrow G = deepest(l(N),X)$
19. $g(t(L,R),D,X,G) \leftarrow depth(L,DL),\ depth(R,DR),\ max(DL,DR,M),$
$plus(M,1,D),\ G = deepest(t(L,R),X)$

Now, by unfolding clauses 18 and 19 w.r.t. the atoms with the *deepest* predicate, we derive:

20. $g(l(N),1,N,true) \leftarrow$
21. $g(t(L,R),D,X,G) \leftarrow depth(L,DL),\ depth(R,DR),$
$max(DL,DR,M),\ plus(M,1,D),$
$G = (depth(L,DL1),\ depth(R,DR1),$
$((DL1 \geqslant DR1,\ deepest(L,X)) \vee (DL1 \leqslant DR1,\ deepest(R,X))))$

(*Step* B.2) We perform two goal replacement steps based on the functionality of *depth*, and from clause 21 we derive:

22. $g(t(L,R),D,X,G) \leftarrow depth(L,DL),\ depth(R,DR),$
    $max(DL,DR,M),\ plus(M,1,D),$
    $G=((DL \geqslant DR,\ deepest(L,X)) \vee (DL \leqslant DR,\ deepest(R,X)))$

(*Step* B.3) To fold clause 22 using clause 17, we first introduce goal equalities and we then perform suitable leftward moves of those goal equalities. We derive the following clause:

23. $g(t(L,R),D,X,G) \leftarrow depth(L,DL),\ GL=deepest(L,X),$
    $depth(R,DR),\ GR=deepest(R,X),$
    $max(DL,DR,M),\ plus(M,1,D),$
    $G=((DL \geqslant DR,GL) \vee (DL \leqslant DR,GR))$

Notice that we can move the goal equality $GL = deepest(L,X)$ to the left of the test $DL \geqslant DR$ without altering the operational semantics of our program. Indeed, this goal equality succeeds and binds the goal variable GL to the goal $deepest(L,X)$ without evaluating it. The goal $deepest(L,X)$ will be evaluated only when GL is called. A similar remark holds for the goal equality $GR = deepest(L,X)$. Now, by folding twice clause 23 using clause 17, we get:

24. $g(t(L,R),D,X,G) \leftarrow g(L,DL,X,GL),\ g(R,DR,X,GR),$
    $max(DL,DR,M),\ plus(M,1,D),$
    $G=((DL \geqslant DR,GL) \vee (DL \leqslant DR,GR))$

(*Step* C) Now we express the predicate *deepest* in terms of the new predicate *g* by transforming clause 16 as follows: (i) we first replace the two *deepest* atoms by the goal variables *GL* and *GR*, (ii) we then introduce suitable goal equalities, (iii) we then suitably move to the left the goal equalities, and (iv) we finally fold using clause 17. We derive the following clause:

25. $deepest(t(L,R),X) \leftarrow g(L,DL,X,GL),\ g(R,DR,X,GR),$
    $((DL \geqslant DR,GL) \vee (DL \leqslant DR,GR))$

Our final program *Deepest*2 is as follows:

1. $deepest(l(N),N) \leftarrow$
25. $deepest(t(L,R),X) \leftarrow g(L,DL,X,GL),\ g(R,DR,X,GR),$
    $((DL \geqslant DR,GL) \vee (DL \leqslant DR,GR))$
20. $g(l(N),1,N,true) \leftarrow$
24. $g(t(L,R),D,X,G) \leftarrow$
    $g(L,DL,X,GL),\ g(R,DR,X,GR),$
    $max(DL,DR,M),\ plus(M,1,D),$
    $G=((DL \geqslant DR,GL) \vee (DL \leqslant DR,GR))$

Now, when we evaluate a goal of the form $deepest(t,X)$, where t is a ground tree and X is a variable, *Deepest*2 does not generate redundant calls and it performs only one traversal of the tree t. *Deepest*2 is more efficient than *Deepest* because in the worst case *Deepest*2 performs $O(n)$ LD-resolution steps to compute an answer to $deepest(t,X)$, where n is the number of nodes of t, while the initial program

Deepest takes $O(n^2)$ LD-resolution steps. The program *Deepest*2 can be run by an ordinary Prolog system and computer experiments confirm substantial efficiency improvements with respect to the initial program *Deepest* (see Section 7.6).

Efficiency improvements, although smaller, are obtained also when comparing the final program *Deepest*2 with respect to the intermediate program *DeepestOr* which has been obtained from the initial program *Deepest* by replacing clauses 2 and 3 by clause 16, thereby avoiding the repetition of the common goals in clauses 2 and 3. Indeed, although more efficient than *Deepest* in the worst case, the program *DeepestOr* still takes a quadratic number of LD-resolution steps to compute an answer to $deepest(t, X)$. □

In Section 7 we will present more examples of program derivation and we will also provide some experimental results.

3 The extended logic language with goals as arguments

Let us now formally define our extended logic language. Suppose that the following pairwise disjoint sets are given: (i) *individual variables*: $X, X_1, X_2, \ldots$, (ii) *goal variables*: $G, G_1, G_2, \ldots$, (iii) *function symbols* (with arity): $f, f_1, f_2, \ldots$, (iv) *primitive predicate symbols*: *true*, *false*, $=_t$ (denoting equality between terms), $=_g$ (denoting equality between goals), and (v) *predicate symbols* (with arity): $p, p_1, p_2, \ldots$ Individual and goal variables are collectively called *variables*, and they are ranged over by $V, V_1, V_2, \ldots$ Occasionally, we shall feel free to depart from these naming conventions, if no confusion arises.

Terms: $t, t_1, t_2, \ldots$, *goals*: $g, g_1, g_2, \ldots$, and *arguments*: $u, u_1, u_2, \ldots$, have the following syntax:

$$t ::= X \mid f(t_1, \ldots, t_n)$$
$$g ::= G \mid true \mid false \mid t_1 =_t t_2 \mid g_1 =_g g_2 \mid p(u_1, \ldots, u_m) \mid g_1 \wedge g_2 \mid g_1 \vee g_2$$
$$u ::= t \mid g$$

The binary operators $\wedge$ (conjunction) and $\vee$ (disjunction) are assumed to be associative with neutral elements *true* and *false*, respectively. Thus, a goal g is the same as $true \wedge g$ and $g \wedge true$. Similarly, g is the same as $false \vee g$ and $g \vee false$. Goals of the form $p(u_1, \ldots, u_m)$ are also called *atoms*. In the sequel, for reasons of simplicity, we will write $=$, instead of $=_t$ or $=_g$, and we leave it to the reader to distinguish between the two equalities according to the context of use. Notice that, according to our operational semantics (see Section 4), $\vee$ is commutative, $\wedge$ is not commutative, $=_t$ is symmetric, and $=_g$ is not symmetric.

Clauses $c, c_1, c_2, \ldots$ have the following syntax:

$$c ::= p(V_1, \ldots, V_m) \leftarrow g$$

where p is a non-primitive predicate symbol and $V_1, \ldots, V_m$ are distinct variables. The atom $p(V_1, \ldots, V_m)$ is called the *head* of the clause and the goal g is called the *body* of the clause. A clause of the form: $p(V_1, \ldots, V_m) \leftarrow true$ will also be written as $p(V_1, \ldots, V_m) \leftarrow$.

Programs $P, P_1, P_2, \ldots$ are sets of clauses of the form:

$$\begin{array}{l} p_1(V_1,\ldots,V_{m1}) \leftarrow g_1 \\ \qquad\qquad \vdots \\ p_k(V_1,\ldots,V_{mk}) \leftarrow g_k \end{array}$$

where $p_1,\ldots,p_k$ are distinct non-primitive predicate symbols, and every non-primitive predicate symbol occurring in $\{g_1,\ldots,g_k\}$ is an element of $\{p_1,\ldots,p_k\}$. Each clause head has distinct variables as arguments. Given a program P and a non-primitive predicate p occurring in P, the unique clause in P of the form $p(V_1,\ldots,V_m) \leftarrow g$, is called the *definition* of p in P. We say that a predicate p is *defined* in a program P iff p has a definition in P.

An *ordinary goal* is a goal without goal variables or goal arguments. Formally, an *ordinary* goal has the following syntax:

$$g ::= \mathit{true} \mid \mathit{false} \mid t_1 =_t t_2 \mid p(t_1,\ldots,t_m) \mid g_1 \wedge g_2 \mid g_1 \vee g_2$$

where $t_1, t_2, \ldots, t_m$ are terms. *Ordinary programs* are programs whose goals are ordinary goals.

Notes on syntax

1. When no confusion arises, we also use comma, instead of $\wedge$, for denoting conjunction.
2. The assumption that in our programs clause heads have only variables as arguments is not restrictive, because we may always replace a non-variable argument, say u, by a variable argument, say V, in the head of a clause, at the expense of adding the extra equality $V = u$ in the body.
3. The assumption that in every program there exists at most one clause for each predicate symbol is not restrictive, because one may use disjunctions in the body of clauses. In particular, every definite logic program written by using the familiar syntax (Lloyd, 1987), can be rewritten into an equivalent program of our language by suitable introductions of equalities and $\vee$ operators in the bodies of clauses.
4. Our logic language is a typed language in the sense that: (i) every individual variable has type *term*, (ii) every function symbol of arity n has type $\mathit{term}^n \to \mathit{term}$, (iii) *true*, *false*, and every goal variable have type *bool*, (iv.1) $=_t$ has type $\mathit{term} \times \mathit{term} \to \mathit{bool}$, (iv.2) $=_g$ has type $\mathit{bool} \times \mathit{bool} \to \mathit{bool}$, and (v) every predicate symbol of arity n has a unique type of the form: $(\mathit{term} \mid \mathit{bool})^n \to \mathit{bool}$. We assume that all our programs can be uniquely typed according to the above rules.

4 The operational semantics

In this section we define the operational semantics of our extended logic language. We choose a syntax-directed style of presentation which makes use of *deduction rules*. For an elementary presentation of this technique, sometimes called *structural operational semantics* or *natural semantics*, the reader may refer to Winskel (1993).

Before defining the semantics of our logic language, we recall the following notions. By $\{V_1/u_1,\ldots,V_m/u_m\}$ we denote the substitution of $u_1,\ldots,u_m$ for the variables $V_1,\ldots,V_m$. As usual, we assume that the V_i's are all distinct and for $i=1,\ldots,m$, u_i is distinct from V_i. By ε we denote the identity substitution. By $\vartheta \restriction S$ we denote the *restriction* of the substitution ϑ to set S of variables, that is, $\vartheta \restriction S = \{V/u \mid V/u \in \vartheta \text{ and } V \in S\}$. Given the substitutions $\vartheta, \eta_1,\ldots,\eta_k$, by $\vartheta \circ \{\eta_1,\ldots,\eta_k\}$ we denote the set of substitutions $\{\vartheta\eta_1,\ldots,\vartheta\eta_k\}$ (where, as usual, juxtaposition of substitutions denotes composition (Lloyd 1987)). By $g\vartheta$ we denote the application of the substitution ϑ to the goal g. By $mgu(t_1,t_2)$ we denote a relevant, idempotent, most general unifier of the terms t_1 and t_2.

The set of all substitutions is denoted by *Subst* and the set of all *finite* subsets of *Subst* is denoted by $\mathscr{P}(Subst)$. Given $A,B \in \mathscr{P}(Subst)$, we say that A and B are *equally general* with respect to a goal g iff (i) for every $\alpha \in A$ there exists $\beta \in B$ such that $g\alpha$ is an instance of $g\beta$, and symmetrically, (ii) for every $\beta \in B$ there exists $\alpha \in A$ such that $g\beta$ is an instance of $g\alpha$. For example, $A=\{\{X/t\}$, $\{X/Y\}$, $\{X/Z\}\}$ and $B=\{\{X/W\}\}$ are equally general with respect to the goal $p(X)$.

Given a set of substitutions $A \in \mathscr{P}(Subst)$ and a goal g, let $mostgen(A,g)$ denote a largest subset of $\{g\vartheta \mid \vartheta \in A\}$ such that for any two goals g_1 and g_2 in $mostgen(A,g)$, g_1 is not an instance of g_2. For example, $mostgen(\{\{X/t\},\{X/Y\},\{X/Z\}\},p(X)) = \{p(Y)\}$. Notice that the set denoted by *mostgen* is not uniquely determined. However, it can be shown that, whatever choice we make for the set denoted by *mostgen*, any two sets of substitutions A and B are equally general with respect to a goal g iff there exists a bijection ρ from $mostgen(A,g)$ to $mostgen(B,g)$ such that for any goal $h \in mostgen(A,g)$, $\rho(h)$ is a variant of h. In this case we write $mostgen(A,g) \approx mostgen(B,g)$.

We use $g[u]$ to denote a goal g in which we have selected an occurrence of its subconstruct u, where u may be either a term or a goal. By $g[\_]$ we denote the goal $g[u]$ without the selected occurrence of its subconstruct u. We say that $g[\_]$ is a *goal context*. For any syntactic construct r, we use $vars(r)$ to denote the set of variables occurring in r and, for any set $\{r_1,\ldots,r_m\}$ of syntactic constructs, we use $vars(r_1,\ldots,r_m)$ to denote the set of variables $vars(r_1) \cup \ldots \cup vars(r_m)$. In particular, given a substitution ϑ, a variable belongs to $vars(\vartheta)$ iff it occurs either in the domain of ϑ or in the range of ϑ. Given two goals g and g_1 and a clause c of the form $p(V_1,\ldots,V_m) \leftarrow g[g_1]$, the *local variables* of g_1 in c are those in the set $vars(g_1) - (\{V_1,\ldots,V_m\} \cup vars(g[\_]))$.

Given a program P, we define the semantics of P as a ternary relation $P \vdash g \mapsto A$, where g is a goal and A is a finite set of substitutions, meaning that for P and g all derivations are finite and A is the finite set of *answer substitutions* which are computed by these derivations. The relation $P \vdash g \mapsto A$ is defined by the deduction rules given in Figure 1.

