On Rewrite Programs: Semantics and Relationship with Prolog *

Maria Paola Bonacina   **Jieh Hsiang**
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400 USA
{bonacina,hsiang}@sbcs.sunysb.edu

Abstract

Rewrite programs are logic programs represented as rewrite rules, whose execution mechanism usually employs some version of Knuth-Bendix type completion. Rewrite programs allow one to express mutually exclusively defined predicates as well as those which are not. In this paper we demonstrate that rewrite programs, although denotationally equivalent to Prolog on the ground level, may produce fewer answers in general. Consequently, a rewrite program may halt with finitely many answers while the corresponding Prolog program goes into an infinite loop. In order to explain these observations, we present a precise operational semantics for rewrite programs, define their denotational (fixpoint) semantics, prove the equivalence of operational, model theoretic and denotational semantics, and clarify the relationship between rewrite programs and Prolog. Comparisons between the pruning effects of simplification and those of subsumption based loop checking mechanisms for Prolog are also included.

Running title: On Rewrite Programs

1   Introduction

Term rewriting systems have been widely applied in functional programming [4, 13, 14, 16] and to a lesser extent in logic programming [10, 9, 11, 20]. In case of functional programs the evaluation mechanism is reduction and a computation consists in reducing a ground input term to an irreducible form, which represents the output. To achieve logic programming the evaluation mechanism is extended to reduction and linear superposition. This amounts to a strongly restricted form of Knuth-Bendix completion, termed *linear completion* [10]. A computation consists in generating an answer substitution for a non ground query as it is done in Prolog.

Despite various approaches suggested, there is a common misconception that rewrite programs have the same semantics as Prolog except for a different evaluation mechanism. Surprisingly enough, this is not true in general: in Section 2 we give examples to show that rewrite programs can avoid certain infinite loops which occur in similar Prolog programs. In the next section, we define the language of rewrite programs. Program units in rewrite programs are *logical equivalences*, not implications as in Prolog. In this work we concentrate on rewrite programs where rewrite rules are equivalences between conjunctions of first-order atoms. The language does not include an equality predicate and is therefore relational, like in pure Prolog. Since implications can be written as equivalences, the programmer may choose between equivalences and implications. The examples of Section 2 show that rewrite programs may turn out to be more effective than Prolog in capturing the user's intended semantics, because of this feature.

Since program units are equivalences, an interpreter for rewrite programs includes naturally an inference rule for *simplification*. Our interpreter (Section 3) is a new *linear completion* procedure that differs from the previous ones [10, 9, 11], because we allow simplification of goals by their ancestors. In Section 4, we prove that the operational semantics represented by our linear completion interpreter is equivalent to the model theoretic semantics.

Simplification prunes some infinite derivations and may optimize the search for solutions. This explains at the operational level the difference with Prolog. For the denotational level, first we define a denotational semantics for rewrite programs (Section 5) and we show its equivalence with the operational semantics (Section 6). Next, in Section 7, we prove that rewrite programs and Prolog programs have the same fixpoint, i.e., the same set of ground true facts. A rewrite program may give fewer answers, because if predicates are defined by equivalences, distinct Prolog answers turn out to be equivalent with respect to the rewrite program. However, for every Prolog answer there is an equivalent answer given by the rewrite program. This is why a smaller set of answers covers the same set of ground true facts. In Section 8 we compare our approach to others, especially those of [2, 3].

2 Rewrite programs

The main reason for the different behaviour of rewrite programs is the utilization of an inference rule for simplification. We demonstrate its use with a simple example. The Prolog program [1]

$append([\ ], L, L).$
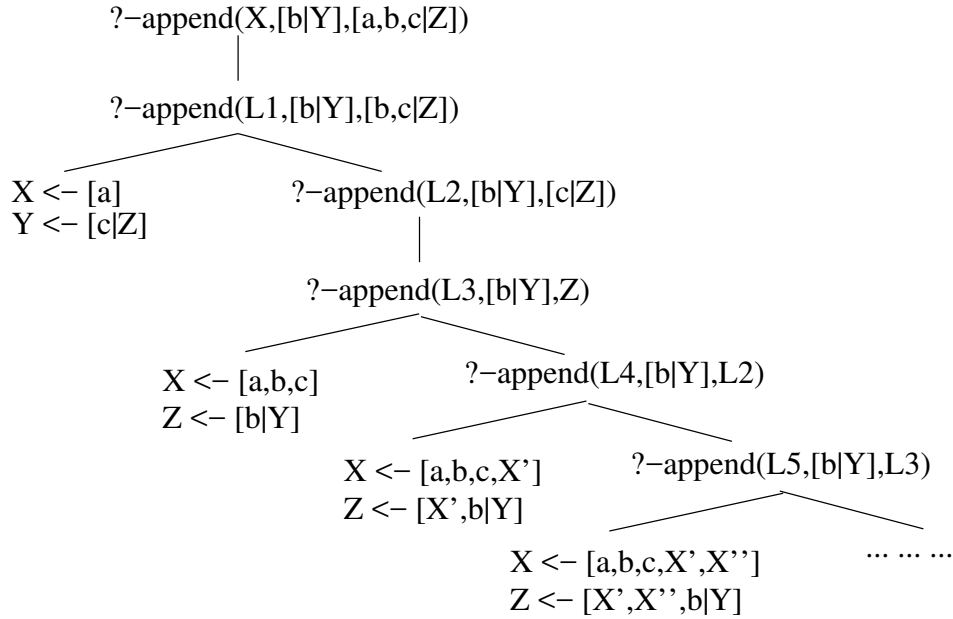
$append([X|L1], Y, [X|L2]) :- append(L1, Y, L2).$

with the query

$?- append(X, [b|Y], [a, b, c|Z]).$

generates an infinite set of solutions as shown in Fig. 1.

---

[1] We recall that $[\ ]$ is *nil* and $[X|Y]$ is $cons(X, Y)$.

$$?-append(X,[b|Y],[a,b,c|Z])$$
$$|$$
$$?-append(L1,[b|Y],[b,c|Z])$$

X <− [a]
Y <− [c|Z]

$$?-append(L2,[b|Y],[c|Z])$$
$$|$$
$$?-append(L3,[b|Y],Z)$$

X <− [a,b,c]
Z <− [b|Y]

$$?-append(L4,[b|Y],L2)$$

X <− [a,b,c,X']
Z <− [X',b|Y]

$$?-append(L5,[b|Y],L3)$$

X <− [a,b,c,X',X'']
Z <− [X',X'',b|Y]

... ...

$\{X \leftarrow [a],\ Y \leftarrow [c|Z]\}$

$\{X \leftarrow [a,b,c],\ Z \leftarrow [b|Y]\}$

$\{X \leftarrow [a,b,c,X'],\ Z \leftarrow [X',b|Y]\}$

$\{X \leftarrow [a,b,c,X',X''],\ Z \leftarrow [X',X'',b|Y]\}$

. . . .

$\{X \leftarrow [a,b,c,X',X'',\ldots,X^n],\ Z \leftarrow [X',X'',\ldots,X^n,b|Y]\}$

. . . .

Figure 1: Prolog generates an infinite set of solutions

The rewrite program, on the other hand, is defined as

$$append([\,], L, L) \rightarrow true$$

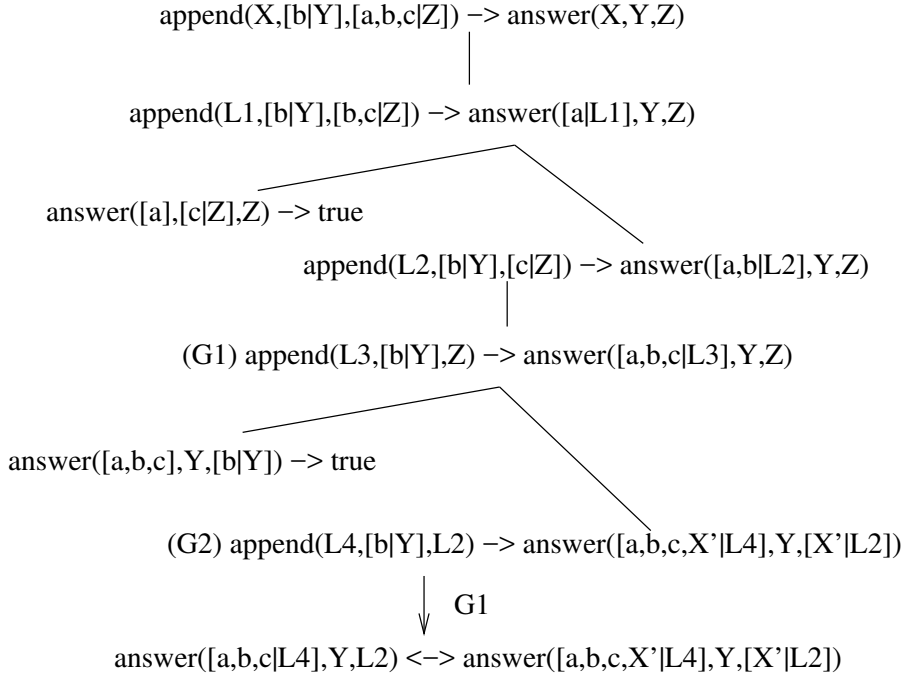$$append([X|L1], Y, [X|L2]) \rightarrow append(L1, Y, L2)$$

and the query is

$$append(X, [b|Y], [a, b, c|Z]) \rightarrow answer(X, Y, Z).$$

If we execute this program by linear completion we get only the first two answers as shown in Fig. 2.

At each step a program rule is superposed onto the current goal, by unifying an atom in the left-hand side of the rule with an atom in the left-hand side of the goal. The last step, labeled by $\downarrow_{G1}$, is a simplification step where the ancestor goal G1 rewrites the current goal G2 to a trivial goal in the form $answer(\ ) \leftrightarrow answer(\ )$[2]. Since all generated rules are consequences of the program and the query, it is intuitively sound to use ancestors to reduce subgoals. No inference can be applied to the last goal and the execution halts with just two answers.

---

[2]We use the double arrow whenever the orientation of the rule is irrelevant.

append(X,[b|Y],[a,b,c|Z]) –> answer(X,Y,Z)

append(L1,[b|Y],[b,c|Z]) –> answer([a|L1],Y,Z)

answer([a],[c|Z],Z) –> true

append(L2,[b|Y],[c|Z]) –> answer([a,b|L2],Y,Z)

(G1) append(L3,[b|Y],Z) –> answer([a,b,c|L3],Y,Z)

answer([a,b,c],Y,[b|Y]) –> true

(G2) append(L4,[b|Y],L2) –> answer([a,b,c,X'|L4],Y,[X'|L2])

G1

answer([a,b,c|L4],Y,L2) <–> answer([a,b,c,X'|L4],Y,[X'|L2])

$\{X \leftarrow [a],\ Y \leftarrow [c|Z]\}$
$\{X \leftarrow [a,b,c],\ Z \leftarrow [b|Y]\}$

Figure 2: Linear completion generates only two answers

The reason for this different behaviour is that the "program units" in a rewrite program and in a Prolog program are interpreted in two different ways. In Prolog, each clause is an implication, where ":−" indicates the logical *if*. In rewrite programs, each unit is a logical equivalence, where "→" (and "↔") means *if and only if*. The infinitely many answers in the form

$$\{X \leftarrow [a,b,c,X',X'',\ldots,X^n],\ Z \leftarrow [X',X'',\ldots,X^n,b|Y]\}$$

given by Prolog are not equivalent if *append* is defined by implications, but they all collapse to the second solution

$$\{X \leftarrow [a,b,c],\ Z \leftarrow [b|Y]\}$$

if *append* is defined by bi-implications. The first answer returned by the rewrite program corresponds to the first answer of the Prolog program. The second answer of the rewrite program corresponds to the second answer of the Prolog program and all the proceeding ones.

We can see why this happens by instantiating the query by the answer substitutions. If the query is instantiated by the first answer substitution, it is rewritten to $append([\,],[b,c|Z],[b,c|Z])$ and then to *true*. If it is instantiated by the second answer or any of the proceeding ones, it is rewritten to $append([\,],[b|Y],[b|Y])$ and then to *true*. All the answers but the first one yield the same true fact if *append* is defined by equivalences and simplification is applied. All those answers are equivalent to the second one with respect to the rewrite program, so that they are

4

not generated.

Since the intended definition of *append* is actually

$$append([X|L1], Y, [X|L2]) \; if \; and \; only \; if \; append(L1, Y, L2),$$

the rewrite program seems closer to the intended meaning of the program than the Prolog program.

The interpretation of program units as logical equivalences may also help resolving some loops which may occur in Prolog. For example, consider the following Prolog program:

$$\dots$$
$$P(X, Y, Z) :- append(X, [b|Y], [a, b, c|Z]), non-member(a, X).$$
$$P(X, Y, Z) :- \dots.$$
$$\dots$$
$$append([\,], L, L).$$
$$append([X|L1], Y, [X|L2]) :- append(L1, Y, L2).$$
$$\dots$$

with the query $?- P(X, Y, Z)$.

Prolog falls into an infinite loop when evaluating the first clause for P, since a is a member of X in all the solutions of $append(X, [b|Y], [a, b, c|Z])$. Such potential loops cannot even be prevented beforehand by using *cut*. The rewrite program does not loop and evaluate the second clause of P, since there are only two answers from evaluating the *append* subgoal and none of them satisfies the $non-member$ subgoal.

One may suspect that simplification may throw away too many answers and change the intended semantics. For instance, consider the following:

$$P(X, Y, Z) :- append(X, [b|Y], [a, b, c|Z]), size(X) > 3.$$
with the query $?- P(X, Y, Z)$.

Since in the two answers for the *append* subgoal generated in the previous examples the size of X is less than or equal to three, it seems that no solution would be provided. This is not the case. When $P(X, Y, Z) \to answer(X, Y, Z)$ is given as the query, the execution first generates the two solutions to the *append* subgoal

$$size([a]) > 3 \to answer([a], [c|Z], Z),$$
$$size([a, b, c]) > 3 \to answer([a, b, c], Y, [b|Y]),$$

both of which fail to give any solution to the query. Then the execution continues with the goal

$$append(L4, [b|Y], L2), size([a, b, c, X'|L4]) > 3 \to answer([a, b, c, X'|L4], Y, [X'|L2]).$$

Assuming that *size* is defined in such a way that $size([a, b, c, X'|L4]) > 3$ simplifies to *true*, this

goal is reduced to

$$append(L4, [b|Y], L2) \rightarrow answer([a, b, c, X'|L4], Y, [X'|L2]) \text{ (G2)}.$$

Note that this is the same goal G2 generated in the previous execution for the *append* query. In that execution G2 is rewritten by its ancestor G1 and it does not yield any answer. In the present execution all ancestors also contain a *size* literal, so that they do not apply to simplify the goal G2, which yields a correct solution

$$answer([a, b, c, X'], Y, [X', b|Y]) \rightarrow true.$$

Then, the computation halts as the new goal

$$append(L5, [b|Y], L3) \rightarrow answer([a, b, c, X', X''|L5], Y, [X', X''|L3])$$

is reduced by its ancestor G2 to

$$answer([a, b, c, X'|L5], Y, [X'|L3]) \leftrightarrow answer([a, b, c, X', X''|L5], Y, [X', X''|L3]) \ {}^{3}.$$

The simplification mechanism keeps automatically into account that the predicate P is defined by the conjunction of the *append* literal and of the *size* literal. The presence of the *size* literal in the definition of P forces the interpreter to generate the answer $\{X \leftarrow [a, b, c, X'], Z \leftarrow [X', b|Y]\}$. The substitution $\{X \leftarrow [a, b, c, X'], Z \leftarrow [X', b|Y]\}$ is no longer equivalent to the substitution $\{X \leftarrow [a, b, c], Z \leftarrow [b|Y]\}$, because the first one is an answer to the query $? - P(X, Y, Z)$ if $size([a, b, c, X']) > 3$ is equivalent to *true*, whereas the second one is an answer to the query $? - P(X, Y, Z)$ only if $size([a, b, c]) > 3$ is evaluated to *true*, which does not hold for any reasonable definition of *size*.

So far, we have seen examples where it is desirable to define predicates by equivalences. However, not all relations are meant to be defined by equivalences. Rewrite programs also allow us to define predicates by implications. For example, for the *ancestor* relation

$$ancestor(X, Y) :- parent(X, Y).$$
$$ancestor(X, Y) :- parent(Z, Y), \ ancestor(X, Z).$$

both clauses are meant to be implications. As it was already pointed out in [9], implications can be written as bi-implications by recalling that $A :- B_1 \ldots B_n$ is equivalent to $A B_1 \ldots B_n \rightarrow B_1 \ldots B_n$. If we add some facts and a query, we get:
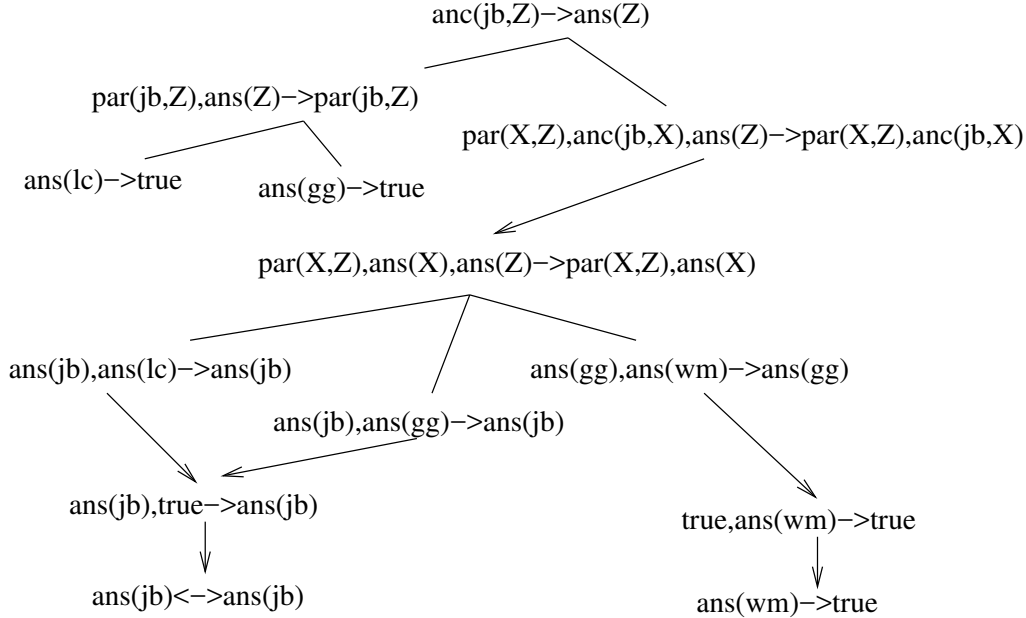
$$parent(jb, lc) \rightarrow true$$
$$parent(jb, gg) \rightarrow true$$
$$parent(gg, wm) \rightarrow true$$
$$ancestor(X, Y), \ parent(X, Y) \rightarrow parent(X, Y)$$

---

[3] We shall see in the following that the inference mechanism of linear completion includes overlap between the current goal and previously generated answers. In this case, the current goal is oriented from right to left, and the previously generated answer does not overlap on the greater side. Therefore, no further step is possible.

$$ancestor(X,Y),\ parent(Z,Y),\ ancestor(X,Z) \rightarrow parent(Z,Y),\ ancestor(X,Z)$$

$$ancestor(jb,Z) \rightarrow answer(Z).$$

This program gives the same answers as Prolog, but the computation is optimized by simplification of goals as shown in Fig. 3.

anc(jb,Z)−>ans(Z)

par(jb,Z),ans(Z)−>par(jb,Z)

par(X,Z),anc(jb,X),ans(Z)−>par(X,Z),anc(jb,X)

ans(lc)−>true   ans(gg)−>true

par(X,Z),ans(X),ans(Z)−>par(X,Z),ans(X)

ans(jb),ans(lc)−>ans(jb)   ans(gg),ans(wm)−>ans(gg)

ans(jb),ans(gg)−>ans(jb)

ans(jb),true−>ans(jb)

true,ans(wm)−>true

ans(jb)<−>ans(jb)

ans(wm)−>true

$\{Z \leftarrow lc\}$
$\{Z \leftarrow gg\}$
$\{Z \leftarrow wm\}$

Figure 3: Optimization of the computation by simplification

The generation of the first two answers is the same as in Prolog. The third answer is different. Having the goal $parent(X,Z) \wedge ancestor(jb,X)$, Prolog first generates $ancestor(jb,jb)$ twice, fails twice, then generates the goal $ancestor(jb,gg)$, which yields the answer $z \leftarrow wm$, and a third failing computation of the goal $ancestor(jb,jb)$. These failing paths are pruned by rewriting. Simplification by previously generated answers reduces the number of recursive applications of the definition of *ancestor* and the amount of backtracking performed by the interpreter. This is not surprising, since simplification is very well known as a powerful way to reduce search space.