A *deduction tree* τ for $P \vdash g \mapsto A$ is a tree such that: (i) the root of τ is $P \vdash g \mapsto A$, and (ii) for every node n of τ with sons $n_1,\ldots,n_k$ (with $k \geqslant 0$), there exists an instance of a deduction rule, say r, whose conclusion is n and whose premises are $n_1,\ldots,n_k$. We say that *n is derived by applying rule r* to $n_1,\ldots,n_k$. A *proof* of $P \vdash g \mapsto A$ is a finite deduction tree for $P \vdash g \mapsto A$ where every leaf is a deduction rule which has no premises.

$$(tt)\qquad \frac{}{P \vdash true \mapsto \{\varepsilon\}}$$

$$(ff)\qquad \frac{}{P \vdash false \wedge g \mapsto \emptyset}$$

$$(teq1)\qquad \frac{}{P \vdash (t_1 = t_2) \wedge g \mapsto \emptyset} \qquad \text{if } t_1 \text{ and } t_2 \text{ are non-unifiable terms}$$

$$(teq2)\qquad \frac{P \vdash g\, mgu(t_1,t_2) \mapsto A}{P \vdash (t_1 = t_2) \wedge g \mapsto (mgu(t_1,t_2) \circ A)} \qquad \text{if } t_1 \text{ and } t_2 \text{ are unifiable terms}$$

$$(geq)\qquad \frac{P \vdash g_2\{G/g_1\} \mapsto A}{P \vdash (G = g_1) \wedge g_2 \mapsto (\{G/g_1\} \circ A)} \qquad \text{if the goal variable } G \text{ is not in } vars(g_1)$$

$$(at)\qquad \frac{P \vdash g_1\{V_1/u_1,\ldots,V_m/u_m\} \wedge g \mapsto A}{P \vdash p(u_1,\ldots,u_m) \wedge g \mapsto A \upharpoonright S}$$

where $p(V_1,\ldots,V_m) \leftarrow g_1$ is a renamed apart clause of P and S is $vars(p(u_1,\ldots,u_m) \wedge g)$

$$(or)\qquad \frac{P \vdash g_1 \wedge g \mapsto A_1 \qquad P \vdash g_2 \wedge g \mapsto A_2}{P \vdash (g_1 \vee g_2) \wedge g \mapsto (A_1 \cup A_2)}$$

Fig. 1. Operational semantics.

We say that $P \vdash g \mapsto A$ *holds* iff there exists a proof of $P \vdash g \mapsto A$. If $P \vdash g \mapsto A$ holds and $A \neq \emptyset$, then we say that g *succeeds* in P, written $P \vdash g \downarrow true$. Otherwise, if $P \vdash g \mapsto \emptyset$ holds, then we say that g *fails* in P, written $P \vdash g \downarrow false$. If g either succeeds or fails in P we say that g *terminates* in P. We say that a goal g is *stuck* iff it is either of the form $G \wedge g_1$, where G is a goal variable, or of the form $(g_0 = g_1) \wedge g_2$, where either g_0 is a non-variable goal or g_0 is a goal variable occurring in g_1. We say that g *gets stuck* in P iff there exist a set A of substitutions and a (finite or infinite) deduction tree τ for $P \vdash g \mapsto A$ such that a leaf of τ is of the form $P \vdash g_1 \mapsto B$ and g_1 is stuck. For instance, the goal $(G = p) \wedge (G = q)$ gets stuck in any program P. We say that g is *safe* in P iff g does not get stuck in P.

For every program P and goal g, the three cases: (i) g succeeds in P, (ii) g fails in P, and (iii) g gets stuck in P, are pairwise mutually exclusive, but not exhaustive. Indeed, there is a fourth case in which the unique maximal deduction tree with root $P \vdash g \mapsto A$ is infinite and each of its leaves, if any, is the conclusion of a deduction rule which has no premises. In this case no A exists such that $P \vdash g \mapsto A$ holds and g does not get stuck in P.

Notes on semantics

1. In our presentation of the deduction rules we have exploited the assumption that $\wedge$ and $\vee$ are associative operators with neutral elements *true* and *false*, respectively. For instance, we have not introduced the rule $\frac{}{P \vdash false \mapsto \emptyset}$ because it is an instance of rule (*ff*) for $g = true$.

2. Given a program P and a goal g, if there exists a proof for $P \vdash g \mapsto A$ for some A, then the proof is unique up to isomorphism. More precisely, given two proofs, say π_1 for $P \vdash g \mapsto A_1$ and π_2 for $P \vdash g \mapsto A_2$, there exists a bijection ρ from the nodes of π_1 to the nodes of π_2 which preserves the application of the deduction rules and if $\rho(P \vdash g_1 \mapsto B_1) = P \vdash g_2 \mapsto B_2$ then
   (i) g_1 is a variant of g_2, and
   (ii) $\forall \beta_1 \in B_1\ \exists \beta_2 \in B_2$ such that $g_1\beta_1$ is a variant of $g_2\beta_2$, and
   (iii) $\forall \beta_2 \in B_2\ \exists \beta_1 \in B_1$ such that $g_2\beta_2$ is a variant of $g_1\beta_1$.

   This property is a consequence of the fact that: (i) for any program P and goal g, there exists at most one rule instance whose conclusion is of the form $P \vdash g \mapsto A$ for some A, and (ii) our rules for the operational semantics are deterministic, in the sense that no choice has to be made when one applies them during the construction of a proof, apart from the choice of how to compute the most general unifiers and how to rename apart the clauses.
   In particular, any two sets A_1 and A_2 of answer substitutions for a program P and a goal g, are related as follows: if $P \vdash g \mapsto A_1$ and $P \vdash g \mapsto A_2$ then $\forall \alpha_1 \in A_1\, \exists \alpha_2 \in A_2\ g\alpha_1$ is a variant of $g\alpha_2$ and $\forall \alpha_2 \in A_2\, \exists \alpha_1 \in A_1\ g\alpha_2$ is a variant of $g\alpha_1$. Thus, A_1 and A_2 are equally general with respect to g. The same property holds also for any two sets of computed answer substitutions which are constructed by LD-resolution (recall that by LD-resolution we can construct different sets of computed answer substitutions by choosing different most general unifiers and different variable renamings).
   Notice that if $P \vdash g \mapsto A_1$ and $P \vdash g \mapsto A_2$ hold then A_1 and A_2 may have different cardinality. Indeed, let us consider the program P consisting of the following clause only:
   $p(X,Y,Z) \leftarrow (X=Y \wedge Z=Y) \vee (X=Z \wedge Y=Z)$
   In this case, since both Z/Y and Y/Z are most general unifiers of $Y=Z$, we have that both $P \vdash p(X,Y,Z) \mapsto \{\{X/Y, Z/Y\}, \{X/Z, Y/Z\}\}$ and $P \vdash p(X,Y,Z) \mapsto \{\{X/Y, Z/Y\}\}$ hold. Notice also that the substitution $\{X/Y, Z/Y\}$ is more general than the substitution $\{X/Z, Y/Z\}$ and vice versa.
3. If $P \vdash g \mapsto A$ and $\vartheta \in A$, then the domain of ϑ is a subset of *vars*(g).
4. In the presentation of the deduction rules for the ternary relation $P \vdash g \mapsto A$, the program P never changes and thus, it could have been omitted. However, the explicit reference to P is useful for presenting our Correctness Theorem (see Theorem 2 in Section 6).
5. We assume that in any relation $P \vdash g \mapsto A$, the program P and the goal g have consistent types, that is, the type of every function and predicate symbol should be the same in P and in g. For instance, if $P = \{p(G) \leftarrow\}$ where G is a goal variable, then $P \vdash p(0) \mapsto \{\varepsilon\}$ does *not* hold, because in the program P the predicate p has type *bool* $\rightarrow$ *bool*, while in the goal $p(0)$ the predicate p has type *term* $\rightarrow$ *bool*. Moreover, for any relation $P \vdash g_1 \mapsto A_1$ occurring in the proof of $P \vdash g \mapsto A$, we have that program P and goal g_1 have consistent types.

Now we discuss the relationship between LD-resolution and the operational semantics defined in this section. Apart from the style of presentation (usually

LD-resolution is presented by means of the notions of *LD-derivation* and *LD-tree* (Apt 1997; Lloyd 1987)), LD-resolution differs from our operational semantics only in the treatment of goal equality. Indeed, by using LD-resolution, the goal equality $g_1 = g_2$ is evaluated by applying the ordinary unification algorithm also in the case where g_1 is not a goal variable or g_1 is a goal variable occurring in $vars(g_2)$. In contrast, according to our operational semantics, a goal of the form $g_1 = g_2$ is evaluated by unifying g_1 and g_2, only if g_1 is a variable which does not occur in $vars(g_2)$ (see rule (*geq*) above).

Thus, if a goal g is safe in P, then the evaluation of g according to our operational semantics agrees with the one which uses LD-resolution in the following sense: if g is safe in P, then there exists a set A of answer substitutions such that $P \vdash g \mapsto A$ holds iff: (i) all LD-derivations starting from g and using P are finite (that is, g *universally terminates* in P (Apt 1997; Vasak and Potter 1986)), and (ii) A is the set of the computed answer substitutions obtained by LD-resolution. Point (i) follows from the fact that in our operational semantics, the evaluation of a disjunction of goals (see the (*or*) rule) requires the evaluation of each disjunct. Thus, in order to compute the relation $P \vdash g \mapsto A$ in the case where g is safe in P, we can use any ordinary Prolog system which implements LD-resolution.

Notice that, given a program P and a goal g, if the LD-tree has an infinite LD-derivation, then no set A of answer substitutions exists such that $P \vdash g \mapsto A$. In particular, for the program $P = \{p(0) \leftarrow, p(X) \leftarrow p(X)\}$ no A exists such that $P \vdash p(X) \mapsto A$, while the set of computed answer substitutions constructed by LD-resolution for the program P and the goal $p(X)$ is the singleton consisting of the substitution $\{X/0\}$ only.

It may also be the case that a goal g is not safe in a program P (thus, there exists no set A of answer substitutions such that $P \vdash g \mapsto A$ holds) while, by using LD-resolution, g succeeds or fails in P. For instance, for any program and for any two distinct nullary predicates p and q, (i) the goal $p = p$ is not safe, while it succeeds by using LD-resolution and (ii) the goal $p = q$ is not safe, while it fails by using LD-resolution.

We recall that our interpretation of goal equality is motivated by the fact that we want the operational semantics to be preserved by program transformations and, in particular, by unfolding. As already shown in the Introduction, unfortunately, unfolding does not preserve the operational semantics based on ordinary LD-resolution.

The following Proposition 1 establishes an important property of our operational semantics. This property is useful for the proof the correctness results in Section 6 (see Theorem 2). The proof of this proposition is similar to the one in the case of LD-resolution for definite programs (see, for instance, Lloyd (1987)), and will be omitted.

Proposition 1

Let P be a program, g be an ordinary goal, and A be a set of substitutions such that $P \vdash g \mapsto A$. Then, for all $\vartheta \in Subst$, the following hold:

(i) $g\vartheta$ terminates, that is, either $P \vdash g\vartheta \downarrow true$ or $P \vdash g\vartheta \downarrow false$, and

(ii.1) $P \vdash g\vartheta \downarrow \mathit{true}$ iff there exists $\alpha \in A$ such that $g\vartheta$ is an instance of $g\alpha$, and
(ii.2) $P \vdash g\vartheta \downarrow \mathit{false}$ iff it does *not* exist $\alpha \in A$ such that $g\vartheta$ is an instance of $g\alpha$.

Let us conclude this section by introducing the notions of *refinement* and *equivalence* between programs which we will use in Section 6 to state the weak and strong correctness of the program transformations that can be realized by applying our transformation rules. These rules are presented in the next section.

Definition 1 (*Refinement and Equivalence*)
Given two programs P_1 and P_2, we say that P_2 is a *refinement* of P_1, written $P_1 \sqsubseteq P_2$, iff for every ordinary goal g and for every $A \in \mathscr{P}(\mathit{Subst})$, if $P_1 \vdash g \mapsto A$ then there exists $B \in \mathscr{P}(\mathit{Subst})$ such that:

(1) $P_2 \vdash g \mapsto B$ and
(2) A and B are equally general with respect to g.

We say that P_1 is *equivalent* to P_2, written $P_1 \equiv P_2$, iff $P_1 \sqsubseteq P_2$ and $P_2 \sqsubseteq P_1$.

Remark 1
Recall that Condition (2) can be written as $\mathit{mostgen}(A, g) \approx \mathit{mostgen}(B, g)$. In this sense we will say that if $P_1 \sqsubseteq P_2$ and the ordinary goal g terminates in P_1, then the most general answer substitutions for g are the same in P_1 and P_2, modulo variable renaming. □

Remark 2
$P_1 \sqsubseteq P_2$ implies that, for every ordinary goal g,
– if g succeeds in P_1 then g succeeds in P_2, and
– if g fails in P_1 then g fails in P_2. □

Theorem 2 stated in Section 6 shows that, if from program P_1 we derive program P_2 by using our transformation rules and suitable conditions hold, then $P_1 \sqsubseteq P_2$. In this case we say that the transformation is *weakly correct*. If additional conditions hold, then we may have that $P_1 \equiv P_2$ and we say that the transformation is *strongly correct*.

In Section 6 we will also show that our transformation rules preserve safety, that is, if from program P_1 we derive program P_2 by using the transformation rules and goal g is safe in P_1, then goal g is safe also in P_2.

5 The transformation rules

In this section we present the transformation rules for our extended logic language. We assume that starting from an initial program P_0 we have constructed the *transformation sequence* $P_0, \ldots, P_i$ (Pettorossi and Proietti 1994; Tamaki and Sato 1984). By an application of a transformation rule, from program P_i we derive a new program P_{i+1}.

Rule R1 (*Definition Introduction*)
We derive the new program P_{i+1} by adding to program P_i a new clause, called a *definition*, of the form:

$\mathit{newp}(V_1, \ldots, V_m) \leftarrow g$

where: (i) *newp* is a new non-primitive predicate symbol not occurring in any program of the sequence $P_0, \ldots, P_i$, (ii) the non-primitive predicate symbols occurring in g are defined in P_0, and (iii) $V_1, \ldots, V_m$ are some of (possibly all) the distinct variables occurring in g.

The set of all definitions introduced during the transformation sequence $P_0, \ldots, P_i$, is denoted by Def_i. Thus, $Def_0 = \emptyset$.

Rule R2 (*Unfolding*)
Let c_1: $h \leftarrow body[p(u_1, \ldots, u_m)]$ be a renamed apart clause in program P_i where p is a non-primitive predicate symbol. Let d: $p(V_1, \ldots, V_m) \leftarrow g$ be a clause in $P_0 \cup Def_i$. By *unfolding* c_1 *w.r.t.* $p(u_1, \ldots, u_m)$ *using* d we derive the new clause c_2: $h \leftarrow body[g\{V_1/u_1, \ldots, V_m/u_m\}]$. We derive the new program P_{i+1} by replacing in program P_i clause c_1 by clause c_2.

Rule R3 (*Folding*)
Let c_1: $h \leftarrow body[g\vartheta]$ be a renamed apart clause in program P_i and let d: $p(V_1, \ldots, V_m) \leftarrow g$ be a clause in Def_i. Suppose that, for every local variable V of g in d, we have that:

(1) $V\vartheta$ is a local variable of $g\vartheta$ in c_1, and
(2) the variable $V\vartheta$ does not occur in $W\vartheta$, for any variable W occurring in g and different from V.

Then, by *folding* c_1 *using* d we derive the new clause c_2: $h \leftarrow body[p(V_1, \ldots, V_m)\vartheta]$. We derive the new program P_{i+1} by replacing in program P_i clause c_1 by clause c_2.

To present the goal replacement rule (see rule R4 below) we introduce the notion of *replacement law*. Basically, a replacement law denotes two goals which can be replaced one for the other in the body of a clause. We have two kinds of replacement laws: the *weak* and the *strong* replacement laws, which ensure weak and strong correctness, respectively (see the end of this section for an informal discussion and Section 6 for a formal proof of this fact).

First we need the following definition.

Definition 2 (*Depth of a Deduction Tree*)
Let τ be a finite deduction tree and let m be the maximal number of applications of the (*at*) rule in a root-to-leaf path of τ. Then we say that τ has *depth m*.

Let π be a proof for $P \vdash g \mapsto A$, for some program P, goal g, and set A of substitutions, and let m be the depth of π. If $A = \emptyset$ we write $P \vdash g \downarrow_m \mathit{false}$; otherwise, if $A \neq \emptyset$ we write $P \vdash g \downarrow_m \mathit{true}$.

Recall that, given a program P and a goal g, if for some set A of substitutions there exists a proof for $P \vdash g \mapsto A$, then the proof is unique up to isomorphism. In particular, given a proof for $P \vdash g \mapsto A_1$ and a proof for $P \vdash g \mapsto A_2$, they have the same depth.

Definition 3 (*Replacement Laws*)
Let P be a program, let g_1 and g_2 be two goals, and let V be a set of variables.

(i) The relation $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ holds iff for every goal context $g[\_]$ such that $vars(g[\_]) \cap vars(g_1, g_2) \subseteq V$, and for every $b \in \{true, false\}$, we have that:

if $P \vdash g[g_1] \downarrow b$ then $P \vdash g[g_2] \downarrow b$. (†)

(ii) The relation $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$, called a *weak replacement law*, holds iff for every goal context $g[\_]$ such that $vars(g[\_]) \cap vars(g_1, g_2) \subseteq V$, and for every $b \in \{true, false\}$, we have that:

if $P \vdash g[g_1] \downarrow_m b$ then $P \vdash g[g_2] \downarrow_n b$ with $m \geqslant n$. (††)

(iii) The relation $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longleftrightarrow} g_2)$, called a *strong replacement law*, holds iff $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$ and $P \vdash \forall V\,(g_2 \longrightarrow g_1)$.

(iv) We write $P \vdash \forall V\,(g_1 \overset{=}{\longleftrightarrow} g_2)$ to mean that the strong replacement laws $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$ and $P \vdash \forall V\,(g_2 \overset{\geqslant}{\longrightarrow} g_1)$ hold.

If $V = \emptyset$ then $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$ is also written as $P \vdash g_1 \overset{\geqslant}{\longrightarrow} g_2$. If $V = \{V_1, \ldots, V_n\}$ then $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$ is also written as $P \vdash \forall V_1, \ldots, V_n\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$. If $V = vars(g_1, g_2)$ then $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$ is also written as $P \vdash \forall\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$.

A few comments on Definition 3 are now in order.

1. In the relation $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ we have used the set V of universally quantified variables as a notational device for indicating that when we replace g_1 by g_2 in a clause $h \leftarrow body[g_1]$, the variables in common between $h \leftarrow body[\_]$ and (g_1, g_2) are those in V (see the goal replacement rule R4 below). Thus, $vars(g_1) - V$ is the set of the local variables of g_1 in $h \leftarrow body[g_1]$ and $vars(g_2) - V$ is the set of the local variables of g_2 in $h \leftarrow body[g_2]$.
2. Implication (††) implies Implication (†).
3. Every strong replacement law is also a weak replacement law.
4. If $P \vdash \forall V\,(g_1 \overset{=}{\longleftrightarrow} g_2)$ then there exists $A_1 \in \mathscr{P}(Subst)$ such that $P \vdash g_1 \mapsto A_1$ has a proof of depth m iff there exists $A_2 \in \mathscr{P}(Subst)$ such that $P \vdash g_2 \mapsto A_2$ has a proof of depth m. Moreover, if both proofs exist, $A_1 = \emptyset$ iff $A_2 = \emptyset$.

The properties listed in the next proposition follow directly from Definition 3.

Proposition 2

Let P be a program, let g_1 and g_2 be goals, and let V be a set of variables.

(i) $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ holds iff for every goal context $g[\_]$ such that $vars(g[\_]) \cap vars(g_1, g_2) \subseteq V$, $P \vdash \forall W\,(g[g_1] \longrightarrow g[g_2])$ holds, where $W = V \cup vars(g[\_])$.

(ii) $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ holds iff $P \vdash \forall W\,(g_1 \longrightarrow g_2)$ holds, where $W = V \cap vars(g_1, g_2)$.

(iii) $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ holds iff for every $W \subseteq V$, $P \vdash \forall W\,(g_1 \longrightarrow g_2)$ holds.

(iv) $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ holds iff for every substitution ϑ such that $vars(\vartheta) \cap vars(g_1, g_2) \subseteq V$, $P \vdash \forall W\,(g_1\vartheta \longrightarrow g_2\vartheta)$ holds, where $W = vars(V\vartheta)$.

(v) $P \vdash \forall V\,(g_1 \longrightarrow g_2)$ holds iff for every renaming substitution ρ such that $vars(\rho) \cap V = \emptyset$, $P \vdash \forall V\,(g_1\rho \longrightarrow g_2\rho)$ holds.

The properties obtained from (i)–(v) by replacing $\longrightarrow$ by $\overset{\geqslant}{\longrightarrow}$ are also true.

Definition 4

We say that a weak replacement law $P \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$ (or a strong replacement law $P \vdash \forall V\,(g1 \overset{\geqslant}{\longleftrightarrow} g_2)$) *preserves safety* iff for every goal context $g[\_]$ such that $vars(g[\_]) \cap vars(g_1, g_2) \subseteq V$, we have that:

if $g[g_1]$ is safe in P then $g[g_2]$ is safe in P.

Rule R4 (*Goal Replacement*)

Let c_1: $h \leftarrow body[g_1]$ be a clause in program P_i and let g_2 be a goal such that: (i) all non-primitive predicate symbols occurring in g_1 or g_2 are defined in P_0, and either (ii.1) $P_0 \vdash \forall V\,(g_1 \overset{\geqslant}{\longrightarrow} g_2)$, or (ii.2) $P_0 \vdash \forall V\,(g_1 \overset{\geqslant}{\longleftrightarrow} g_2)$, where $V = vars(h, body[\_]) \cap vars(g_1, g_2)$.

By *goal replacement* we derive the new clause c_2: $h \leftarrow body[g_2]$, and we derive the new program P_{i+1} by replacing in program P_i clause c_1 by clause c_2.

In case (ii.1) we say that the goal replacement is *based on a weak replacement law*. In case (ii.2) we say that the goal replacement is *based on a strong replacement law*. We say that the goal replacement *preserves safety* iff it is based on a (weak or strong) replacement law which preserves safety.

Implication (††) of Definition 3 makes $\overset{\geqslant}{\longrightarrow}$ and $\overset{\geqslant}{\longleftrightarrow}$ to be *improvement* relations in the sense of Sands (1996). As stated in Theorem 2 of Section 6, Implication (††) is required for ensuring the weak correctness of a goal replacement step, while Implication (†) of Definition 3 does not suffice. This fact is illustrated by the following example.

Example 6

Let us consider the program P_1:

1. $p \leftarrow q$
2. $q \leftarrow$

We have that $P_1 \vdash q \longrightarrow p$ and thus, Implication (†) holds by taking g_1 to be q, g_2 to be p, and $g[\_]$ to be the empty goal context. The replacement of q by p in clause 1 produces the following program P_2:

1*. $p \leftarrow p$

2. $q \leftarrow$

This replacement is not an application of rule R4, because Implication (††) does not hold. (Indeed, we have that the depth of the proof for $P_1 \vdash q \mapsto \{\varepsilon\}$ is smaller than the depth of the proof for $P_1 \vdash p \mapsto \{\varepsilon\}$). The transformation from program P_1 to program P_2 is not weakly correct (nor strongly correct), because p succeeds in P_1, while p does not terminate in P_2, and thus, it is not the case that $P_1 \sqsubseteq P_2$. □

The reader may check that, for any program P, and goals g, g_1, g_2, and g_3, we have the following replacement laws. It can be shown that these replacement laws preserve safety.

1. *Boolean Laws*:

$$\begin{array}{ll}
P \vdash \forall(g \wedge true \overset{=}{\longleftrightarrow} g) & P \vdash \forall(g \wedge g \overset{\geqslant}{\longrightarrow} g) \\
P \vdash \forall(true \wedge g \overset{=}{\longleftrightarrow} g) & P \vdash \forall(g \vee g \overset{=}{\longleftrightarrow} g) \\
P \vdash \forall(true \vee g \overset{\geqslant}{\longrightarrow} true) & P \vdash \forall(g_1 \vee g_2 \overset{=}{\longleftrightarrow} g_2 \vee g_1) \\
P \vdash \forall(g \wedge false \overset{\geqslant}{\longrightarrow} false) & P \vdash \forall((g_1 \wedge g_2) \vee (g_1 \wedge g_3) \overset{=}{\longleftrightarrow} g_1 \wedge (g_2 \vee g_3)) \\
P \vdash \forall(false \wedge g \overset{=}{\longleftrightarrow} false) & P \vdash \forall((g_1 \wedge g_2) \vee (g_3 \wedge g_2) \overset{=}{\longleftrightarrow} (g_1 \vee g_3) \wedge g_2) \\
P \vdash \forall(false \vee g \overset{=}{\longleftrightarrow} g) & P \vdash \forall((g_1 \vee g_2) \wedge (g_1 \vee g_3) \overset{\geqslant}{\longrightarrow} g_1 \vee (g_2 \wedge g_3))
\end{array}$$

In the following replacement laws 2.1 and 2.2, according to our conventions, V stands for either an individual variable or a goal variable, and u stands for either a term or a goal, respectively.

2.1. *Introduction and elimination of equalities*:

$$P \vdash \forall U\,(g[u] \overset{=}{\longleftrightarrow} ((V = u) \wedge g[V])) \quad \text{where } U = vars(g[u]) \text{ and } V \notin U.$$

2.2. *Rearrangement of equalities*:

$$P \vdash \forall U\,(g[(V = u) \wedge g_1] \overset{=}{\longleftrightarrow} ((V = u) \wedge g[g_1])) \quad \text{where } U = vars(g[g_1], u) \text{ and } V \notin U.$$

When referring to goal variables, laws 2.1 and 2.2 will also be called 'Introduction and elimination of goal equalities' and 'Rearrangement of goal equalities', respectively.

3. *Rearrangement of term equalities*:

$$P \vdash \forall\,(g \wedge (t_1 = t_2) \overset{\geqslant}{\longrightarrow} (t_1 = t_2) \wedge g)$$

4. *Clark Equality Theory* (also called CET, see Lloyd (1987)):

$$P \vdash \forall X\,(eq_1 \overset{=}{\longleftrightarrow} eq_2) \qquad \text{if CET} \vdash \forall X\ (\exists Y\ eq_1 \leftrightarrow \exists Z\ eq_2)$$

where: (i) eq_1 and eq_2 are goals constructed by using *true*, *false*, term equalities, conjunctions, and disjunctions, and (ii) $Y = (vars(eq_1) - X)$ and $Z = (vars(eq_2) - X)$.

Notice that, for some program P and for some goals g, g_1, g_2, and g_3, the following do *not* hold:

$$\begin{array}{l}
P \vdash \forall\,(true \longrightarrow true \vee g) \\
P \vdash \forall\,(false \longrightarrow g \wedge false) \\
P \vdash \forall\,((t_1 = t_2) \wedge g \longrightarrow g \wedge (t_1 = t_2)) \\
P \vdash \forall\,(g_1 \vee (g_2 \wedge g_3) \longrightarrow (g_1 \vee g_2) \wedge (g_1 \vee g_3)) \\
P \vdash \forall V\,(g_2[g_1] \longrightarrow g_2[G] \wedge (G = g_1)) \qquad \text{where } V = vars(g_2[g_1]) \text{ and } G \notin V \\
P \vdash \forall V\,(g[(G = g_1) \wedge g_2] \longrightarrow (G = g_1) \wedge g[g_2]) \\
\qquad \text{where } V = (vars(g[g_2], g_1) - \{G\}) \text{ and } G \in vars(g[\_], g_1) \\
P \vdash \forall\,(g[(G = g_1) \wedge g_2] \longrightarrow (G = g_1) \wedge g[g_2]) \qquad \text{where } G \notin vars(g[\_], g_1)
\end{array}$$

Let us now make some remarks on the goal replacement rule.

In the Weak Correctness part of Theorem 2 (see Section 6) we shall prove that if program P_2 is derived from program P_1 by an application of the goal replacement rule based on a weak replacement law, then P_2 is a refinement of P_1, that is, $P_1 \sqsubseteq P_2$. Thus, there may be some ordinary goal g which either succeeds or fails in P_2, while g does not terminate in P_1, as shown by the following example.

Example 7

Let us consider the following two programs P_1 and P_2, where P_2 is derived from P_1 by applying the goal replacement rule based on the weak (and not strong) replacement law $P_1 \vdash \forall(\mathit{true} \vee g \xrightarrow{\geqslant} \mathit{true})$:

$$\begin{array}{llll} P_1: & p \leftarrow \mathit{true} \vee q & P_2: & p \leftarrow \mathit{true} \\ & q \leftarrow q & & q \leftarrow q \end{array}$$

We have that p does not terminate in P_1 and p succeeds in P_2.