3 Syntax and operational semantics of rewrite programs

3.1 The syntax of rewrite programs

As we have seen in the previous section, rewrite programs allow us to define predicates by either equivalences or implications. The choice is left to the programmer. However, it is also possible to give a criterion to decide automatically whether a predicate should be defined by equivalences

or implications. We assume that we have a set of Prolog clauses defining a predicate. We say that a predicate is *mutually exclusively defined* if it is defined by a set of clauses such that no two heads unify[4]. Then, we can establish the following criterion: if a predicate A is defined by a set of clauses

$$A(\bar{t}_1).$$

$$\ldots$$

$$A(\bar{t}_m).$$

$$A(\bar{t}_{m+1}) :- B_{11} \ldots B_{1p_1}.$$

$$\ldots$$

$$A(\bar{t}_{m+n}) :- B_{n1} \ldots B_{np_n}.$$

its rewrite program contains the rules

$$A(\bar{t}_1) \rightarrow true.$$

$$\ldots$$

$$A(\bar{t}_m) \rightarrow true.$$

$$A(\bar{t}_{m+1}) \rightarrow B_{11} \ldots B_{1p_1}.$$

$$\ldots$$

$$A(\bar{t}_{m+n}) \rightarrow B_{n1} \ldots B_{np_n}.$$

if A is mutually exclusively defined and $\forall i, 1 \leq i \leq n, A(\bar{t}_{m+i}) \succ B_{i1} \ldots B_{ip_i}$, where $\succ$ is a simplification ordering [5]. Otherwise, A is transformed into

$$A(\bar{t}_1) \rightarrow true.$$

$$\ldots$$

$$A(\bar{t}_m) \rightarrow true.$$

$$A(\bar{t}_{m+1})B_{11} \ldots B_{1p_1} \rightarrow B_{11} \ldots B_{1p_1}.$$

$$\ldots$$

$$A(\bar{t}_{m+n})B_{n1} \ldots B_{np_n} \rightarrow B_{n1} \ldots B_{np_n}.$$

We assume this criterion for translation throughout this paper. All the rewrite rules are equivalences of conjunctions of atoms. Since the bi-implication $AB_1 \ldots B_n \rightarrow B_1 \ldots B_n$ is equivalent to the implication $A :- B_1 \ldots B_n$, in the second case A is defined by implications. We call the rewrite rules representing facts, implications and bi-implications *fact rules*, *if-rules* and *iff-rules*.

---

[4]According to this definition, the clauses $p(x) :- x > 0 \ldots$ and $p(x) :- x \leq 0 \ldots$ are not mutually exclusive and clearly this is not entirely satisfactory. However, this definition is sufficient for the purposes of this paper. A more general notion of mutually exclusive clauses can be found in [6].

[5]A simplification ordering is an ordering with the following properties: $s \succ t$ implies $s\sigma \succ t\sigma$ for all substitutions σ (stability), $s \succ t$ implies $c[s] \succ c[t]$ for all contexts c (monotonicity) and $c[s] \succ s$ (subterm property). These properties imply that $\succ$ is well founded [7].

Our criterion consists then in defining A by iff-rules if its clauses are mutually exclusive, by if-rules otherwise. However, if the condition $\forall i, 1 \leq i \leq n, A(\bar{t}_{m+i}) \succ B_{i1} \ldots B_{ip_i}$ does not hold, the clauses for A are converted into if-rules even if it is mutually exclusively defined. This guarantees that all rewrite rules can be oriented. A *rewrite program* is a rewrite system of if-rules, iff-rules, and fact rules. If a program has only if-rules and fact rules, then we also call it an *if-program*. Note that every Prolog program can be transformed into an if-program. Otherwise, it is called an *iff-program*.

3.2   The operational semantics of rewrite programs

Rewrite programs are interpreted by *linear completion*. A query $\exists \bar{x} Q_1 \ldots Q_m$ (where $Q_1 \ldots Q_m$ is a conjunction of atoms) is negated into $Q_1 \ldots Q_m \rightarrow false$ and written as a *query rule* $Q_1 \ldots Q_m \rightarrow answer(\bar{x})$, where $\bar{x}$ contains all the free variables in $Q_1 \ldots Q_m$. When a rule in the form $answer(\bar{x})\sigma \rightarrow true$ is deduced, $\bar{x}\sigma$ is a solution to the query. The *answer* literal was used by Dershowitz in [9], who refers to [15] for its introduction. The *state* of a computation is defined by a triple

$$(E; L_1 \ldots L_l \rightarrow R_1 \ldots R_r; S),$$

where E is the program, $L_1 \ldots L_l \rightarrow R_1 \ldots R_r$ is the current goal, and S contains all the ancestors of the current goal. A *goal rule* is any rule generated by the program starting from the query. A computation can be described by a tree, where each node is labeled by a triple. The root represents the initial state

$$(E; Q_1 \ldots Q_m \rightarrow answer(\bar{x}); \emptyset).$$

A *final state* is a state such that no inference step can be performed on the current goal. The computation stops when all the leaves in the tree are labeled by final states. A final state in the form

$$(E; answer(\bar{x})\sigma \rightarrow true; S)$$

means that the answer substitution σ for the query is found. An answer rule represents a contradiction in the refutational sense, since *answer* stands for $false$. The interpreter builds a refutation starting with the query and ending with a contradiction:

$$E \cup \{Q_1 \ldots Q_m \rightarrow answer(\bar{x})\} \vdash_{LC} answer(\bar{x})\sigma \rightarrow true.$$

We also denote it by $E \vdash_{LC} Q_1 \ldots Q_m \sigma$, meaning that $Q_1 \ldots Q_m \sigma$ is proved from program E by linear completion. If a ground query is given, the answer substitution is empty and we write $E \vdash_{LC} Q_1 \ldots Q_m$. A computation step transforms the current state by applying one of the following **inference rules**, explained below:

Answer:

$$\frac{(E; answer(\bar{x})\sigma \rightarrow true; S)}{(E \cup \{answer(\bar{x})\sigma \rightarrow true\}; \text{---}; S)}$$

Delete:

$$\frac{(E; L_1 \ldots L_l \leftrightarrow L_1 \ldots L_l; S)}{(E; \text{---}; S)}$$

Orient:

$$\frac{(E; L_1 \ldots L_l \leftrightarrow R_1 \ldots R_r; S)}{(E; L_1 \ldots L_l \to R_1 \ldots R_r; S)} \quad if \quad L_1 \ldots L_l \succ R_1 \ldots R_r$$

Simplify:

$$\frac{(E; L_1 \ldots L_l \to R_1 \ldots R_r; S)}{(E; L'_1 \ldots L'_n \leftrightarrow R'_1 \ldots R'_s; S)}$$

Overlap:

$$\frac{(E; L_1 \ldots L_l \to R_1 \ldots R_r; S)}{(E; L'_1 \ldots L'_n \leftrightarrow R'_1 \ldots R'_s; S \cup \{L_1 \ldots L_l \to R_1 \ldots R_r\})}$$

No overlap between two program rules, no overlap between two goal rules and no simplification of program rules are used. The name linear completion emphasizes that the process is *linear* with respect to superposition.

In **Answer**, an answer is found and the answer rule is added to the program. In this way computed answers are saved and can be used while searching for other solutions.

In **Delete**, a goal which is an identity is deleted. The bar "---" simply indicates that after and Answer or Delete step the goal is empty.

In **Orient**, the goal is oriented according to a simplification ordering $\succ$ such that goal rules in the form $B_1 \ldots B_n answer(\bar{t}) \to B_1 \ldots B_n$ and $B_1 \ldots B_n \to answer(\bar{t})$ are oriented from left to right.

In **Simplify**, $L_1 \ldots L_l \to R_1 \ldots R_r$ is simplified into a new goal $L'_1 \ldots L'_n \leftrightarrow R'_1 \ldots R'_s$ using $E \cup S$ and $L_1 \ldots L_l \to R_1 \ldots R_r$ is discarded. Since the product is associative, commutative and idempotent, we regard conjunctions of atoms as *sets* of atoms. A rule $\bar{C} \to \bar{D}$ in $E \cup S$ matches a side $L_1 \ldots L_l$ of the current goal rule if there is a subset of $L_1 \ldots L_l$ which is an instance $\bar{C}\sigma$ of $\bar{C}$. The new goal $L'_1 \ldots L'_n \leftrightarrow R'_1 \ldots R'_s$ is $L_1 \ldots L_l \to R_1 \ldots R_r$ where $\bar{C}\sigma$ is replaced by $\bar{D}\sigma$. No AC-matching is needed, since all the matching operations occur below the product, between pairs of literals. The current goal is simplified on both sides and we assume that simplification of the left side and simplification of the right side are performed in the same simplification step. The rewrite rules $x \cdot true \to x$ and $x \cdot x \to x$ are also implicitly applied. They delete any repeated atom and any occurrence of $true$ in a conjunction. Therefore, if the simplifier $\bar{C} \to \bar{D}$ is a fact $A \to true$ or an answer $answer(\bar{t}) \to true$ or an if-rule $AB_1 \ldots B_n \to B_1 \ldots B_n$, then the atom L_j such that $A\sigma = L_j$ or $answer(\bar{t})\sigma = L_j$ is deleted.

Simplification of the current goal by program rules, ancestor goal rules and answer rules is the key feature of our linear completion interpreter. In the previous definitions of linear completion [10, 9, 11] only simplification by program rules was allowed. Since goal rules are logical consequences of the program and the query, it is sound to use them to simplify. If simplification of goals by goals is forbidden, the effects of interpreting program units as equivalences are artificially

limited, leading to the misconception that rewrite programs and Prolog programs behave in the same way.

In **Overlap**, a new goal is generated by overlapping the current goal with a rule in E. The new goal replaces the current one, which is moved to the ancestors set. The overlap steps in LC are similar to the resolution steps in Prolog. Given a goal rule $A(\bar{u})L_1 \dots L_l \to R_1 \dots R_r$ and a program rule, one of the three overlapping inferences can be used according to the type of the program rule:

Overlap with an if-rule:

$$\frac{A(\bar{s})B_1 \dots B_n \to B_1 \dots B_n, \ A(\bar{u})L_1 \dots L_l \to R_1 \dots R_r}{(B_1 \dots B_n L_1 \dots L_l)\sigma \to (B_1 \dots B_n R_1 \dots R_r)\sigma}$$

Overlap with an iff-rule:

$$\frac{A(\bar{s}) \to B_1 \dots B_n, \ A(\bar{u})L_1 \dots L_l \to R_1 \dots R_r}{(B_1 \dots B_n L_1 \dots L_l)\sigma \to (R_1 \dots R_r)\sigma}$$

Overlap with a fact rule:

$$\frac{A(\bar{s}) \to true, \ A(\bar{u})L_1 \dots L_l \to R_1 \dots R_r}{(L_1 \dots L_l)\sigma \leftrightarrow (R_1 \dots R_r)\sigma}$$

where σ is the most general unifier of $A(\bar{s})$ and $A(\bar{u})$. Here and in the following, we assume that the leftmost literal in a goal is selected[6]. An overlap step replaces a literal in the goal by a proper instance of its predicate's definition. If the defining formula is an implication, the new set of subgoals is added to both sides of the goal equation. If it is a bi-implication, it is added to the left side only. An overlap with a fact deletes a literal in the goal list.

No overlap on an atom different from the head of a rule needs to be considered. To see this, if the atom $A(\bar{u})$ in the goal rule $A(\bar{u})\bar{L} \to \bar{R}$ unifies with mgu σ with an atom $A(\bar{v})$ in an if-rule $CA(\bar{v})\bar{B} \to A(\bar{v})\bar{B}$, the overlap step generates the goal

$$(C\bar{B}\bar{R})\sigma \to (A(\bar{v})\bar{B}\bar{L})\sigma$$

which is reduced by its predecessor $A(\bar{u})\bar{L} \to \bar{R}$ to

$$(C\bar{B}\bar{R})\sigma \to (\bar{B}\bar{R})\sigma.$$

A following overlap on literal C between this new goal and the same program rule $CA(\bar{v})\bar{B} \to A(\bar{v})\bar{B}$ will lead to the identity $(A(\bar{v})\bar{B}\bar{R})\sigma \leftrightarrow (A(\bar{v})\bar{B}\bar{R})\sigma$.

A linear completion interpreter builds the tree sequentially according to some **search plan**. The search plan selects the inference rule and the program rule for the next step. We require that the search plan try the inference rules in the following order: *Delete, Answer, Orient, Simplify* and *Overlap*. Therefore, the current goal is always fully simplified before the next overlap step is performed. This choice ensures that the interpreter performs an overlap step and expands the

---

[6]It is well known that computations differing in the order of selection of literals give the same answers up to renaming of variables [5, 19].

11

search space only if the current search space has been pruned first as much as possible.

A search plan for linear completion includes backtracking: if no inference rule applies to the current goal or the goal is empty, such as after an *Answer* or *Delete* step, the interpreter backtracks. Backtracking consists in undoing steps. Our interpreter undoes only modifications on the second and third components of the state, that is, the goal and the ancestors set. Modifications on the program component, i.e., additions of answer rules, are not undone. In this way, the interpreter does not forget the already computed answers and applies them as rules while visiting other paths in the tree. The *ancestor* example in Section 2 shows that keeping answer rules around is necessary to generate all the answer substitutions. In addition, simplification by answers rules contributes to pruning the search tree and, therefore, to optimizing the computation. In the following we assume that a fair search plan is adopted whenever needed.

4 Equivalence of operational and model theoretic semantics

For a rewrite program E, we denote by $\mathcal{B}$ its Herbrand base and by $\mathcal{P}(\mathcal{B})$ the set of all subsets of $\mathcal{B}$, i.e. the set of all the Herbrand interpretations. The operational semantics of E is its *success set*, $\{G \mid G \in \mathcal{B},\ E \vdash_{LC} G\}$. The model theoretic semantics is the set $\{G \mid G \in \mathcal{B},\ E^* \models G = true\}$, where $E^* = E \cup \{x \cdot x \to x,\ x \cdot true \to x\}$. In order to show the equivalence of operational and model theoretic semantics, we need to show that these two sets of ground atoms are equal, i.e. for all $G \in \mathcal{B}$, $E \vdash_{LC} G$ implies $E^* \models G = true$ and vice versa. For the first direction, a stronger result holds: we say that a substitution σ is a *correct answer substitution* for the query $Q_1 \ldots Q_m$ if $E^* \models Q_1 \ldots Q_m \sigma = true$. We can prove that all the answer substitutions given by linear completion are correct, i.e. $E \vdash_{LC} Q_1 \ldots Q_m \sigma$ implies $E^* \models Q_1 \ldots Q_m \sigma = true$. This amounts to proving the *soundness* of linear completion:

Theorem 4.1 *If $E \vdash_{LC} Q_1 \ldots Q_m \sigma$ then $E^* \models Q_1 \ldots Q_m \sigma = true$.*

Proof: soundness of linear completion follows from soundness of replacing equals by equals, since the inference rules of linear completion are applications of equational replacement.

$E \vdash_{LC} Q_1 \ldots Q_m \sigma$ stands for $E \cup \{Q_1 \ldots Q_m \to answer(\bar{x})\} \vdash_{LC} answer(\bar{x})\sigma \to true$. The equation $answer(\bar{x})\sigma \to true$ is derived by LC from E and $Q_1 \ldots Q_m \to answer(\bar{x})$. Since the *answer* literal is just a place holder for $Q_1 \ldots Q_m$, this actually means that the equation $Q_1 \ldots Q_m \sigma \to true$ is derived from E by equational replacement, or $Q_1 \ldots Q_m \sigma \leftrightarrow^*_{E^*} true$[7]. Then, $Q_1 \ldots Q_m \sigma \leftrightarrow^*_{E^*} true$ implies $E^* \models Q_1 \ldots Q_m \sigma = true$ by the soundness of replacing equals by equals. $\square$

We consider now *completeness*, that is whether $E^* \models Q_1 \ldots Q_m \sigma = true$ implies $E \vdash_{LC} Q_1 \ldots Q_m \sigma$. In this section we prove completeness for ground queries, which is sufficient to establish, together with soundness, the equivalence of operational and model theoretic semantics. We shall discuss in Section 7 completeness for non-ground queries. We show that all ground queries proved by linear completion from E are equivalent to *true*:

---

[7]The relation $\leftrightarrow^*_{E^*}$ is the transitive, symmetric and reflexive closure of equational replacement by the rules in E

Theorem 4.2 $\forall Q_1 \ldots Q_m \in \mathcal{B}$, $E \vdash_{LC} Q_1 \ldots Q_m$ *if and only if* $Q_1 \ldots Q_m \leftrightarrow^*_{E^*} true$.

Proof:

$\Rightarrow$) See the soundness theorem (Theorem 4.1).

$\Leftarrow$) The proof is by induction on the length i of the chain $Q_1 \ldots Q_m \leftrightarrow^i_{E^*} true$.

Base: if $i = 1$, then $m = 1$ and there is a fact rule $A \to true$ such that $Q_1 = A\sigma$. It follows that $E \vdash_{LC} Q_1$ in one step.

Induction hypothesis: $\forall i, 1 < i \le l$, $Q_1 \ldots Q_m \leftrightarrow^i_{E^*} true$ implies $E \vdash_{LC} Q_1 \ldots Q_m$.

Induction step: if $i = l+1$, then $Q_1 \ldots Q_m \leftrightarrow_{E^*} \bar{C} \leftrightarrow^l_{E^*} true$ and we consider two cases depending on the direction of the first step:

1. if $Q_1 \ldots Q_m \to_{E^*} \bar{C} \leftrightarrow^l_{E^*} true$, then the LC interpreter can derive the goal $\bar{C}$ from the query $Q_1 \ldots Q_m$ by applying this rewrite step. Since $E \vdash_{LC} \bar{C}$ holds by induction hypothesis, $E \vdash_{LC} Q_1 \ldots Q_m$ follows.

2. If $Q_1 \ldots Q_m \leftarrow_{E^*} \bar{C} \leftrightarrow^l_{E^*} true$, then there are two more cases.

   If an if-rule or a fact rule or $x \cdot x \to x$ or $x \cdot true \to x$ reduces $\bar{C}$ to $Q_1 \ldots Q_m$, then all the atoms Q_j, $1 \le j \le m$, occur in $\bar{C}$, i.e. $\bar{C}$ is $Q_1 \ldots Q_m \bar{D}$. Since $\bar{C} \leftrightarrow^l_{E^*} true$, it follows that $Q_1 \ldots Q_m \leftrightarrow^k_{E^*} true$ for some $k < l$. By induction hypothesis we have that $E \vdash_{LC} Q_1 \ldots Q_m$.

   If an iff-rule $A \to B_1 \ldots B_n$ reduces $\bar{C}$ to $Q_1 \ldots Q_m$, then $\bar{C}$ is $C_1 \ldots C_p$, $C_1 = A\sigma$ and $Q_1 \ldots Q_m$ is $(B_1 \ldots B_n)\sigma C_2 \ldots C_p$. Since $C_1 = A\sigma$ and A is the head of an iff-rule, the predicate of C_1 is mutually exclusively defined. It follows that the same iff-rule $A \to B_1 \ldots B_n$ has to be applied to reduce C_1 in the equality chain between $\bar{C}$ and $true$. The product $(B_1 \ldots B_n)\sigma$ is reduced to $true$ by a path shorter than l steps, i.e. $(B_1 \ldots B_n)\sigma \leftrightarrow^k_{E^*} true$ for $k < l$. By induction hypothesis we have $E \vdash_{LC} B_1 \ldots B_n\sigma$. Also $C_2 \ldots C_p$ is reduced to $true$ by a path shorter than l steps, i.e. $C_2 \ldots C_p \leftrightarrow^j_{E^*} true$ for $j < l$. By induction hypothesis we have $E \vdash_{LC} C_2 \ldots C_p$. Since $Q_1 \ldots Q_m$ is $(B_1 \ldots B_n)\sigma C_2 \ldots C_p$, it follows that $E \vdash_{LC} Q_1 \ldots Q_m$. $\qquad\square$

The equivalence of operational and model theoretic semantics follows as a corollary:

Theorem 4.3 $\forall G \in \mathcal{B}$, $E \vdash_{LC} G$ *if and only if* $E^* \models G = true$.