Next, let us consider the following programs:

$$\begin{array}{llll} P_3: & p \leftarrow q \wedge \mathit{false} & P_4: & p \leftarrow \mathit{false} \\ & q \leftarrow q & & q \leftarrow q \end{array}$$

where P_4 is derived from P_3 by a goal replacement rule based on a weak (and not strong) replacement law $P \vdash \forall(g \wedge \mathit{false} \xrightarrow{\geqslant} \mathit{false})$. We have that p does not terminate in P_3, while p fails in P_4. □

In the Strong Correctness part of Theorem 2 we will prove that if program P_2 is derived from program P_1 by an application of the goal replacement rule based on a strong replacement law, then P_1 and P_2 are equivalent, that is $P_1 \equiv P_2$. Thus, in particular, for any goal g, g terminates in P_1 iff g terminates in P_2.

Moreover, in Theorem 3 of Section 6 we will prove that if program P_2 is derived from program P_1 by goal replacements which preserve safety, then every goal which is safe in P_1, is safe also in P_2.

6 Correctness of program transformations

The unrestricted use of our rules for transforming programs may allow the construction of incorrect transformation sequences, as the following example shows.

Example 8

Let us consider the following initial program:

$$\begin{array}{ll} P_0: & p \leftarrow q \\ & q \leftarrow \end{array}$$

By two definition introduction steps, we get:

$$\begin{array}{ll} P_1: & p \leftarrow q \\ & q \leftarrow \\ & \mathit{newp}1 \leftarrow q \\ & \mathit{newp}2 \leftarrow q \end{array}$$

By three folding steps, from program P_1 we get the final program:

$$\begin{array}{ll} P_2: & p \leftarrow \mathit{newp}1 \\ & q \leftarrow \\ & \mathit{newp}1 \leftarrow \mathit{newp}2 \\ & \mathit{newp}2 \leftarrow \mathit{newp}1 \end{array}$$

We have that p succeeds in P_0, while p does not terminate in P_2. □

In this section we present some conditions which ensure that every transformation sequence $P_0, \ldots, P_k$ constructed by using our rules, is:

(i) weakly correct, in the sense that $P_0 \cup Def_k \sqsubseteq P_k$ (see Point (1) of Theorem 2),
(ii) strongly correct, in the sense that $P_0 \cup Def_k \equiv P_k$ (see Point (2) of Theorem 2),
(iii) preserves safety, in the sense that, for every goal g, if g is safe in $P_0 \cup Def_k$ then g is safe also in P_k (see Theorem 3).

Similar to other correctness results presented in the literature (Bossi and Cocco 1994; Pettorossi and Proietti 1994; Sands 1996; Tamaki and Sato 1984), some of the conditions which ensure (weak or strong) correctness, require that the transformation sequences are constructed by performing suitable unfolding steps before performing folding steps.

In particular, Theorem 2 below ensures the (weak or strong) correctness of a given transformation sequence in the case where this sequence is *admissible*, that is, it is constructed by performing *parallel leftmost* unfoldings (see Definition 5) on all definitions which are used for performing subsequent foldings.

To present our correctness results it is convenient to consider admissible transformation sequences which are *ordered*, that is, transformation sequences constructed by:

- first, applying the definition introduction rule,
- then, performing parallel leftmost unfoldings of the definitions that are used for subsequent foldings, and
- finally, performing unfoldings, foldings, and goal replacements in any order.

Thus, an ordered, admissible transformation sequence has all its definition introductions performed at the beginning, and it can be written in the form $P_0, \ldots, P_0 \cup Def_k, \ldots, P_k$, where Def_k is the set of all definitions introduced during the entire transformation sequence $P_0, \ldots, P_0 \cup Def_k, \ldots, P_k$. By Proposition 3 below we may assume, without loss of generality, that all admissible transformation sequences are ordered.

To prove that an admissible transformation sequence is weakly correct (see Point (1) of Theorem 2), we proceed as follows.

(i) In Lemma 1 we consider a generic transformation by which we derive a program *NewP* from a program P by replacing the bodies of the clauses of P by new bodies. We show that, if these body replacements can be viewed as goal replacements based on weak replacement laws, then the transformation from P to *NewP* preserves successes and failures, that is,
   - if a goal g succeeds in P then g succeeds in *NewP*, and
   - if a goal g fails in P then g fails in *NewP*.

(ii) Then, in Lemma 2 we prove that in an ordered, admissible transformation sequence $P_0, \ldots, P_0 \cup Def_k, \ldots, P_k$, any application of the unfolding, folding, and goal replacement rule is an instance of the generic transformation considered in Lemma 1, that is, it consists in the replacement of the body of a clause by a new body, and this replacement can be viewed as a goal replacement based on a weak replacement law.

(iii) Thus, by using Lemmata 1 and 2 we get Point (1) of Theorem 1. In particular, we have that in any admissible transformation sequence $P_0, \ldots, P_0 \cup Def_k, \ldots, P_k$, successes and failures are preserved, i.e.:
   - if a goal g succeeds in $P_0 \cup Def_k$ then g succeeds in P_k, and
   - if a goal g fails in $P_0 \cup Def_k$ then g fails in P_k.

(iv) Finally, Proposition 1 allows us to infer the preservation of most general answer substitutions from the preservation of successes and failures. Indeed, by Proposition 1 and Point (1) of Theorem 1 we prove that if an ordinary goal g succeeds in $P_0 \cup Def_k$ then the set of answer substitutions for g in $P_0 \cup Def_k$ and the set of answer substitutions for g in P_k are equally general.

According to Definition 1, Points (iii) and (iv) mean that $P_0 \cup Def_k \sqsubseteq P_k$, that is, the ordered, admissible transformation sequence $P_0, \ldots, P_0 \cup Def_k, \ldots, P_k$ is weakly correct (see Point (1) of Theorem 2).

To prove that an admissible transformation sequence is strongly correct (see Point (2) of Theorem 2), we make the additional hypothesis that all goal replacements performed during the construction of the transformation sequence are based on strong replacement laws. Analogously to the proof of weak correctness which is based on Lemmata 1 and 2, the proof of strong correctness is based on Lemmata 3 and 4 which we give below. By using these lemmata, we prove Point (2) of Theorem 1, that is:

- if a goal g succeeds in P_k then g succeeds in $P_0 \cup Def_k$, and
- if a goal g fails in P_k then g fails in $P_0 \cup Def_k$.

Finally, by Proposition 1 and Theorem 1, we prove that any admissible transformation sequence in which all goal replacements are based on strong replacement laws, is strongly correct (see Point (2) of Theorem 2), that is, $P_0 \cup Def_k \equiv P_k$.

Now let us formally define the notions of *parallel leftmost unfolding* of a clause, *admissible* transformation sequence, and *ordered* admissible transformation sequence as follows.

Definition 5
Let c be a clause in a program P. If c is of the form:

$$p(V_1, \ldots, V_m) \leftarrow (a_1 \wedge g_1) \vee \ldots \vee (a_s \wedge g_s)$$

where $a_1, \ldots, a_s$ are atoms with non-primitive predicates, $g_1, \ldots, g_s$ are goals, and $s > 0$, then the *parallel leftmost unfolding* of clause c in program P is the program Q obtained from P by applying s times the unfolding rule w.r.t. $a_1, \ldots, a_s$, respectively.

If clause c is not of the form indicated in Definition 5 above, then the parallel leftmost unfolding of c is not defined.

Definition 6
A transformation sequence $P_0, \ldots, P_k$ is said to be *admissible* iff for every h, with $0 \leqslant h < k$, if P_{h+1} has been obtained from P_h by folding clause c using clause d, then there exist i, j, with $0 \leqslant i < j \leqslant k$, such that $d \in P_i$ and P_j is obtained from P_i by parallel leftmost unfolding of d.

Definition 7
An admissible transformation sequence $P_0,\ldots,P_k$ is said to be *ordered* iff it is of the form $P_0,\ldots,P_i,\ldots,P_j,\ldots,P_k$, where: (i) the sequence $P_0,\ldots,P_i$ is constructed by applying the definition introduction rule, (ii) the sequence $P_i,\ldots,P_j$ is constructed by parallel leftmost unfolding of all definitions which have been introduced during the sequence $P_0,\ldots,P_i$ and are used for folding during the sequence $P_j,\ldots,P_k$, and (iii) the definition introduction rule is never applied in the sequence $P_j,\ldots,P_k$.

Given an ordered, admissible transformation sequence $P_0,\ldots,P_i,\ldots,P_j,\ldots,P_k$, the set of definitions introduced during $P_0,\ldots,P_i$ is the same as the set of definitions introduced during the entire sequence $P_0,\ldots,P_k$, and thus, in the above Definition 7 we have that P_i is $P_0 \cup Def_k$.

An admissible transformation sequence $P_0,\ldots,P_k$ which is ordered, is also denoted by $P_0,\ldots,P_i,\ldots,P_j,\ldots,P_k$, where we explicitly indicate the program P_i after the introduction of the definitions, and the program P_j after the parallel leftmost unfolding steps.

Proposition 3
For any admissible transformation sequence $P_0,\ldots,P_n$ there exists an ordered, admissible transformation sequence $P_0,\ldots,P_i,\ldots,\ P_j,\ldots,P_k$ such that $P_n = P_k$ and $Def_n = Def_k$.

Now, to prove the correctness of transformation sequences, we state the following Lemmata 1, 2, 3, and 4, whose proofs are given in Pettorossi and Proietti (2003). As already mentioned, Lemmata 1–4 will allow us to show that, under suitable conditions, for every admissible transformation sequence $P_0,\ldots,P_k$, (i) successes and failures are preserved (see Theorem 1 below), and (ii) weak correctness holds (that is, $P_0 \cup Def_k \sqsubseteq P_k$) or strong correctness holds (that is, $P_0 \cup Def_k \equiv P_k$) (see Theorem 2 below).

Lemma 1
Let P and $NewP$ be programs of the form:

$$\begin{array}{llcll} P: & hd_1 \leftarrow bd_1 & \qquad & NewP: & hd_1 \leftarrow newbd_1 \\ & \vdots & & & \vdots \\ & hd_s \leftarrow bd_s & & & hd_s \leftarrow newbd_s \end{array}$$

For $r = 1,\ldots,s$, let V_r be $vars(hd_r)$ and suppose that $P \vdash \forall V_r\,(bd_r \overset{\geqslant}{\longrightarrow} newbd_r)$. Then, for every goal g and for every $b \in \{true, false\}$, we have that:

if $P \vdash g \downarrow_m b$ then $NewP \vdash g \downarrow_n b$ with $m \geqslant n$.

Lemma 2
Let us consider an ordered, admissible transformation sequence $P_0,\ldots,\ P_i,\ldots,\ P_j,\ldots,P_k$, where P_i is $P_0 \cup Def_k$.

(i) For $h = i,\ldots,j{-}1$ and for any pair of clauses c_1: $hd \leftarrow bd$ in program P_h and c_2: $hd \leftarrow newbd$ in program P_{h+1}, such that c_2 is derived from c_1 by applying the unfolding rule, we have that:
$P_i \vdash \forall V\,(bd \overset{\geqslant}{\longrightarrow} newbd)$
where $V = vars(hd)$. (Notice that the unfolding rule does not change the heads of the clauses.)

(ii) For $h = j, \ldots, k-1$ and for any pair of clauses c_1: $hd \leftarrow bd$ in program P_h and c_2: $hd \leftarrow newbd$ in program P_{h+1}, such that c_2 is derived from c_1 by applying the unfolding, or folding, or goal replacement rule, we have that:
$P_j \vdash \forall V\,(bd \overset{\geqslant}{\longrightarrow} newbd)$
where $V = vars(hd)$. (Notice that the unfolding, folding, and goal replacement rules do not change the heads of the clauses.)

Lemma 3
Let P and $NewP$ be programs of the form:

$$\begin{array}{llcll} P: & hd_1 \leftarrow bd_1 & \qquad & NewP: & hd_1 \leftarrow newbd_1 \\ & \vdots & & & \vdots \\ & hd_s \leftarrow bd_s & & & hd_s \leftarrow newbd_s \end{array}$$

For $r = 1, \ldots, s$, let V_r be $vars(hd_r)$ and suppose that $P \vdash \forall V_r\,(newbd_r \longrightarrow bd_r)$.
Then, for every goal g and for every $b \in \{true, false\}$, we have that if $NewP \vdash g \downarrow b$ then $P \vdash g \downarrow b$.

Notice that Lemma 3 is a partial converse of Lemma 1. These two lemmata imply that if we derive a program $NewP$ from a program P by replacing the bodies of the clauses of P by new bodies, and these body replacements are goal replacements based on strong replacement laws, then every goal terminates in $NewP$ iff it terminates in P.

Lemma 4
Let us consider a transformation sequence $P_0, \ldots, P_k$ and let Def_k be the set of definitions introduced during that sequence. For $h = 0, \ldots, k-1$ and for any pair of clauses c_1: $hd \leftarrow bd$ in program P_h and c_2: $hd \leftarrow newbd$ in program P_{h+1}, such that c_2 is derived from c_1 by applying the unfolding rule, or the folding rule, or the goal replacement rule based on strong replacement laws, we have that:
$P_0 \cup Def_k \vdash \forall V\,(newbd \longrightarrow bd)$
where $V = vars(hd)$.

In particular, as a consequence of Lemma 2 and Lemma 4, we have that in any ordered, admissible transformation sequence the unfolding and folding rules can be viewed as goal replacements based on strong replacement laws.

The following theorem states that for every admissible transformation sequence successes and failures are preserved.

Theorem 1 (*Preservation of Successes and Failures*)
Let $P_0, \ldots, P_k$ be an admissible transformation sequence and let Def_k be the set of definitions introduced during that sequence. Then for every goal g and for every $b \in \{true, false\}$, we have that:

1. if $P_0 \cup Def_k \vdash g \downarrow_m b$ then $P_k \vdash g \downarrow_n b$ with $m \geqslant n$, and
2. if all applications of the goal replacement rule are based on strong replacement laws and $P_k \vdash g \downarrow b$, then $P_0 \cup Def_k \vdash g \downarrow b$.