Proof: $E \vdash_{LC} G$ if and only if $G \leftrightarrow^*_{E^*} true$ by Theorem 4.2 if and only if $E^* \models G = true$ by completeness of replacing equals by equals (Birkhoff's theorem). $\qquad\square$

5 The denotational semantics of rewrite programs

Similar to the denotational semantics of Prolog, the denotational semantics of a rewrite program E is the *least fixpoint* of a function associated to E. First, we introduce the lattice $\mathbb{B} = \{I' \mid I' = I \cup \{true\}, I \subseteq \mathcal{B}\}$. The order relation on $\mathbb{B}$ is set inclusion $\subseteq$, the greatest lower-bound operation

is intersection $\bigcap$ and the least upper-bound operation is union $\bigcup$. The bottom element is $\{true\}$ and the top element is $\mathcal{B} \cup \{true\}$. A function $T_E : \mathbb{B} \to \mathbb{B}$ is associated to a program E as follows:

Definition 5.1 *Given a rewrite program E, its associated function is the function $T_E : \mathbb{B} \to \mathbb{B}$ such that $P \in T_E(I)$ if and only if there exists in E a rule $A_1 \ldots A_n \leftrightarrow B_1 \ldots B_m$ $(n \geq 1, m \geq 1)$ such that $P = A_i \sigma$ and $\{A_1 \sigma, \ldots, A_{i-1}\sigma, A_{i+1}\sigma, \ldots, A_n\sigma, B_1\sigma, \ldots, B_m\sigma\} \subseteq I$ for some i, $1 \leq i \leq n$, and some ground substitution σ. (The double arrow $\leftrightarrow$ means there is no distinction between the left-hand side and the right-hand side.)*

Lemma 5.1 *Given a rewrite program E, T_E is continuous, that is, for every nondecreasing chain $X_1 \subseteq X_2 \subseteq \ldots$ of elements in $\mathbb{B}$, $T_E(\bigcup\{X_i \mid i < \omega\}) = \bigcup\{T_E(X_i) \mid i < \omega\}$.*

Proof: let $X_1 \subseteq X_2 \subseteq \ldots$ be a chain in $\mathbb{B}$.

1. $T_E(\bigcup\{X_i \mid i < \omega\}) \subseteq \bigcup\{T_E(X_i) \mid i < \omega\}$:

   $P \in T_E(\bigcup\{X_i \mid i < \omega\})$ if and only if there exists a ground instance $A_1\sigma \ldots A_{j-1}\sigma P A_{j+1}\sigma \ldots A_n\sigma \leftrightarrow B_1\sigma \ldots B_m\sigma$ of a rule in E and $\{A_1\sigma \ldots A_{j-1}\sigma, A_{j+1}\sigma \ldots A_n\sigma, B_1\sigma \ldots B_m\sigma\} \subseteq \bigcup\{X_i \mid i < \omega\}$. Then $\{A_1\sigma \ldots A_{j-1}\sigma, A_{j+1}\sigma \ldots A_n\sigma, B_1\sigma \ldots B_m\sigma\} \subseteq X_i$ for some i, so that $P \in T_E(X_i)$ and $P \in \bigcup\{T_E(X_i) \mid i < \omega\}$.

2. $\bigcup\{T_E(X_i) \mid i < \omega\} \subseteq T_E(\bigcup\{X_i \mid i < \omega\})$:

   $P \in \bigcup\{T_E(X_i) \mid i < \omega\}$ if and only if $P \in T_E(X_i)$ for some i if and only if there exists a ground instance $A_1\sigma \ldots A_{j-1}\sigma P A_{j+1}\sigma \ldots A_n\sigma \leftrightarrow B_1\sigma \ldots B_m\sigma$ and $\{A_1\sigma \ldots A_{j-1}\sigma, A_{j+1}\sigma \ldots A_n\sigma, B_1\sigma \ldots B_m\sigma\} \subseteq X_i$. Since $X_i \subseteq \bigcup\{X_i \mid i < \omega\}$, it follows that $P \in T_E(\bigcup\{X_i \mid i < \omega\})$. $\square$

It follows that T_E has the properties of continuous functions on a lattice. Namely the least fixpoint of T_E is an ordinal power of T_E:

$$lfp(T_E) = T_E \uparrow \omega,$$

where ordinal powers are defined on $\mathbb{B}$ in the usual way:

$$T \uparrow 0 = \{true\}$$

$$T \uparrow n = \begin{cases} T(T \uparrow (n-1)) & \text{if } n \text{ is a successor ordinal} \\ \bigcup\{T \uparrow k \mid k < n\} & \text{if } n \text{ is a limit ordinal.} \end{cases}$$

For example, the least fixpoint of the rewrite program for the *ancestor* relation is obtained as follows:

$$T_E(\{true\}) = \{true, parent(jb, lc), parent(jb, gg), parent(gg, wm)\}$$

$$T_E^2(\{true\}) = T_E(\{true\}) \cup \{ancestor(jb, lc), ancestor(jb, gg), ancestor(gg, wm)\}$$

$$lfp(T_E) = T_E^3(\{true\}) = T_E^2(\{true\}) \cup \{ancestor(jb, wm)\}.$$

6   Equivalence of operational and denotational semantics

The denotational semantics $lfp(T_E)$ of a program E relates to its model theoretic semantics as follows:

Theorem 6.1 $\forall Q_1 \ldots Q_m \in \mathcal{B}$, $Q_1 \ldots Q_m \leftrightarrow^*_{E^*} true$ *if and only if* $\forall i$, $1 \leq i \leq m$, $Q_i \in lfp(T_E)$.

Proof:

$\Leftarrow$) We prove that if $Q_i \in lfp(T_E)$, then $Q_i \leftrightarrow^*_{E^*} true$. Then, $Q_1 \ldots Q_m \leftrightarrow^*_{E^*} true$ follows.

If $Q_i \in lfp(T_E)$, then there exists a $j \geq 1$ such that $Q_i \in T^j_E(\{true\})$ and $Q_i \notin T^k_E(\{true\})$ for all $k < j$, i.e. j is the minimum power such that Q_i is included. The proof is done by induction on this power j.

Base: if $j = 1$, then there exists a fact rule $A \rightarrow true$ such that $Q_i = A\sigma$ for some ground substitution σ. It follows that $Q_i \leftrightarrow_{E^*} true$.

Induction hypothesis: $\forall j, 1 < j \leq l$, $Q_i \in T^j_E(\{true\})$ implies $Q_i \leftrightarrow^*_{E^*} true$.

Induction step: if $j = l+1$, then there exists a ground instance $A_1\sigma \ldots A_{k-1}\sigma Q_i A_{k+1}\sigma \ldots A_n\sigma \leftrightarrow B_1\sigma \ldots B_m\sigma$ of a rule in E such that $\{A_1\sigma \ldots A_{k-1}\sigma, A_{k+1}\sigma \ldots A_n\sigma, B_1\sigma \ldots B_m\sigma\} \subseteq T^l_E(\{true\})$. By induction hypothesis, all the atoms in this set are E^*-equivalent to $true$. It follows that $Q_i \leftrightarrow^*_{E^*} true$ as well.

$\Rightarrow$) The proof is done by induction on the length j of $Q_1 \ldots Q_m \leftrightarrow^j_{E^*} true$.

Base: if $j = 1$, then $m = 1$ and $Q_1 = A\sigma$ for a fact rule $A \rightarrow true$. It follows that $Q_1 \in T_E(\{true\})$ and $Q_1 \in lfp(T_E)$.

Induction hypothesis: $\forall j, 1 < j \leq l$, $Q_1 \ldots Q_m \leftrightarrow^j_{E^*} true$ implies $Q_i \in lfp(T_E)$ for $1 \leq i \leq m$.

Induction step: if $j = l+1$, then $Q_1 \ldots Q_m \leftrightarrow_{E^*} \bar{C} \leftrightarrow^l_{E^*} true$ and we have two cases as in the proof of Theorem 4.2.

1. Let us consider $Q_1 \ldots Q_m \rightarrow_{E^*} \bar{C} \leftrightarrow^l_{E^*} true$. If $x \cdot x \rightarrow x$ or $x \cdot true \rightarrow x$ reduces $Q_1 \ldots Q_m$ to $\bar{C}$, all the atoms Q_i, $1 \leq i \leq m$ occur in $\bar{C}$ and so by induction hypothesis all of them are in $lfp(T_E)$. If $A \rightarrow true$ reduces $Q_1 \ldots Q_m$ to $\bar{C}$, the atom Q_k matched by A is in $T_E(\{true\})$ and so in $lfp(T_E)$. All the remaining atoms $Q_2 \ldots Q_m$ are in $\bar{C}$ and so in $lfp(T_E)$ by induction hypothesis. If $AB_1 \ldots B_n \rightarrow B_1 \ldots B_n$ applies, then $(AB_1 \ldots B_n)\sigma = Q_1 Q_2 \ldots Q_{n+1}$ with $n + 1 < m$ and C is $Q_2 \ldots Q_m$. The atoms $Q_2 \ldots Q_m$ are in $lfp(T_E)$ by induction hypothesis. Since $(B_1 \ldots B_n)\sigma = Q_2 \ldots Q_{n+1}$, the atoms $B_1\sigma \ldots B_n\sigma$ are in $lfp(T_E)$. Let p be the minimum power such that all the $B_k\sigma$'s are in $T^p_E(\{true\})$. It follows that $Q_1 \in T^{p+1}_E(\{true\})$ and so in $lfp(T_E)$. If an iff-rule $A \rightarrow B_1 \ldots B_n$ reduces $Q_1 \ldots Q_m$ to $\bar{C}$, then $Q_1 = A\sigma$ and $\bar{C}$ is $(B_1 \ldots B_n)\sigma Q_2 \ldots Q_m$. By induction hypothesis, all the atoms in $\bar{C}$ are in $lfp(T_E)$. Let p be the minimum power such that all the $B_k\sigma$'s are in $T^p_E(\{true\})$. It follows that $Q_1 \in T^{p+1}_E(\{true\})$ and so in $lfp(T_E)$.

2. If $Q_1 \ldots Q_m \leftarrow_{E^*} \bar{C} \leftrightarrow^l_{E^*} true$ and an if-rule or a fact rule or $x \cdot x \rightarrow x$ or $x \cdot true \rightarrow x$ reduces $\bar{C}$ to $Q_1 \ldots Q_m$, all the atoms Q_i, $1 \leq i \leq m$, occur in $\bar{C}$. By induction hypothesis, all of them are in $lfp(T_E)$. If an iff-rule $A \rightarrow B_1 \ldots B_n$ applies, then $\bar{C}$ is $C_1 \ldots C_h$, $C_1 =$

$A\sigma$ and $Q_1 \ldots Q_m$ is $(B_1 \ldots B_n)\sigma C_2 \ldots C_h$. By induction hypothesis, we have that all the atoms in $\bar{C}$ are in $lfp(T_E)$. Only the atoms in $(B_1 \ldots B_n)\sigma$ are missing. However, since an iff-rule applies, it means that the predicate of C_1 is mutually exclusively defined. It follows that in the chain $\bar{C} \leftrightarrow^l_{E*} true$, this rule must be applied, i.e., the proof has the form $\bar{C} \leftrightarrow^{j_1}_{E*} \ldots (B_1 \ldots B_n)\sigma \ldots \leftrightarrow^{j_2}_{E*} true$, where $j_2 < l$. By applying the induction hypothesis to the last subchain we get that all the $B_k\sigma$'s are in $lfp(T_E)$ as well. □

From this result and the previously shown equivalence of operational and model theoretic semantics, the equivalence of operational and denotational semantics follows:

Theorem 6.2 $\forall G \in \mathcal{B}$, $G \in lfp(T_E)$ if and only if $E \vdash_{LC} G$.

Proof: $G \in lfp(T_E)$ if and only if $G \leftrightarrow^*_{E*} true$ by Theorem 6.1 if and only if $E \vdash_{LC} G$ by Theorem 4.2. □

7 Comparison with Prolog

The above fixpoint characterization of rewrite programs is basically equivalent to the fixpoint characterization of Prolog programs. The fixpoint semantics of a Prolog program P is $lfp(T_P) = T_P \uparrow \omega$, where T_P is the function $T_P : \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$ such that $A \in T_P(I)$ if and only if there exists in P a clause $A' : -B_1 \ldots B_m$ $(m \geq 0)$ such that $A = A'\sigma$ and $\{B_1\sigma \ldots B_m\sigma\} \subseteq I$ for some ground substitution σ [17, 1]. The following theorem shows that the two semantics are indeed the same. We write $E \equiv P$ and we say that the rewrite program E corresponds to the Prolog program P if they define the same predicates.

Theorem 7.1 If $E \equiv P$, then $lfp(T_E) = lfp(T_P)$.

Proof:

1. $lfp(T_E) \subseteq lfp(T_P)$.

   If $G \in lfp(T_E)$, then there exists an $i \geq 1$ such that $G \in T_E^i(\{true\})$ and $\forall j < i, G \notin T_E^j(\{true\})$. The proof is done by induction on the power i.

   Base: if $i = 1$, then there exists a fact rule $A \to true$ in E such that $G = A\sigma$ for some ground substitution σ. Since $E \equiv P$, the fact A. belongs to the Prolog program P and $G \in lfp(T_P)$.

   Induction hypothesis: $\forall i, 1 < i \leq l$, $G \in T_E^i(\{true\})$ implies $G \in lfp(T_P)$.

   Induction step: if $i = l + 1$, there exists a ground instance $C_1\sigma \ldots C_{j-1}\sigma G C_{j+1}\sigma \ldots C_p\sigma \leftrightarrow D_1\sigma \ldots D_q\sigma$ of a rule in E whose atoms but G are in $T_E^l(\{true\})$. By induction hypothesis all the atoms in this set belong to $lfp(T_P)$. It follows that there is some $k \geq 1$ such that $\{C_1\sigma \ldots C_{j-1}\sigma, C_{j+1}\sigma \ldots C_p\sigma, D_1\sigma \ldots D_q\sigma\} \subseteq T_P^k(\emptyset)$. The above ground rule is either an instance of an if-rule $AB_1 \ldots B_n \to B_1 \ldots B_n$ or an instance of an iff-rule $A \to B_1 \ldots B_n$. Since $P \equiv E$, in both cases the Prolog program P contains the corresponding clause A :

16

$-B_1 \ldots B_n$. In the if-rule case, if $G = B_h \sigma$ for some $h, 1 \le h \le n$, then it is in $lfp(T_P)$ by induction hypothesis, because each B_h occurs twice in the rule. If $G = A\sigma$, then $G \in T_P^{k+1}(\emptyset)$ and $G \in lfp(T_P)$. In the iff-rule case, if $G = A\sigma$, then $G \in T_P^{k+1}(\emptyset)$ and $G \in lfp(T_P)$. If $G = B_h \sigma$ for some $h, 1 \le h \le n$, then $A\sigma$ is already in $T_P^k(\emptyset)$. Since the A's predicate is mutually exclusively defined, $A\sigma$ must have been included in $T_P^k(\emptyset)$ because of the clause $A : -B_1 \ldots B_n$ and $G = B_h \sigma$ must be in $T_P^z(\emptyset)$ for some $z < k$, so that $G \in lfp(T_P)$.

2. $lfp(T_P) \subseteq lfp(T_E)$.

   The proof is analogous to the previous one, applying the same induction argument on the power of T_P and exploiting the correspondence between the two programs. □

This theorem establishes that a Prolog program and a rewrite program defining the same predicates have the same model. For a ground atom G, $E \vdash_{LC} G$, $G \leftrightarrow_{E^*}^* true$, $E^* \models G = true$, $G \in lfp(T_E)$, $G \in lfp(T_P)$, $P \models G$ and $P \vdash_{Prolog} G$ are all equivalent. The behaviour of the two programs P and E may be different, though. The rewrite program E may generate less answers than the corresponding Prolog program P. However, for all answers given by P there is an answer given by E which is E-equivalent. In order to obtain this result, we first prove that all the answers given by linear completion are also given by Prolog. This result follows from completeness of SLD-resolution.

Theorem 7.2 *If $E \equiv P$ and $E \vdash_{LC} Q_1 \ldots Q_m \sigma$, there exists an answer θ, $P \vdash_{Prolog} Q_1 \ldots Q_m \theta$, such that $\sigma = \theta \rho$ for some substitution ρ.*

Proof: if $E \vdash_{LC} Q_1 \ldots Q_m \sigma$, then σ is a correct answer substitution by soundness of linear completion. Then by completeness of SLD-resolution there exists a computed answer substitution θ, $P \vdash_{Prolog} Q_1 \ldots Q_m \theta$ and a substitution ρ, such that $\sigma = \theta \rho$. □

We now prove that linear completion is also complete, by proving that all Prolog answers are represented by some answers given by linear completion.

In the following we assume that all queries are single literal queries. There is no loss of generality because a query $Q_1 \ldots Q_m \rightarrow answer(\bar{x})$ can be written as a single literal query by introducing a new predicate symbol N, a new program rule $N \rightarrow Q_1 \ldots Q_m$ and the query $N \rightarrow answer(\bar{x})$.

First of all we prove that rewrite programs and Prolog programs give the same answers if linear completion is restricted to overlap steps only, i.e. no simplification is performed. This is straightforward, since overlap steps and resolution steps clearly correspond:

Theorem 7.3 *Let LC' be a subset of the linear completion interpreter performing overlap steps only. If $E^* \models G\theta = true$, i.e. θ is a correct answer for G, there exists a computed answer σ, $E \vdash_{LC'} G\sigma$, such that $\theta = \sigma \rho$ for some substitution ρ.*

Proof: since SLD-resolution is complete, we prove the completeness of LC' by showing that if $E \vdash_{LC'} G\sigma$, then $P \vdash_{Prolog} G\sigma$ and vice versa. Overlap steps correspond to resolution steps and generate the same unifiers. Since the answer substitutions are given by the composition of

the unifiers, the two programs generate the same answers. It is irrelevant whether a Prolog step corresponds to an overlap step by an if-rule or to an overlap step by an iff-rule. The only difference is that an iff-overlap adds the body of the rule to the left side of the new goal rule only, whereas an if-overlap adds it to both sides. The occurrence of an atom on both sides of a goal does not affect overlap steps. When the left occurrence of an atom is eliminated by an overlap step, its right occurrence is instantiated by the mgu of the overlap step and will eventually be eliminated by an overlap step whose mgu does not instantiate the variables in the goal. Note that overlaps with answer rules do not occur, since *answer* atoms may occur on the left-hand side of a goal only as effect of previous simplification by goals, which is not performed by LC'. □

In order to prove an analogous result for linear completion with simplification we first need to prove two more lemmas.

Lemma 7.1 *If linear completion generates a computation path* $(E; G \to answer(\bar{x}); \emptyset) \vdash^*_{LC} (E; \bar{W} \to \bar{V}; {}_{-}) \vdash^*_{LC} (E; H; {}_{-}) \vdash_{LC} (E; \bar{R} \leftrightarrow \bar{R}; {}_{-})$, *where the goal* H *is simplified to an identity by its ancestor* $\bar{W} \to \bar{V}$, *then* H *is* $\bar{Z}(\bar{W}\lambda) \to \bar{Z}(\bar{V}\lambda)$ *for some substitution* λ *and literals* $\bar{Z}$. *(The hyphen* ${}_{-}$ *in* $(E; H; {}_{-})$ *means that the third component is not relevant.)*

Proof: let H be $\bar{S}\bar{X} \to \bar{S}\bar{Y}$, where $\bar{S}$ are the literals occurring on both sides, the literals in $\bar{X}$ occur on the left side only, and the literals in $\bar{Y}$ occur on the right side only. If $\bar{W} \to \bar{V}$ applies to both sides of H, then $\bar{W}$ matches a subset of $\bar{S}$, i.e. $\bar{S} = (\bar{W}\lambda)\bar{Z}$ for some substitution λ and $\bar{W} \to \bar{V}$ rewrites H to $(\bar{V}\lambda)\bar{Z}\bar{X} \to (\bar{V}\lambda)\bar{Z}\bar{Y}$ against the hypothesis that $\bar{W} \to \bar{V}$ rewrites H to an identity. It follows that $\bar{W} \to \bar{V}$ applies to one side of H only.