Proof
By Proposition 3, without loss of generality we may assume that the admissible sequence $P_0, \ldots, P_k$ is ordered. Let P_j be the program obtained at the end of the second subsequence of $P_0, \ldots, P_k$, that is, after unfolding every clause in Def_k which is used for folding. Point (1) of this theorem is a consequence of the following two facts:

(F1) By Lemma 1 and Point (i) of Lemma 2, we have that, for every goal g and for every $b \in \{true, false\}$, if $P_0 \cup Def_k \vdash g \downarrow_m b$ then $P_j \vdash g \downarrow_{n1} b$ with $m \geqslant n1$.

(F2) By Lemma 1 and Point (ii) of Lemma 2, we have that: for every goal g and for every $b \in \{true, false\}$, if $P_j \vdash g \downarrow_{n1} b$ then $P_k \vdash g \downarrow_n b$ with $n1 \geqslant n$.

Point (2) of this theorem is a straightforward consequence of Lemmata 3 and 4. □

The following theorem establishes the weak correctness and, under suitable conditions, the strong correctness of admissible transformation sequences.

Theorem 2 (*Correctness Theorem*)
Let $P_0, \ldots, P_k$ be an admissible transformation sequence. Let Def_k be the set of definitions introduced during that sequence. We have that:

1. (Weak Correctness) $P_0 \cup Def_k \sqsubseteq P_k$, that is, P_k is a refinement of $P_0 \cup Def_k$.
2. (Strong Correctness) if all applications of the goal replacement rule are based on strong replacement laws then $P_0 \cup Def_k \equiv P_k$, that is, P_k is equivalent to $P_0 \cup Def_k$.

Proof

1. First we prove that $P_0 \cup Def_k \sqsubseteq P_k$. Let g be an ordinary goal and let A be a set of substitutions such that $P_0 \cup Def_k \vdash g \mapsto A$. We have to prove that there exists $B \in \mathscr{P}(Subst)$ such that $P_k \vdash g \mapsto B$ and A and B are equally general with respect to g.
Since $P_0 \cup Def_k \vdash g \mapsto A$, by definition there exists $b \in \{true, false\}$ such that $P_0 \cup Def_k \vdash g \downarrow b$. By Point (1) of Theorem 1, we have that $P_k \vdash g \downarrow b$ and, thus, there exists $B \in \mathscr{P}(Subst)$ such that $P_k \vdash g \mapsto B$.
To prove that A and B are equally general with respect to g, we have to show that: (a) for every substitution $\alpha \in A$ there exists a substitution $\beta \in B$ such that $g\alpha$ is an instance of $g\beta$, and (b) for every $\beta \in B$ there exists $\alpha \in A$ such that $g\beta$ is an instance of $g\alpha$.
   (a) Let α be a substitution in A. From $P_0 \cup Def_k \vdash g \mapsto A$, by Proposition 1 (ii.1), we have that $P_0 \cup Def_k \vdash g\alpha \downarrow true$. Thus, by Point (1) of Theorem 1, we have that $P_k \vdash g\alpha \downarrow true$. Since $P_k \vdash g \mapsto B$ holds, by Proposition 1 (ii.1), there exists a substitution $\beta \in B$ such that $g\alpha$ is an instance of $g\beta$.
   (b) Let β be a substitution in B. From $P_k \vdash g \mapsto B$, by Proposition 1 (ii.1), we have that $P_k \vdash g\beta \downarrow true$. From $P_0 \cup Def_k \vdash g \mapsto A$, by Proposition 1 (i), we have that *either* $P_0 \cup Def_k \vdash g\beta \downarrow true$ *or* $P_0 \cup Def_k \vdash g\beta \downarrow false$. Now

$P_0 \cup Def_k \vdash g\beta \downarrow false$ is impossible because by Point (1) of Theorem 1, we would have $P_k \vdash g\beta \downarrow false$. Thus, $P_0 \cup Def_k \vdash g\beta \downarrow true$. Since $P_0 \cup Def_k \vdash g \mapsto A$, by Proposition 1 (ii.1), there exists $\alpha \in A$ such that $g\beta$ is an instance of $g\alpha$.

- We have to prove that if all applications of the goal replacement rule in the sequence $P_0, \ldots, P_k$ are based on strong replacement laws, then $P_0 \cup Def_k \equiv P_k$. Since $P_0 \cup Def_k \sqsubseteq P_k$ has been shown at Point (1) of this proof, it remains to show that: $P_k \sqsubseteq P_0 \cup Def_k$. The proof is similar to that of Point (1) and it is based on Point (2) of Theorem 1 and Proposition 1 (ii.1).

□

The following two examples show that in the statement of Theorem 2 we cannot drop the admissibility condition. Indeed, in these examples we construct transformation sequences which are not admissible and not weakly correct.

Example 9
Let us construct a transformation sequence as follows. The initial program is:

$$
\begin{array}{ll}
P_0: & p \leftarrow p \wedge q \\
 & q \leftarrow false
\end{array}
$$

By definition introduction we get:

$$
\begin{array}{ll}
P_1: & p \leftarrow p \wedge q \\
 & q \leftarrow false \\
 & newp \leftarrow false \wedge p
\end{array}
$$

Then we perform the unfolding of $newp \leftarrow false \wedge p$ w.r.t. p. (Notice that this is not a parallel leftmost unfolding.) We get:

$$
\begin{array}{ll}
P_2: & p \leftarrow p \wedge q \\
 & q \leftarrow false \\
 & newp \leftarrow false \wedge p \wedge q
\end{array}
$$

By folding we get the final program:

$$
\begin{array}{ll}
P_3: & p \leftarrow p \wedge q \\
 & q \leftarrow false \\
 & newp \leftarrow newp \wedge q
\end{array}
$$

We have that $newp$ fails in $P_0 \cup Def_3$ (that is, P_1), while $newp$ does not terminate in P_3. □

Example 10
Let us construct a transformation sequence as follows. The initial program is:

$$
\begin{array}{ll}
P_0: & p \leftarrow false \\
 & q \leftarrow true \vee q
\end{array}
$$

By definition introduction we get:

$$
\begin{array}{ll}
P_1: & p \leftarrow false \\
 & q \leftarrow true \vee q \\
 & newp \leftarrow p \vee (p \wedge q)
\end{array}
$$

Then we perform the unfolding of $newp \leftarrow p \vee (p \wedge q)$ w.r.t. q. (Notice that this is not a parallel leftmost unfolding.) We get:

$$\begin{aligned} P_2: \quad & p \leftarrow false \\ & q \leftarrow true \vee q \\ & newp \leftarrow false \vee (p \wedge (true \vee q)) \end{aligned}$$

By goal replacement based on boolean laws we get:

$$\begin{aligned} P_3: \quad & p \leftarrow false \\ & q \leftarrow true \vee q \\ & newp \leftarrow p \vee (p \wedge q) \end{aligned}$$

By folding we get the final program:

$$\begin{aligned} P_4: \quad & p \leftarrow false \\ & q \leftarrow true \vee q \\ & newp \leftarrow newp \end{aligned}$$

We have that *newp* fails in $P_0 \cup Def_4$ (that is, P_1), while *newp* does not terminate in P_4. □

Finally, the following theorem, whose proof is given in Pettorossi and Proietti (2003), states that a (possibly not admissible) transformation sequence preserves safety, if all goal replacements performed during that sequence preserve safety.

Theorem 3 (*Preservation of Safety*)
Let $P_0, \ldots, P_k$ be a transformation sequence and let Def_k be the set of definitions introduced during that sequence. Let us also assume that all applications of the goal replacement rule R4 preserve safety. Then, for every goal g, if g is safe in $P_0 \cup Def_k$ then g is safe in P_k.

We end this section by making some comments about our correctness results. Let us consider an admissible transformation sequence $P_0, \ldots, P_k$, during which we introduce the set Def_k of definitions. Then, by Point (1) of Theorem 1 program P_k may be *more defined* than program $P_0 \cup Def_k$ in the sense that there may be a goal which terminates (i.e. succeeds or fails) in P_k, while it does not terminate in $P_0 \cup Def_k$. This 'increase of termination' is often desirable when transforming programs and it may be achieved by goal replacements which are not based on strong replacement laws (see, for instance, Example 7 in Section 5).

Now suppose that during the construction of the admissible transformation sequence $P_0, \ldots, P_k$ all applications of the goal replacement rule are based on strong replacement laws. Then, by Theorem 1 we have that for all goals g, g terminates in $P_0 \cup Def_k$ iff g terminates in P_k. However, safety may be not preserved, in the sense that there may be a goal g which is safe in $P_0 \cup Def_k$ (but g neither succeeds nor fails in $P_0 \cup Def_k$) and g is not safe in P_k (or vice versa), as shown by the following example.

Example 11
Let us consider the following two programs P_1 and P_2:

$$P_1: \quad p \leftarrow p \qquad\qquad P_2: \quad p \leftarrow G$$

Program P_2 is derived from P_1 by applying the goal replacement rule based on the strong replacement law $P_1 \vdash p \overset{=}{\longleftrightarrow} G$, which does not preserve safety. We have that p is safe, p does not terminate in P_1, and p is not safe in P_2. Notice that the replacement law $P_1 \vdash p \overset{=}{\longleftrightarrow} G$ trivially holds because, for any $b \in \{true, false\}$, $P_1 \vdash p \downarrow b$ does not hold and $P_1 \vdash G \downarrow b$ does not hold. □

To ensure that if g is safe in P_1 then g is safe in P_2, it is enough to use replacement laws which preserve safety (see Theorem 3). Indeed, unfolding and folding always preserve safety (Pettorossi and Proietti 2003).

We have not presented any result which guarantees that if a goal is safe in the final program P_k then it is safe in the program $P_0 \cup Def_k$. This result could have been achieved by imposing further restrictions on the goal replacement rule. However, we believe that this 'inverse preservation of safety' is not important in practice, because usually we start from an initial program where all goals of interest are safe and we want to derive a final program where those goals of interest are still safe. In particular, if in the transformation sequence $P_0, \ldots, P_k$ the initial program P_0 is an ordinary program, then every ordinary goal g is safe in P_0 and, by Theorem 3, we have that g is safe also in P_k. Thus, as discussed in Section 4, we can use ordinary implementations of LD-resolution to compute the relation $P_k \models g \mapsto A$.

Notice also that, if $P_0 \cup Def_k \sqsubseteq P_k$ and an ordinary goal g terminates in P_0, then g has the same most general answer substitutions in $P_0 \cup Def_k$ and P_k, modulo variable renaming (see Point (i) of Remark 1 at the end of Section 4). However, the set of *all* answer substitutions may not be preserved, and in particular, there are programs P_1 and P_2 such that $P_1 \sqsubseteq P_2$ and, for some goal g, we have that $P_1 \vdash g \mapsto A_1$ and $P_2 \vdash g \mapsto A_2$, where A_1 and A_2 have different cardinality, as shown by the following example adapted from Bossi *et al.* (1992). A similar property holds if we assume that $P_1 \equiv P_2$, instead of $P_1 \sqsubseteq P_2$.

Example 12
Let us consider the following two programs P_1 and P_2, where P_2 is derived from P_1 by applying the goal replacement rule based on the weak replacement law $P \vdash \forall(g \wedge g \overset{\geqslant}{\longrightarrow} g)$, which holds for every program P and and goal g:

$$\begin{array}{llll}
P_1: & p(X) \leftarrow q(X) \wedge q(X) & P_2: & p(X) \leftarrow q(X) \\
 & q(X) \leftarrow X = f(a,Z) & & q(X) \leftarrow X = f(a,Z) \\
 & q(X) \leftarrow X = f(Y,a) & & q(X) \leftarrow X = f(Y,a)
\end{array}$$

We have that:

$P_1 \vdash p(X) \mapsto \{\{X/f(a,Z)\}, \{X/f(a,a)\}, \{X/f(Y,a)\}\}$, and
$P_2 \vdash p(X) \mapsto \{\{X/f(a,Z)\}, \{X/f(Y,a)\}\}$. □

The above example shows that, if during program transformation we want to preserve the set of answer substitutions, then we should not apply goal replacements based on the replacement law $P \vdash \forall(g \wedge g \overset{\geqslant}{\longrightarrow} g)$ which, however, may be useful for avoiding the computation of redundant goals and improving program efficiency.

Another replacement law which is very useful in many examples of program transformation, is the law which expresses the functionality of a predicate. For instance, in the *Deepest* example of Section 2, the *depth* predicate is functional with respect to its first argument in the sense that, for every goal context $g[\_]$, the following replacement law holds:

$$Deepest \vdash \forall\,(depth(T,X) \wedge g[depth(T,Y)] \overset{\geqslant}{\longleftrightarrow} depth(T,X) \wedge g[X=Y]).$$

The following example, similar to Example 12, shows that in general the functionality law does not preserve the set of answer substitutions.

Example 13

Let us consider the following two programs P_1 and P_2, where P_2 is derived from P_1 by applying the goal replacement rule based on the (strong) replacement law $P_1 \vdash \forall\,(q(X,Y) \wedge q(X,Z) \overset{\geqslant}{\longleftrightarrow} q(X,Y) \wedge Y=Z)$:

$$\begin{array}{llll}
P_1: & p(X) \leftarrow q(X,Y) \wedge q(X,Z) & P_2: & p(X) \leftarrow q(X,Y) \wedge Y=Z \\
 & q(f(a,Z),b) \leftarrow & & q(f(a,Z),b) \leftarrow \\
 & q(f(Y,a),b) \leftarrow & & q(f(Y,a),b) \leftarrow
\end{array}$$

As in Example 12, we have that:

$P_1 \vdash p(X) \mapsto \{\{X/f(a,Z)\},\{X/f(a,a)\},\{X/f(Y,a)\}\}$ and
$P_2 \vdash p(X) \mapsto \{\{X/f(a,Z)\},\{X/f(Y,a)\}\}$. □

Finally, notice that Theorem 2 ensures the preservation of most general answer substitutions for ordinary goals only. Thus, the answer substitutions computed for goals with occurrences of goal variables, may not be preserved, as shown by the following example.