We assume that $\bar{W} \to \bar{V}$ applies to the left side of H. The case where it applies to the right side is symmetrical. It follows that H is reduced to $\bar{S}\bar{Y} \to \bar{S}\bar{Y}$. If $\bar{W}$ matches only a subset of $\bar{X}$, then the unmatched subset of $\bar{X}$ would still appear on the left side and not on the right side after the simplification step, contradicting the hypothesis that $\bar{W} \to \bar{V}$ rewrites H to an identity. It follows that $\bar{W}$ must match $\bar{X}$ completely and possibly a subset of $\bar{S}$: we have $\bar{W} = \bar{W}_1\bar{W}_2$, $\bar{S} = (\bar{W}_1\lambda)\bar{Z}$ and $\bar{X} = (\bar{W}_2\lambda)$. The left side of H is $(\bar{W}_1\lambda)\bar{Z}(\bar{W}_2\lambda)$ and it is rewritten to $\bar{Z}(\bar{V}\lambda)$. Furthermore, $\bar{Z}(\bar{V}\lambda) = \bar{S}\bar{Y}$, since the left side of H must be rewritten to its right side. Since $\bar{S} = (\bar{W}_1\lambda)\bar{Z}$, it follows that $(\bar{V}\lambda) = (\bar{W}_1\lambda)\bar{Y}$, i.e. $\bar{V} = \bar{W}_1\bar{W}_3$, where $\bar{W}_3\lambda = \bar{Y}$. It follows that $\bar{W} \to \bar{V}$ is $\bar{W}_1\bar{W}_2 \to \bar{W}_1\bar{W}_3$ and H is $\bar{Z}(\bar{W}_1\bar{W}_2\lambda) \to \bar{Z}(\bar{W}_1\bar{W}_3\lambda)$. □

The following lemma shows that linear completion with simplification is refutationally complete. First, we introduce some terminology used in the proof: a computation path starting at the root, i.e. the query, and halting with an answer is a *successful path*. A computation path starting at the root and halting with a goal such that no inference rule applies is an *unsuccessful path*. There are three kinds of unsuccessful path in an LC tree:

- unsuccessful paths where the goal is reduced to an identity, which is next deleted by a *Delete* step,

- unsuccessful paths where the goal is reduced to an equation containing only *answer* literals to which no inference rule applies and

18

- unsuccessful paths where the last goal still contains literals whose predicate is not *answer*, but no program rule applies to them.

The first two kinds of unsuccessful paths are determined by simplification and therefore they do not occur in computations by LC'. The third kind corresponds to the notion of *finite failure* in Prolog and is the only kind of unsuccessful path in an LC' computation. We use Δ to indicate either an identity or an equation containing only *answer* literals to which no inference rule applies.

Lemma 7.2 *Given a query G, if there exists an answer θ generated by LC', i.e. $E \vdash_{LC'} G\theta$, then there exists an answer σ generated by LC, i.e. $E \vdash_{LC} G\sigma$.*

Proof: the proof is done by way of contradiction. Assuming that LC does not generate any answer, we consider the tree T_{LC} generated by LC for the query G. Since LC does not generate any answer, all paths in T_{LC} are unsuccessful. LC differs from LC' because of simplification of goals by program rules, by their ancestors and by previously generated answers. Simplification by rules in the program is irrelevant since a simplification step by a program rule is an overlap step where the unifier is a matching substitution. Simplification and overlap with previously generated answers do not apply because we assume that LC does not generate any answer for the query G. Therefore, we only have to consider the case where all the successful paths generated by LC' are pruned by simplification by ancestor goal rules in LC. More precisely, all the successful paths generated by LC' are replaced as an effect of simplification by ancestors by unsuccessful paths ending with a Δ goal:

$$(E; G \rightarrow answer(\bar{x}); \emptyset) \vdash^*_{LC} (E; H'; \_) \vdash^*_{LC} (E; H; \_) \vdash_{LC} (E; \Delta; \_),$$

where H' simplifies H to Δ.

For any such path α, we denote by $I(\alpha)$ the set of all the successful paths in the computation tree generated by LC' which have the subpath $(E; G \rightarrow answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; H'; \_) \vdash^*_{LC'} (E; H; \_)$ as a prefix, i.e. all the paths in the form:

$$(E; G \rightarrow answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; H'; \_) \vdash^*_{LC'} (E; H; \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta \rightarrow true; \_).$$

Given a set of paths A, we denote by $min(A)$ the shortest path in the set A. If it is not unique, we just select one among the shortest paths. Then $min(I(\alpha))$ is the shortest path in the set $I(\alpha)$. We consider the path α^*, defined as follows:

$$\alpha^* = min(\{min(I(\alpha)) | \alpha \in T_{LC}\}).$$

α^* has the form

$$(E; G \rightarrow answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; H'; \_) \vdash^*_{LC'} (E; H; \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta \rightarrow true; \_).$$

By our assumptions, α^* is pruned by goal simplification in the tree T_{LC}. We show that if this is the case, then LC' generates a successful path shorter than α^*, which is a contradiction. There are two cases:

case 1: the goal H is reduced by its ancestor H' to an identity,

case 2: the goal H is reduced by its ancestor H' to an equation containing only *answer* literals to which no inference rule applies.

1. By Lemma 7.1, it follows that if H' is $\bar{W} \to \bar{V}$, then H is $\bar{Z}(\bar{W}\lambda) \to \bar{Z}(\bar{V}\lambda)$ for some substitution λ and literals $\bar{Z}$. We know that $(E; \bar{Z}(\bar{W}\lambda) \to \bar{Z}(\bar{V}\lambda); \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta \to true; \_)$. The literals $\bar{Z}$ must be eliminated in order to achieve a solution. Since the order of literals selection for overlap/resolution does not affect the answers generated by LC' or Prolog, we can rearrange the order of steps in the path α^* in such a way that the literals in $\bar{Z}$ are eliminated first. This does not modify the length of the path. It follows that α^* is $(E; G \to answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; \bar{W} \to \bar{V}; \_) \vdash^*_{LC'} (E; \bar{Z}(\bar{W}\lambda) \to \bar{Z}(\bar{V}\lambda); \_) \vdash^*_{LC'} (E; \bar{W}\lambda \to \bar{V}\lambda; \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta \to true; \_)$. Such a path cannot be the shortest path to a solution. If we unify a literal $P(\bar{t})\lambda$ in $\bar{W}\lambda$ with a head of a rule $P(\bar{s})$, we can also unify $P(\bar{t})$ in $\bar{W}$ with $P(\bar{s})$. It follows that there is another successful path $(E; G \to answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; \bar{W} \to \bar{V}; \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta' \to true; \_)$ which is obtained by applying to $\bar{W} \to \bar{V}$ the same clauses applied to $\bar{W}\lambda \to \bar{V}\lambda$ to get the solution. This path is shorter than α^*.

2. Let us assume now that H' reduces H to an equation containing only *answer* literals. It follows that H' has the form $\bar{W} \to answer(\_)$ and H is $\bar{W}\lambda \to answer(\_)$ or $\bar{W}\lambda answer(\_) \to \bar{W}\lambda$. There may be more than one *answer* literal in H' and H, but this is irrelevant. It follows that α^* is $(E; G \to answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; \bar{W} \to answer(\_); \_) \vdash^*_{LC'} (E; \bar{W}\lambda \to answer(\_); \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta \to true; \_)$. By the same argument applied in the previous case, there is a shorter successful path $(E; G \to answer(\bar{x}); \emptyset) \vdash^*_{LC'} (E; \bar{W} \to answer(\_); \_) \vdash^*_{LC'} (E; answer(\bar{x})\theta' \to true; \_)$. $\qquad \square$

We can finally state our completeness results:

Theorem 7.4 *If* $E^* \models G\theta = true$, *i.e.* θ *is a correct answer for* G, *there exists a computed answer* σ, $E \vdash_{LC} G\sigma$, *such that* $G\theta \leftrightarrow^*_{E^*} G\sigma\rho$ *for some substitution* ρ.

Proof: if $E^* \models G\theta = true$, then by Theorem 7.3, there exists a computed answer τ, $E \vdash_{LC'} G\tau$, such that $G\theta = G\tau\rho$. By Lemma 7.2, there exists an answer for the same query computed by LC: $E \vdash_{LC} G\sigma$. $E \vdash_{LC'} G\tau$ implies that $G\tau \leftrightarrow^*_{E^*} true$. Similarly, $E \vdash_{LC} G\sigma$ implies $G\sigma \leftrightarrow^*_{E^*} true$. It follows that $G\tau \leftrightarrow^*_{E^*} G\sigma$. Then $G\tau\rho \leftrightarrow^*_{E^*} G\sigma\rho$ or $G\theta \leftrightarrow^*_{E^*} G\sigma\rho$. $\qquad \square$

Theorem 7.5 *If* $E \equiv P$, *if* $P \vdash_{Prolog} G\theta$, *then there exists an answer* σ *given by linear completion,* $E \vdash_{LC} G\sigma$, *such that* $G\theta \leftrightarrow^*_{E^*} G\sigma\rho$ *for some substitution* ρ.

Proof: if $P \vdash_{Prolog} G\theta$, then θ is a correct answer substitution by soundness of SLD-resolution. Then by Theorem 7.4, there exist a computed answer substitution σ, $E \vdash_{LC} G\sigma$ and a substitution ρ, such that $G\theta \leftrightarrow^*_{E^*} G\sigma\rho$. $\qquad \square$

To summarize, we have proved first that a rewrite program E and the corresponding Prolog program P have the same fixpoint of ground true facts and therefore, the same semantics. Next, we have proved a completeness theorem (Theorem 7.4) establishing that for every correct answer substitution θ to a query G, linear completion computes an answer σ such that $G\theta \leftrightarrow^*_{E^*} G\sigma\rho$, i.e. σ is "E-equivalent" to θ. As a consequence (Theorem 7.5), even if linear completion may return fewer answers than Prolog, for every answer substitution θ generated by Prolog, linear completion computes an "E-equivalent" answer σ.