Example 14

Let us consider the following two programs P_1 and P_2, where P_2 is derived from P_1 by unfolding clause 1 w.r.t. p using clause 2:

$$\begin{array}{llllll}
P_1: & 1. & a(G) \leftarrow (G=p) \wedge G & P_2: & 1^*. & a(G) \leftarrow (G=q) \wedge G \\
 & 2. & p \leftarrow q & & 2. & p \leftarrow q \\
 & 3. & q \leftarrow & & 3. & q \leftarrow
\end{array}$$

We have that $P_1 \vdash a(G) \mapsto \{\{G/p\}\}$, and $P_2 \vdash a(G) \mapsto \{\{G/q\}\}$. □

7 Program derivation in the extended language

In this section we present some examples which illustrate the use of our transformation rules. In these examples, by using goal variables and goal arguments, we introduce and manipulate *continuations*. For this reason we have measured the improvements of program efficiency by running our programs using the BinProlog continuation passing compiler (Tarau 1996). These run-time improvements have been reported in Section 7.6. Compilers based on different implementation methodologies, such as SICStus Prolog, may not give the same improvements. However, it should be noticed that the efficiency improvements we get, do not come from the use of continuations, but from the program transformations performed by applying our transformation rules (see Section 5). Indeed, in BinProlog the continuation passing style transformation in itself gives no speed-ups.

Let us introduce the following terminology which will be useful in the sequel. We say that: (i) a clause is in continuation passing style iff its body has no occurrences of the conjunction operator, and (ii) a program is in continuation passing style iff all its clauses are in continuation passing style. Thus, every program in continuation passing style is a *binary* program in the sense of Tarau and Boyer (1990), i.e. a program with at most one atom in the body of its clauses.

When writing programs in this section we use the following primitive predicates: $=$, $\neq$, $\geqslant$, and $<$. For the derivation of programs in continuation passing style, we assume that, for each of these predicates there exists a corresponding primitive predicate with an extra argument denoting a continuation. Let us call these predicates *eq_c*, *diff_c*, *geq_c*, and *lt_c*, respectively.

We assume that, for every program P, the following strong replacement laws hold:

$$P \vdash \forall((X=Y) \wedge C \overset{=}{\longleftrightarrow} eq\_c(X,Y,C))$$
$$P \vdash \forall((M \neq N) \wedge C \overset{=}{\longleftrightarrow} diff\_c(M,N,C))$$
$$P \vdash \forall((M \geqslant N) \wedge C \overset{=}{\longleftrightarrow} geq\_c(M,N,C))$$
$$P \vdash \forall((M < N) \wedge C \overset{=}{\longleftrightarrow} lt\_c(M,N,C))$$

In this section we use the following syntactical conventions:

1. the conjunction operator $\wedge$ is replaced by comma,
2. a clause of the form $h \leftarrow g_1 \vee g_2$ is also written as two clauses, namely, $h \leftarrow g_1$ and $h \leftarrow g_2$, and
3. a clause of the form $h \leftarrow (V=u), g$ where the variable V does not occur in the argument u, is also written as $(h \leftarrow g)\{V/u\}$.

7.1 Tree flipping

This example is borrowed from Jørgensen *et al.* (1997), where it is used for showing that *conjunctive partial deduction* may affect program termination when transforming programs for eliminating multiple traversals of data structures. A similar problem arises when multiple traversals of data structures are avoided by applying Tamaki and Sato's unfold/fold transformation rules (Tamaki and Sato 1984) according to the tupling strategy (see Section 2). In this example by using goal arguments and introducing continuations, we are able to derive a program in continuation passing style which eliminates multiple traversals of data structures and, at the same time, preserves universal termination.

Let us consider the initial program *FlipCheck*:

1. $flipcheck(X,Y) \leftarrow flip(X,Y),\ check(Y)$
2. $flip(l(N),l(N)) \leftarrow$
3. $flip(t(L,N,R),t(FR,N,FL)) \leftarrow flip(L,FL),\ flip(R,FR)$
4. $check(l(N)) \leftarrow nat(N)$
5. $check(t(L,N,R)) \leftarrow nat(N),\ check(L),\ check(R)$
6. $nat(0) \leftarrow$
7. $nat(s(N)) \leftarrow nat(N)$

where (i) the term $l(N)$ denotes a leaf with label N and the term $t(L,N,R)$ denotes a tree with label N and the two subtrees L and R, (ii) $nat(X)$ holds iff X is a natural number, (iii) *check*(X) holds iff all labels in the tree X are natural numbers, and (iv) *flip*(X, Y) holds iff the tree Y can be obtained by flipping all subtrees of the tree X.

We would like to transform this program so to avoid the double traversal of trees (see the double occurrence of Y in the body of clause 1). By applying the tupling strategy (or, equivalently, *conjunctive partial deduction*), we derive the following program *FlipCheck*1:

8. $\mathit{flipcheck}(l(N), l(N)) \leftarrow \mathit{nat}(N)$
9. $\mathit{flipcheck}(t(L, N, R), t(FR, N, FL)) \leftarrow \mathit{nat}(N),$
   $\mathit{flipcheck}(L, FL),\ \mathit{flipcheck}(R, FR)$

Program *FlipCheck*1 performs only one traversal of any input tree which is the first argument of *flipcheck*. However, as already mentioned, *FlipCheck*1 does not preserve termination. Indeed, the goal $\mathit{flipcheck}(t(l(N), 0, l(a)), Y)$ fails in *FlipCheck*, while this goal does not terminate in the derived program *FlipCheck*1.

Now we present a second derivation starting from the same program *FlipCheck* and producing a final program *FlipCheck*2 which: (i) is in continuation passing style, (ii) traverses the input tree only once, and (iii) preserves termination. During this second derivation we introduce goal arguments and we make use of the transformation rules introduced in Section 5. The initial step of this derivation is the introduction of the following new clause:

10. $\mathit{newp}(X, Y, G, C, D) \leftarrow \mathit{flip}(X, Y),\ G = (\mathit{check}(Y), C),\ D$

As already mentioned, in this paper we do not illustrate the strategies needed for guiding the application of our transformation rules and, in particular, we do not indicate how to construct the new definitions to be introduced, such as clause 10 above. For clause 10 we notice that: (i) by introducing a definition with the goal equality $G = (\mathit{check}(Y), C)$, instead of the goal $\mathit{check}(Y)$, we will be able to apply the folding rule by first performing leftward moves of goal equalities, instead of (possibly incorrect) leftward moves of goals, and (ii) by introducing the continuations C and D, we will avoid the expensive use of the conjunction operator for constructing goal arguments.

We continue our derivation by unfolding clause 10 w.r.t. $\mathit{flip}(X, Y)$ and we get:

11. $\mathit{newp}(l(N), l(N), G, C, D) \leftarrow (G = (\mathit{check}(l(N)), C)),\ D$
12. $\mathit{newp}(t(L, N, R), t(FR, N, FL), G, C, D) \leftarrow \mathit{flip}(L, FL),\ \mathit{flip}(R, FR)$
    $(G = (\mathit{check}(t(FR, N, FL)), C)),\ D$

We then unfold clauses 11 and 12 w.r.t. the *check* atoms, and after some applications of the goal replacement rule based on boolean laws and CET, we get:

13. $\mathit{newp}(l(N), l(N), G, C, D) \leftarrow G = (\mathit{nat}(N), C),\ D$
14. $\mathit{newp}(t(L, N, R), t(FR, N, FL), G, C, D) \leftarrow \mathit{flip}(L, FL),\ \mathit{flip}(R, FR),$
    $(G = (\mathit{nat}(N), \mathit{check}(FR), \mathit{check}(FL), C)),\ D$

By introducing and rearranging goal equalities (see laws 2.1 and 2.2, respectively, in Section 5), we transform clause 14 into:

15. $\mathit{newp}(t(L, N, R), t(FR, N, FL), G, C, D) \leftarrow \mathit{flip}(L, FL),\ U = (\mathit{check}(FL), C),$
    $\mathit{flip}(R, FR),\ V = (\mathit{check}(FR), U),\ \ (G = (\mathit{nat}(N), V)),\ D$

Now we fold twice clause 15 using clause 10 and we get:

16. $newp(t(L,N,R), t(FR,N,FL), G, C, D) \leftarrow$
    $newp(L, FL, U, C, newp(R, FR, V, U, (G = (nat(N), V), D)))$

To express *flipcheck* in terms of *newp* we introduce a goal equality into clause 1 and we derive:

17. $flipcheck(X,Y) \leftarrow flip(X,Y),\ G = (check(Y), true),\ G$

Then we fold clause 17 using clause 10 and we get:

18. $flipcheck(X,Y) \leftarrow newp(X,Y,G,true,G)$

The program we have derived so far consists of clauses 13, 16 and 18. Notice that clauses 13 and 16 are not in continuation passing style because the conjunction operator occurs in their bodies. In order to derive clauses in continuation passing style we introduce the following new definition:

19. $nat\_c(N,C) \leftarrow nat(N),\ C$

By unfolding, folding, and goal replacement steps based on the replacement law $FlipCheck \vdash \forall((X = Y), C \overset{\equiv}{\longleftrightarrow} eq\_c(X,Y,C))$, we derive the following final program *FlipCheck*2:

18. $flipcheck(X,Y) \leftarrow newp(X,Y,G,true,G)$
20. $newp(l(N), l(N), G, C, D) \leftarrow eq\_c(G, nat\_c(N,C), D)$
21. $newp(t(L,N,R), t(FR,N,FL), G, C, D) \leftarrow$
    $newp(L, FL, U, C,\ newp(R, FR, V, U,$
    $eq\_c(G, nat\_c(N,V), D)\ ))$
22. $nat\_c(0,C) \leftarrow C$
23. $nat\_c(s(N),C) \leftarrow nat\_c(N,C)$

Program *FlipCheck*2 traverses the input tree only once. Moreover, Theorem 1 ensures that, for every goal g of the form $flipcheck(t_1,t_2)$, where t_1 and t_2 are any two terms, g terminates in *FlipCheck* iff g terminates in *FlipCheck*2 (see also Section 7.5 for a more detailed discussion of the correctness properties of our program derivations).

7.2 Summing the leaves of a tree

Let us consider the following program *TreeSum* that, given a binary tree t whose leaves are labeled by natural numbers, computes the sum of the labels of the leaves of t.

1. $treesum(l(N), N) \leftarrow$
2. $treesum(t(L,R), N) \leftarrow treesum(L, NL),\ treesum(R, NR),\ plus(NL, NR, N)$
3. $plus(0, X, X) \leftarrow$
4. $plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z)$

By using Tamaki and Sato's transformation rules, from program *TreeSum* we may derive a more efficient program with *accumulator* arguments. In particular, during this program derivation we introduce the following new predicate:

5. $acc\_ts(T,Y,Z) \leftarrow treesum(T,X),\ plus(X,Y,Z)$

We also use the associativity of the predicate *plus*, that is, we use the following equivalence which holds in the least Herbrand model $M(\mathit{TreeSum})$ of the given program *TreeSum*:

$$M(\mathit{TreeSum}) \models \forall X1, X2, X3, S\, (\exists I\, (\mathit{plus}(X1,X2,I),\ \mathit{plus}(I,X3,S)) \leftrightarrow \\ \exists J\, (\mathit{plus}(X1,J,S),\ \mathit{plus}(X2,X3,J)))$$

During the derivation, we also make suitable goal rearrangements needed for performing foldings that use clause 5. We derive the following program *TreeSum*1.

6. $\mathit{treesum}(l(N),N) \leftarrow$
7. $\mathit{treesum}(t(L,R),N) \leftarrow \mathit{acc\_ts}(L,NR,N),\ \mathit{treesum}(R,NR)$
8. $\mathit{acc\_ts}(l(N),\mathit{Acc},Z) \leftarrow \mathit{plus}(N,\mathit{Acc},Z)$
9. $\mathit{acc\_ts}(t(L,R),\mathit{Acc},N) \leftarrow \mathit{acc\_ts}(L,\mathit{Acc},\mathit{NewAcc}),\ \mathit{acc\_ts}(R,\mathit{NewAcc},N)$

The least Herbrand models of programs *TreeSum* and *TreeSum*1 define the same relation for the predicate *treesum*. However, the two programs do not have the same termination behaviour. For instance, the goal $\mathit{treesum}(t(l(N),0),Z)$ fails in *TreeSum* while it does not terminate in *TreeSum*1.

By introducing goal arguments and using the transformation rules presented in Section 5, we are able to derive a program which: (i) is in continuation passing style, (ii) preserves termination, and (iii) is asymptotically more efficient than the original program *TreeSum*. Our derivation begins by introducing the following new clause:

10. $\mathit{gen\_ts}(T,Y,Z,G,C,D) \leftarrow \mathit{treesum}(T,X),\ (G{=}(\mathit{plus}(X,Y,Z),C)),\ D$

We unfold clause 10 and we get:

11. $\mathit{gen\_ts}(l(N),Y,Z,G,C,D) \leftarrow (G{=}(\mathit{plus}(N,Y,Z),C)),\ D$
12. $\mathit{gen\_ts}(t(L,R),Y,Z,G,C,D) \leftarrow \mathit{treesum}(L,LS),\ \mathit{treesum}(R,RS),$
    $\mathit{plus}(LS,RS,S),\ (G{=}(\mathit{plus}(S,Y,Z),C)),\ D$

Now we may exploit the following generalized associativity law for *plus*:

$$\mathit{TreeSum} \vdash \forall V\, ((\mathit{plus}(X1,X2,I),\ g[\mathit{plus}(I,X3,S)]) \overset{\geqslant}{\longleftrightarrow} \\ (\mathit{plus}(X1,J,S),\ g[\mathit{plus}(X2,X3,J)]))$$

where $V = \{X1,X2,X3,S\} \cup \mathit{vars}(g[\_])$ and $\{I,J\} \cap \mathit{vars}(g[\_]) = \emptyset$. By this law, from clause 12 we get the following clause:

13. $\mathit{gen\_ts}(t(L,R),Y,Z,G,C,D) \leftarrow \mathit{treesum}(L,LS),\ \mathit{treesum}(R,RS),$
    $\mathit{plus}(LS,S1,Z),\ (G{=}(\mathit{plus}(RS,Y,S1),C)),\ D$

By introducing and rearranging goal equalities (see laws 2.1 and 2.2 in Section 5), we transform clause 13 into:

14. $\mathit{gen\_ts}(t(L,R),Y,Z,G,C,D) \leftarrow$
    $\mathit{treesum}(L,LS),\ (GL{=}(\mathit{plus}(LS,S1,Z),\ G{=}GR,\ D)),$
    $\mathit{treesum}(R,RS),\ (GR{=}(\mathit{plus}(RS,Y,S1),C)),\ GL$

To derive clauses in continuation passing style we introduce the following new definitions:

15. $\mathit{ts\_c}(T,N,C) \leftarrow \mathit{treesum}(T,N),\ C$
16. $\mathit{plus\_c}(X,Y,Z,C) \leftarrow \mathit{plus}(X,Y,Z),\ C$

By unfolding clauses 15 and 16 we get:

17. $ts\_c(l(N), N, C) \leftarrow C$
18. $ts\_c(t(L,R), N, C) \leftarrow treesum(L, LN),\ treesum(R, RN),$
    $plus(LN, RN, N),\ C$
19. $plus\_c(0, X, X, C) \leftarrow C$
20. $plus\_c(s(X), Y, s(Z), C) \leftarrow plus(X, Y, Z),\ C$

By introducing and rearranging goal equalities, we transform clause 18 into:

21. $ts\_c(t(L,R), N, C) \leftarrow treesum(L, LN),\ (G = (plus(LN, RN, N), C)),$
    $treesum(R, RN),\ G$

By folding steps and goal replacements (based on, among others, the replacement law $TreeSum \vdash \forall((X = Y), C \overset{=}{\longleftrightarrow} eq\_c(X, Y, C)))$, we get the following final program *TreeSum*2:

22. $treesum(T, N) \leftarrow ts\_c(T, N, true)$
18. $ts\_c(l(N), N, C) \leftarrow C$
23. $ts\_c(t(L,R), N, C) \leftarrow gen\_ts(L, RN, N, G, C, ts\_c(R, RN, G))$
24. $gen\_ts(l(N), Y, Z, G, C, D) \leftarrow eq\_c(G, plus\_c(N, Y, Z, C), D)$
25. $gen\_ts(t(L,R), Y, Z, G, C, D) \leftarrow gen\_ts(L, S1, Z, GL, eq\_c(G, GR, D),$
    $gen\_ts(R, Y, S1, GR, C, GL))$
19. $plus\_c(0, X, X, C) \leftarrow C$
20. $plus\_c(s(X), Y, s(Z), C) \leftarrow plus\_c(X, Y, Z, C)$

This final program *TreeSum*2 is more efficient than *TreeSum*. Indeed, in the worst case, *TreeSum*2 takes $O(n)$ steps for solving a goal of the form $treesum(t, N)$, where t is a ground tree and $s^n(0)$ is the sum of the labels of the leaves of t, while the initial program *TreeSum* takes $O(n^2)$ steps. Moreover, by our Theorem 1 of Section 6, for every goal g of the form $treesum(t_1, t_2)$, where t_1 and t_2 are any terms, g terminates in *TreeSum* iff g terminates in *TreeSum*2 (see also Section 7.5).

7.3 Matching a regular expression

Let us consider the following matching problem: given a string S in $\{0,1,2\}^*$, we want to find the position N of an occurrence of a substring P of S such that P is generated by the regular expression 0^*1. The following program *RegExprMatch* computes such a position:

1. $match(S, N) \leftarrow pattern(S),\ N = 0$
2. $match([C|S], N) \leftarrow char(C),\ match(S, M),\ plus(s(0), M, N)$
3. $pattern([0|S]) \leftarrow pattern(S)$
4. $pattern([1|S]) \leftarrow$
5. $char(0) \leftarrow$
6. $char(1) \leftarrow$
7. $char(2) \leftarrow$
8. $plus(0, X, X) \leftarrow$
9. $plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z)$

If we assume the depth-first, left-to-right evaluation strategy of Prolog, the running time of this program *RegExprMatch* is $O(n^2)$ in the worst case, where n is the length of the input string. For a goal of the form *match*(s, N), where s is a ground string made out of n 0's, the program *RegExprMatch* performs one resolution step using clause 1 for the call to *match*, and then n resolution steps using clause 3 for the successive calls to *pattern*. When the computation backtracks, for the successive call of *match*$(s1, N)$, where $s1$ is the tail of s, the program *RegExprMatch* performs again $n-1$ resolution steps using clause 3.

By using the transformation rules of Section 5, we now present the derivation of a new program *RegExprMatch*1 which: (i) is in continuation passing style, (ii) preserves termination, and (iii) is asymptotically more efficient than the original program *RegExprMatch*. Indeed, program *RegExprMatch*1 avoids the redundant resolution steps performed by *RegExprMatch* using clause 3. For our derivation we introduce the following new predicates with goal arguments which are continuations:

10. $match\_c(S,N,C) \leftarrow match(S,N),\ C$
11. $newp(S,N,C1,C2) \leftarrow (pattern(S),\ C1)\ \vee\ (match(S,N),\ C2)$
12. $plus\_c(X,Y,Z,C) \leftarrow plus(X,Y,Z),\ C$

By unfolding clauses 10, 11, and 12 we get:

13. $match\_c([0|S],N,C) \leftarrow (pattern(S), N=0, C) \vee$
    $(match(S,M), plus(s(0),M,N), C)$
14. $match\_c([1|S],N,C) \leftarrow (N=0, C) \vee$
    $(match(S,M), plus(s(0),M,N), C)$
15. $match\_c([2|S],N,C) \leftarrow match(S,M), plus(s(0),M,N), C$
16. $newp([0|S],N,C1,C2) \leftarrow (pattern(S), C1) \vee$
    $(pattern(S), N=0, C2) \vee$
    $(match(S,M), plus(s(0),M,N), C2)$
17. $newp([1|S],N,C1,C2) \leftarrow C1 \vee$
    $(N=0, C2) \vee$
    $(match(S,M), plus(s(0),M,N), C2)$
18. $newp([2|S],N,C1,C2) \leftarrow match(S,M), plus(s(0),M,N), C2$
19. $plus\_c(0,X,X,C) \leftarrow C$
20. $plus\_c(s(X),Y,s(Z),C) \leftarrow plus(X,Y,Z),\ C$

By goal replacement using boolean laws, from clause 16 we get:

21. $newp([0|S],N,C1,C2) \leftarrow (pattern(S), (C1\ \vee\ (N=0, C2))) \vee$
    $(match(S,M), plus(s(0),M,N), C2)$

By performing folding and goal replacement steps (based on the replacement law *RegExprMatch* $\vdash \forall((X=Y), C \overset{\equiv}{\longleftrightarrow} eq\_c(X,Y,C))$ and other laws), we derive the following program *RegExprMatch*1:

22. $match(S,N) \leftarrow match\_c(S,N,true)$
23. $match\_c([0|S],N,C) \leftarrow newp(S,M,\ eq\_c(N,0,C),\ plus\_c(s(0),M,N,C))$
24. $match\_c([1|S],N,C) \leftarrow eq\_c(N,0,C)$
25. $match\_c([1|S],N,C) \leftarrow match\_c(S,M,\ plus\_c(s(0),M,N,C))$
26. $match\_c([2|S],N,C) \leftarrow match\_c(S,M,\ plus\_c(s(0),M,N,C))$

27. $newp([0|S], N, C1, C2) \leftarrow$
    $newp(S, M,\ (C1 \vee eq\_c(N, 0, C2)), plus\_c(s(0), M, N, C2))$
28. $newp([1|S], N, C1, C2) \leftarrow C1$
29. $newp([1|S], N, C1, C2) \leftarrow eq\_c(N, 0, C2)$
30. $newp([1|S], N, C1, C2) \leftarrow match\_c(S, M,\ plus\_c(s(0), M, N, C2))$
31. $newp([2|S], N, C1, C2) \leftarrow match\_c(S, M,\ plus\_c(s(0), M, N, C2))$
19. $plus\_c(0, X, X, C) \leftarrow C$
32. $plus\_c(s(X), Y, s(Z), C) \leftarrow plus\_c(X, Y, Z, C)$

This program *RegExprMatch*1 is in continuation passing style, avoids redundant calls in case of backtracking, and takes $O(n)$ resolution steps in the worst case, to find an occurrence of a substring of the form 0^*1, where n is the length of the input string. Moreover, by our Theorem 1 of Section 6, for every goal g of the form $match(t_1, t_2)$, where t_1 and t_2 are any terms, g terminates in *RegExprMatch* iff g terminates in *RegExprMatch*1 (see also Section 7.5).

7.4 Marking maximal elements

Let us consider the following marking problem. We are given: (i) a list $L1$ of the form $[x_0, \ldots, x_r]$, where for $i = 0, \ldots, r$, x_i is a list of integers, and (ii) an integer n ($\geqslant 0$). A list l of $s+1$ elements will also be denoted by $[l[0], \ldots, l[s]]$. We assume that for $i = 0, \ldots, r$, the list x_i has at least $n+1$ elements (and thus, the element $x_i[n]$ exists) and we denote by m the maximum element of the set $\{x_0[n], \ldots, x_r[n]\}$. From the list $L1$ we want to compute a new list $L2$ of the form $[y_0, \ldots, y_r]$ such that, for $i = 0, \ldots, r$, if $x_i[n] = m$ then $y_i[n] = \top$ else $y_i[n] = x_i[n]$.

For instance, if $L1 = [[3, 8, -2, 4],\ [1, 3],\ [1, 8, 1]]$ and $n = 1$, then $m = 8$, that is, the maximum element in $\{8, 3\}$. Thus, $L2 = [[3, \top, 2, 4],\ [1, 3],\ [1, \top, 1]]$.

The following program *MaxMark* computes the desired list $L2$ from the list $L1$ and the value N:

1. $mmark(N, L1, L2) \leftarrow max\_nth(N, L1, 0, M),\ mark(N, M, L1, L2)$
2. $max\_nth(N, [\,], M, M) \leftarrow$
3. $max\_nth(N, [X|Xs], A, M) \leftarrow nth(N, X, XN),\ max(A, XN, B),$
   $max\_nth(N, Xs, B, M)$
4. $nth(0, [H|T], H) \leftarrow$
5. $nth(s(N), [H|T], E) \leftarrow nth(N, T, E)$
6. $mark(N, M, [\,], [\,]) \leftarrow$
7. $mark(N, M, [X|Xs], [Y|Ys]) \leftarrow mark\_nth(N, M, X, Y),$
   $mark(N, M, Xs, Ys)$
8. $mark\_nth(0, M, [H1|T], [H2|T]) \leftarrow (M = H1, H2 = \top) \vee (M \neq H1, H2 = H1)$
9. $mark\_nth(s(N), M, [H|T1], [H|T2]) \leftarrow mark\_nth(N, M, T1, T2)$
10. $max(X, Y, X) \leftarrow X \geqslant Y$
11. $max(X, Y, Y) \leftarrow X < Y$

When running this program, the input list $L1 = [x_0, \ldots, x_r]$ is traversed twice: (i) the first time $L1$ is traversed to compute the maximum m of the set $\{x_0[n], \ldots, x_r[n]\}$ (see the goal $max\_nth(N, L1, 0, M)$ in the body of clause 1), and (ii) the second time

$L1$ is traversed to construct the list $L2$ by replacing, for $i=0,\dots,r$, the element $x_i[n]$ by $\top$ whenever $x_i[n]=m$ (see the goal $mark(N,M,L1,L2)$).

Now we use the transformation rules of Section 5 and from program *MaxMark* we derive a new program *MaxMark*1 which: (i) is in continuation passing style, (ii) preserves termination, and (iii) traverses the list $L1$ only once.

By the definition introduction rule we introduce the following new predicates with goal arguments:

12. $newp1(N,L1,L2,A,M,G,C1,C2) \leftarrow$
    $max\_nth(N,L1,A,M),\ (G=(mark(N,M,L1,L2),\ C1)),\ C2$
13. $newp2(N,X,M,Y,A,B,G1,G2,C) \leftarrow$
    $nth(N,X,XN),\ (G1=(mark\_nth(N,M,X,Y),G2)),$
    $max(A,XN,B),\ C$
14. $max\_c(X,Y,Z,C) \leftarrow max(X,Y,Z),\ C$

We unfold clauses 12, 13, and 14, and then we move leftwards term equalities (see law 3 in Section 5 which allows us to rearrange term equalities). We get the following clauses:

15. $newp1(N,[\,],[\,],M,M,C1,C1,C2) \leftarrow C2$
16. $newp1(N,[X|Xs],[Y|Ys],A,M,G,C1,C2) \leftarrow$
    $nth(N,X,XN),\ max(A,XN,B),\ max\_nth(N,Xs,B,M),$
    $(G=(mark\_nth(N,M,X,Y),\ mark(N,M,Xs,Ys),\ C1)),$
    $C2$
17. $newp2(0,[H1|T],M,[H2|T],A,B,G1,G2,C) \leftarrow$
    $(G1=(((M=H1,H2=\top) \vee (M\neq H1,H2=H1)),G2)),$
    $max(A,H1,B),\ C$
18. $newp2(s(N),[H|T1],M,[H|T2],A,B,G1,G2,C) \leftarrow$
    $nth(N,T1,XN),\ (G1=(mark\_nth(N,M,T1,T2),G2)),$
    $max(A,XN,B),\ C$
19. $max\_c(X,Y,X,C) \leftarrow X \geqslant Y,\ C$
20. $max\_c(X,Y,Y,C) \leftarrow X < Y,\ C$

By introducing and rearranging goal equalities, from clause 16 we get:

21. $newp1(N,[X|Xs],[Y|Ys],A,M,G,C1,C2) \leftarrow$
    $nth(N,X,XN),\ (G1=(mark\_nth(N,M,X,Y),\ G2)),$
    $max(A,XN,B),$
    $max\_nth(N,Xs,B,M),\ (G2=(mark(N,M,Xs,Ys),\ C1)),$
    $(G=G1),\ C2$

Finally, by folding steps and goal replacements based on the laws for the primitive predicates $=$, $\neq$, $\geqslant$, and $<$, we derive the following final program *MaxMark*1:

22. $mmark(N,L1,L2) \leftarrow newp1(N,L1,L2,0,M,G,true,G)$
15. $newp1(N,[\,],[\,],M,M,C1,C1,C2) \leftarrow C2$
23. $newp1(N,[X|Xs],[Y|Ys],A,M,G,C1,C2) \leftarrow$
    $newp2(N,X,M,Y,A,B,G1,G2),$
    $newp1(N,Xs,Ys,B,M,G2,C1,eq\_c(G,G1,C2)))$

24. $newp2(0, [H1|T], M, [H2|T], A, B, G1, G2, C) \leftarrow$
    $eq\_c(G1, (eq\_c(M, H1, eq\_c(H2, \top, G2)) \vee$
    $diff\_c(M, H1, eq\_c(H2, H1, G2))),$
    $max\_c(A, H1, B, C))$
25. $newp2(s(N), [H|T1], M, [H|T2], A, B, G1, G2, C) \leftarrow$
    $newp2(N, T1, M, T2, A, B, G1, G2, C)$
26. $max\_c(X, Y, X, C) \leftarrow geq\_c(X, Y, C)$
27. $max\_c(X, Y, Y, C) \leftarrow lt\_c(X, Y, C)$

This final program *MaxMark*1 is in continuation passing style and traverses the input list $L1$ only once. Moreover, by our Theorem 1 of Section 6, for every goal g of the form $mmark(t_1, t_2, t_3)$, where t_1, t_2, and t_3 are any terms, if g terminates in *MaxMark* then g terminates in *MaxMark*1 (see also Section 7.5).

7.5 Correctness of the program derivations

Let us briefly comment on the correctness properties of the program derivations we have presented in Section 7.

In all program derivations of Section 7, when using the transformation rules, we have complied with the restrictions indicated at Point (1) of Theorem 2 (Weak Correctness). Thus, for every program derivation from an initial program P_0 to a final program P_k, we have that P_k is a refinement of $P_0 \cup Def_k$, where Def_k is the set of definitions introduced during the derivation. In particular, for every ordinary goal g, if g terminates in P_0, then g terminates in P_k and the most general answer substitutions for g computed by P_0 are the same as those computed by P_k.

In the examples of Sections 7.1, 7.2, and 7.3 we have also complied with the restrictions of Point (2) of Theorem 2 (Strong Correctness), because all applications of the goal replacement rule are based on strong replacement laws. Thus, in these examples we have that P_k is equivalent to $P_0 \cup Def_k$. In particular, for every ordinary goal g, if g terminates in P_k then g terminates in $P_0 \cup Def_k$.

However, in the derivation of Section 7.4 we have not complied with the restrictions of Point (2) of Theorem 2. In particular, after unfolding clauses 12, 13, and 14, we have made leftward moves of term equalities by using law 3 of Section 5, and law 3 is not a strong replacement law. Thus, there may be an ordinary goal which does not terminate in the initial program *MaxMark* and terminates in the final program *MaxMark*1. Indeed, the goal $mmark(0, [H|T], [\,])$ does not terminate in *MaxMark* and terminates in *MaxMark*1.

Finally, in all program derivations of this Section 7, we have complied with the restrictions of Theorem 3 (Preservation of Safety), because all replacement laws we have applied preserve safety. Thus, since every ordinary goal is safe in the ordinary initial program P_0, we have that every ordinary goal is safe in the final program P_k.

7.6 Experimental results

In Table 1 below we have reported the speed-ups achieved in the examples presented in this paper. The *speed-up* (see Column D) is defined as the ratio between the

Table 1. *Speed-ups of the final programs with respect to the initial programs*

A. Initial program: Asymptotic Complexity	B. Final program: Asymptotic Complexity	C. Input goal	D. Speed-up:[a] $\frac{\text{run-time(A)}}{\text{run-time(B)}}$
1. *Deepest*: $O(n^2)$[b]	*Deepest*2: $O(n)$	$deepest(t_1, N)$	5.2
2. *DeepestOr*: $O(n^2)$[c]	*Deepest*2: $O(n)$	$deepest(t_2, N)$	2.7
3. *FlipCheck*: $O(n)$[d]	*FlipCheck*2: $O(n)$	$flipcheck(t_3, T)$	1.0
4. *TreeSum*: $O(n^2)$[e]	*TreeSum*2: $O(n)$	$treesum(t_4, N)$	9.2
5. *RegExprMatch*: $O(n^2)$[f]	*RegExprMatch*1: $O(n)$	$match(s, N)$	1.8
6. *MaxMark*: $O(n)$[g]	*MaxMark*1: $O(n)$	$mmark(n_1, l_1, L_2)$	1.8

[a] run-time(A) denotes the run-time of the program in Column A for the input goal in Column C. run-time(B) denotes the run-time of the program in Column B for the input goal in Column C.
[b] n is the number of nodes of the tree t_1.
[c] n is the number of nodes of the tree t_2.
[d] n is the number of nodes of the tree t_3. For the goal $flipcheck(t_3, T)$, the program *FlipCheck* visits the tree t_3 twice, while the program *FlipCheck*2 visits t_3 only once.
[e] n is the sum of the leaves of the tree t_4.
[f] n is the length of the string s.
[g] n is the sum of the lengths of the lists in l_1.

run-time of the initial program (see Column A) and the run-time of the derived, final program (see Column B). In Columns A and B we have also indicated the asymptotic worst-case time complexity of the initial and final programs, respectively. For each program the complexity is measured in terms of the size of the proofs relative to that program (or, equivalently, the number of LD-resolution steps performed using that program). The input goal is indicated in Column C. We performed our measurements by using BinProlog on a SUN workstation. This use is justified by the fact that every ordinary goal g is safe both in the initial program P_0 and in the final program P_k. Thus, we can use any Prolog system which implements LD-resolution (and, in particular, the BinProlog system) for computing the relations $P_0 \vdash g \longmapsto A$ and $P_k \vdash g \longmapsto A$ defined by our operational semantics.

In Column C of Table 1 we have that:

1. t_1 is a random binary tree with 100,000 nodes;
2. t_2 is a random binary tree with 100,000 nodes;
3. t_3 is a random binary tree with 20,000 nodes and each node is labeled by a numeral of the form $s^k(0)$, where $0 \leqslant k \leqslant 500$;
4. t_4 is a random binary tree with 20,000 nodes whose leaves are labeled by numerals of the form $s^k(0)$, where $0 \leqslant k \leqslant 500$;
5. s is a random sequence of integers of the form: $\{0, 2\}^{50000}1$; and
6. n_1 is 700, l_1 is a random list of 1000 lists, and each of these lists consists of 800 integers.

When measuring the speed-ups for the programs *Deepest* and *DeepestOr* in Rows 1 and 2 we have computed the set of all answer substitutions, while for the programs *FlipCheck*, *TreeSum*, *RegExprMatch*, and *MaxMark* in Rows 3–6 we have computed one answer substitution only.

As already mentioned at the end of Section 2, the value of the speed-up relative to the initial program *Deepest* (see Row 1) is higher than the value of the speed-up relative to the initial program *DeepestOr* (see Row 2), and this is not due to the use of goals as arguments, but to the introduction of a disjunction, thereby clauses 2 and 3 have been replaced by clause 16.

The absence of speed-up for the final program *FlipCheck*2 (see Row 3) with respect to the initial program *FlipCheck*, is caused by the fact that the efficiency improvements due to the elimination of the double traversal of the input tree t_4 are cancelled out by the slowdown due to the introduction of multiple continuation arguments. However, the experimental results for the initial program *MaxMark* and the final program *MaxMark*1 (see Row 6) show that the elimination of double traversals of data structures may yield a significant speed-up, especially when the access to the data structure is very costly. Recall that the program *MaxMark* traverses twice the list l_1, and for each list l in the list l_1, the program has to access n_1 elements of l. We have verified that the speed-up obtained by eliminating the double traversal of l_1 increases with the value of n_1.

8 Final Remarks and related work

We have shown that a simple extension of logic programming, where variables may range over goals and goals may appear as arguments of predicate symbols, can be very useful for transforming programs and improving their efficiency.

We have presented a set of transformation rules for our extended logic language and we have shown their correctness with respect to the operational semantics given in Section 4. In particular, in Section 6 we have shown that, under suitable conditions, our transformation rules preserve termination (see Theorem 1), most general answer substitutions (see Theorem 2), and safety (see Theorem 3). As in Bossi and Cocco (1994), for our logic programs we consider an operational semantics based on universal termination (that is, the operational semantics of a goal is defined iff all LD-derivations starting from that goal are finite). Theorem 2 extends the results presented in Bossi and Cocco (1994) for definite logic programs in that: (i) our language is an extension of definite logic programs, and (ii) our folding rule is more powerful. Indeed, even restricting ourselves to programs that do not contain goal variables and goal arguments, we allow folding steps which use clauses whose bodies contain disjunctions, and this is not possible in Bossi and Cocco (1994), where for applying the folding rule one is required to use exactly one clause whose body is a conjunction of atoms. However, one should notice that the transformations presented in Bossi and Cocco (1994) preserve *all* computed answer substitutions, while ours preserve the *most general* answer substitutions only.

Our logic language has some higher order capabilities because goals may occur as arguments, but these capabilities are limited by the fact that the quantification of function or predicate variables is not allowed. However, the objective of this paper is not the design of a new higher order logic language, such as those presented by other authors (Chen *et al.* 1993; Hill and Gallagher 1998; Nadathur and Miller 1998). Rather, our aim was to demonstrate the usefulness of some higher order constructs for deriving efficient logic programs by transformation. Indeed, we have shown that variables which range over goals are useful in the context of program transformation. Moreover, the use of these variables may avoid the need for goal rearrangements which could generate programs that do not preserve termination.

The approach we have proposed in this paper for avoiding incorrect goal rearrangements, is complementary to the approach described in Bossi *et al.* (1996), where the authors give sufficient conditions for goal rearrangements to preserve *left termination.* (Recall that a program P is said to be left terminating iff all *ground* goals universally terminate in P.) Thus, when these sufficient conditions are not met or their validity cannot be proved, one may apply our technique which avoids incorrect goal rearrangements by the introduction and the rearrangement of goal equalities. Indeed, we have proved that the application of our technique preserves universal termination, and thus, it preserves left termination as well.

The theory we have presented may also be used to give sound semantic foundations to the development of logic programs which use *higher order generalizations* and *continuations.* In Pettorossi and Proietti (1997), Tarau and Boyer (1990), Pettorossi and Skowron (1997) and Wand (1980), the reader may find some examples of use of these techniques in the case of logic and functional programs, respectively.

We leave for future work the development of suitable strategies for directing the use of the transformation rules we have proposed in this paper.

Acknowledgements

We would like to thank Michael Leuschel for pointing out an error in a preliminary version of this paper and for his helpful comments. We also thank the anonymous referees of the LoPSTr '99 Workshop, where a preliminary version of this paper was presented (Pettorossi and Proietti 2000), and the referees of the Theory and Practice of Logic Programming Journal for their suggestions.

This work has been partially supported by MURST Progetto Cofinanziato 'Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software' (Italy), and Progetto Coordinato CNR 'Verifica, Analisi e Trasformazione dei Programmi Logici' (Italy).

References

Apt, K. R. 1997. *From Logic Programming to Prolog*. Prentice Hall, London, UK.

Bossi, A. and Cocco, N. 1994. Preserving universal termination through unfold/fold. In *Proceedings ALP '94*. Lecture Notes in Computer Science 850. Springer-Verlag, Berlin, 269–286.

Bossi, A., Cocco, N. and Etalle, S. 1992. Transforming normal programs by replacement. In *Proceedings of Meta '92, Uppsala, Sweden*, A. Pettorossi, Ed. Lecture Notes in Computer Science 649. Springer-Verlag, Berlin, 265–279.

Bossi, A., Cocco, N. and Etalle, S. 1996. Transforming left-terminating programs: The reordering problem. In *Logic Program Synthesis and Transformation, Proceedings LoPSTr '95, Utrecht, The Netherlands*, M. Proietti, Ed. Lecture Notes in Computer Science 1048. Springer, Berlin, 33–45.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM 24,* 1 (January), 44–67.

Chen, W., Kifer, M. and Warren, D. S. 1993. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming 15,* 3, 187–230.

Hill, P. M. and Gallagher, J. 1998. Meta-programming in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. J. Hogger and J. A. Robinson, Eds. Vol. 5. Oxford University Press, Oxford, UK, 421–497.

Jørgensen, J., Leuschel, M. and Martens, B. 1997. Conjunctive partial deduction in practice. In *Proceedings of LoPSTr '96, Stockholm, Sweden*, J. Gallagher, Ed. Lecture Notes in Computer Science 1207. Springer-Verlag, Berlin, 59–82.

Lloyd, J. W. 1987. *Foundations of Logic Programming.* Springer-Verlag, Berlin. Second Edition.

Nadathur, G. and Miller, D. A. 1998. Higher-order logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Vol. 5. Oxford University Press, Oxford, UK, 499–590.

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

Pettorossi, A. and Proietti, M. 1997. Flexible continuations in logic programs via unfold/fold transformations and goal generalization. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Continuations, January 14, 1997, ENS, Paris (France) 1997*, O. Danvy, Ed. BRICS Notes Series, N6-93-13, Aahrus, Denmark, 9.1–9.22.

Pettorossi, A. and Proietti, M. 2000. Transformation rules for logic programs with goals as arguments. In *Proceedings of LoPSTr '99, Venezia, Italy*, A. Bossi, Ed. Lecture Notes in Computer Science 1817. Springer-Verlag, Berlin, 177–196.

Pettorossi, A. and Proietti, M. 2003. Transformations for logic programs with goals as arguments. Technical report, Computing Research Repository (CoRR), `http://xxx.lanl.gov/archive/cs/intro.html`.

Pettorossi, A. and Skowron, A. 1987. Higher order generalization in program derivation. In *Proceedings of TAPSOFT '87*. Lecture Notes in Computer Science 250. Springer-Verlag, Berlin, 182–196.

Sands, D. 1996. Total correctness by local improvement in the transformation of functional programs. *ACM Toplas 18,* 2, 175–234.

Sterling, L. S. and Shapiro, E. 1986. *The Art of Prolog.* The MIT Press, Cambridge, MA.

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

Tarau, P. 1996. BinProlog 5.25. User Guide. Technical report, University of Moncton, Canada.

Tarau, P. and Boyer, M. 1990. Elementary logic programs. In *Proceedings of PLILP '90*, P. Deransart and J. Małuszyński, Eds. Lecture Notes in Computer Science 456. Springer-Verlag, Berlin, 159–173.

Vasak, T. and Potter, J. 1986. Characterization of terminating logic programs. In *Proceedings of the Third IEEE International Symposium on Logic Programming, Salt Lake City, Utah.* IEEE Press, Washington, DC, 140–147.

Wand, M. 1980. Continuation-based program transformation strategies. *Journal of the ACM 27,* 1, 164–180.

Warren, D. H. D. 1982. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence*, Y.-H. P. J. E. Hayes, D. Michie, Ed. Vol. 10. Ellis Horwood, Chichester, 441–454.

Winskel, G. 1993. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, Cambridge, MA.