8 Comparisons with related work: some loop checking mechanisms in Prolog

Rewrite systems have been applied in both functional and logic programming and there is a vast literature on this subject. In this paper we have considered only a special class of rewrite systems. The interpreter is *linear completion*, which is a restriction of Knuth-Bendix completion. The rewrite rules are *equivalences* between conjunctions of first-order atoms, with no equality predicate. Therefore, our language is relational like pure Prolog. The inference system of linear completion includes simplification and overlap. An overlap step intuitively corresponds to a resolution step and therefore, we have carefully avoided calling it *narrowing*. The name narrowing was introduced first in [21, 18] and it refers to a paramodulation step by an oriented equation whose sides are terms. Our method is not related to those using narrowing since we do not use equations or rewrite rules between first order terms. It is closer instead to those given in [10, 9, 11, 20]. The approach proposed in the first three papers does not allow simplification, thus cannot fully utilize the power of rewriting. Our approach is similar to [20], although they did not study the case where a predicate is defined by implications and not by logical equivalences. Neither did they explain the semantics of their method. The interpreter of [20] includes also an inference rule for unit subsumption between equations. Such an inference rule is not needed in our method, because all rules in a rewrite programs are oriented and therefore, subsumption reduces to a subcase of simplification.

One of the positive effects of simplification in executing rewrite programs is loop avoidance. Techniques for loop detection and avoidance in the execution of Prolog programs have received considerable attention. Here we focus on the results in [2, 3] since their loop-checking mechanisms are based on *subsumption of goals by ancestors* and therefore, they may yield pruning effects similar to those of simplification.

Loop-checking mechanisms are described in [2, 3] according to the following terminology: a loop checking mechanism L is said to be

- *complete*, if it prunes all infinite derivations,

- *sound*, if no answer substitution is lost, that is, for all answers σ to a query G generated by Prolog, there is an answer σ' generated by Prolog with L such that $G\sigma = G\sigma'\rho$ for some substitution ρ,

- *weakly sound*, if whenever there are answers generated by Prolog to a query G, there is at least one answer to G generated by Prolog with L.

Completeness is a very strong requirement and, indeed, it is proved in [2] that no weakly sound and complete loop check exists for general Prolog programs, even in the absence of function symbols.

Weak soundness does not seem to be a sufficiently significant property: a weakly sound loop check is guaranteed to yield at least one answer if there are any, but no information is given about how the generated substitution is possibly related to those which are not given. Our Theorem 7.5 instead, establishes more than weak soundness, since it says that for all Prolog answers σ to a query G, there exists an answer θ by linear completion such that $G\sigma \leftrightarrow^*_{E^*} G\theta\rho$ for some substitution ρ.

Three kinds of loop checking mechanisms are introduced in [2, 3]. The first one is called *Contains a Variant/Instance of Atom (CVA/CIA)* check. The basic idea is to interrupt a derivation if the current goal contains an atom which is subsumed by an atom occurring in an ancestor goal. As observed in [2, 3], the CVA/CIA check is not even weakly sound. Take the *append* examples which we presented in Section 2: Prolog with CVA/CIA would interrupt the infinite derivation of the first *append* example, but, unlike linear completion, it would also halt with no answer the computation for the query

$$?- append(X, [b|Y], [a, b, c|Z]), size(X) > 3$$

of the third example. This behaviour is not correct and it shows that a loop check mechanism which is not weakly sound is unacceptable. Intuitively, the CVA/CIA checks are not correct because they check single atoms out of their contexts: the condition to halt the derivation applies to the *append* literal without taking the *size* literal into consideration. On the other hand, linear completion answers the above query because simplification takes the context into account, since it requires that the entire left-hand side of the ancestor match the current goal.

The second family of loop checks introduced in [2, 3] is obtained by replacing "atom" by "goal": a derivation is interrupted if the current goal is subsumed by one of its ancestors. These checks are called *Equal Variant of Goal (EVG)*, *Equal Instance of Goal (EIG)*, *Subsumed as Variant of Goal (SVG)* and *Subsumed as Instance of Goal (SIG)*, where "equal" means that the current goal is exactly an instance of one of its ancestors. These four loop checks are all proved to be weakly sound [2, 3], and they are complete for very restricted classes of programs defined in [2, 3]. As defined earlier, weak soundness just guarantees that at least one answer is generated. The following example from [2] shows that weak soundness is not satisfactory: given the program

$p(a).$

$p(Y) :- p(Z).$

and the query

$?- p(X).,$

Prolog generates first the answer $\{X \leftarrow a\}$, then the answer $\{X \leftarrow Y\}$ and then it loops forever. Prolog with any of the above weakly sound checks is able to find only the answer a as shown in Fig. 4, where the computation tree below $?- p(Z)$ is pruned because $p(Z)$ is subsumed by $p(X)$.
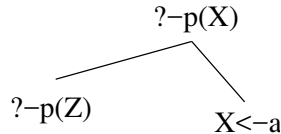

Figure 4: Prolog finds only answer a

If the current goal is subsumed by an ancestor, simplification applies. Therefore, one may expect that also linear completion loses all answers but $\{X \leftarrow a\}$. The result is instead very different: the rewrite program

22

$$p(a) \rightarrow true.$$

$$p(Y)p(Z) \rightarrow p(Z).$$

with the query

$$p(X) \rightarrow answer(X).,$$

generates the answers $\{X \leftarrow a\}$ and $\{X \leftarrow X\}$ and then halts as shown in Fig. 5.

```
                          p(X)->answer(X)
                        /                \
        answer(a)->true       p(Z),answer(X)->p(Z)
                                       |
                                       v
                         answer(Z),answer(X)->answer(Z)
                                       |  Z<-a
                              answer(X)->true
```

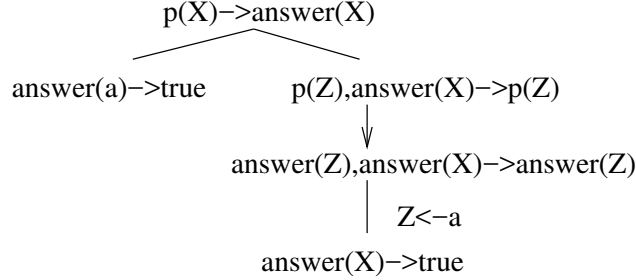Figure 5: Linear completion generates $\{X \leftarrow a\}$, $\{X \leftarrow X\}$, and halts

Linear completion correctly infers that $p(a)$ is true and that $p(X)$ is true for all X, since the second clause in the program says that if there exists an element Z such that $p(Z)$ is true, then $p(Y)$ is true for all Y. Operationally, the reason for this different behaviour is that simplification is not a mere pruning mechanism, but an additional inference rule: if a goal is simplified, the computation does not halt but continues with a reduced goal.

The third type of loop-checking mechanisms in [2, 3] is based on *resultants*: if G_0 is the query, G_i is the current goal and σ_i is the partial answer substitution computed on the path from G_0 to G_i, the resultant associated to G_i is $G_0\sigma_i$. According to the loop checks based on resultants, a derivation is interrupted if the current goal and its associated resultant are subsumed respectively by an ancestor goal and its resultant. These loop checks are called *EVR, EIR, SVR* and *SIR*. The resultant based loop checks are proved in [2, 3] to be sound in general and complete for very restricted classes of programs.

Loop checks based on variants rather than instances may be undesirably weak: for instance, the EVR and SVR checks would not prune the infinite derivation in our first *append* example, since the goals are variants, but the resultants are not. The EIR and SIR checks would prune it like linear completion.

The SIR loop check seems to be the one whose effects are the closest to those of simplification in linear completion. Intuitively, the reason for this similarity is that linear completion embeds some check on the resultants by recording partial answer substitutions in the *answer* literals. If the query has the form $G_0(\bar{x}) \rightarrow answer(\bar{x})$, the literal $answer(\bar{x}\sigma_i)$ in the goal G_i is exactly the resultant $G_0(\bar{x}\sigma_i)$. This is not the case if the query has the form $G_0(c[\bar{x}]) \rightarrow answer(\bar{x})$ for some context c, but such a query can be reformulated as $G_0(c[\bar{x}]) \rightarrow answer(c[\bar{x}])$, should it turn out to be convenient to carry more information in the *answer* literal. If an *answer* literal occurs in the left-hand side of the ancestor applied as simplifier, simplification automatically checks resultants as well. This explains intuitively why in the following example from [3] the loop check whose

behaviour is the closest to linear completion is SIR. Given the program

$a(0).$

$a(Y) :- a(0), c(Y).$

$b(1).$

$c(Z) :- b(Z), a(W).$

and the query $?- a(X).$,

the Prolog execution is pruned by the loop checks in [2, 3] as shown in Fig. 6, where each frame delimits the portion of the computation tree generated by Prolog augmented with the indicated loop check.
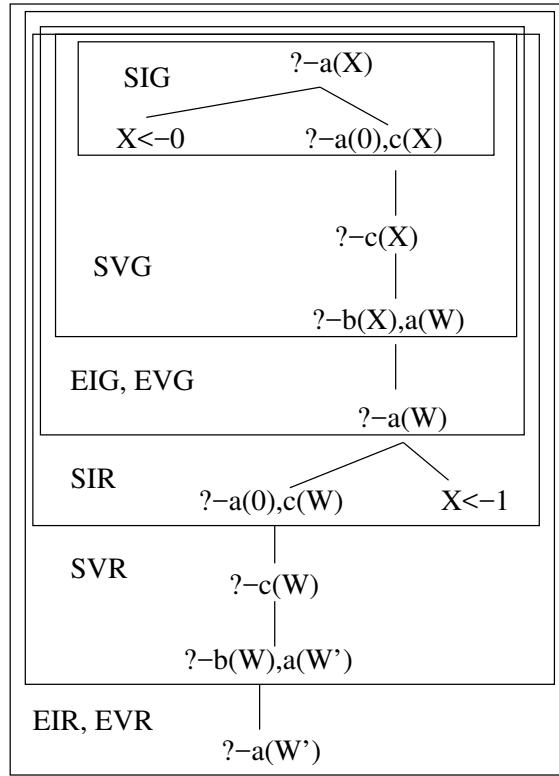

Figure 6: Loop checks prune the Prolog execution

In particular, the SIR check halts the derivation after the answers $\{X \leftarrow 0\}$ and $\{X \leftarrow 1\}$ are derived because the goal $?- a(0), c(W)$ and its resultant $a(1)$ are respectively a variant and an instance of the ancestor $?- a(0), c(X)$ and its resultant $a(X)$. Linear completion generates the two answers $\{X \leftarrow 0\}$ and $\{X \leftarrow 1\}$ and then it halts as shown in Fig. 7.

The SIG/SVG and EIG/EVG loop checks halt the computation too early and inhibit the generation of the answer $\{X \leftarrow 1\}$. The SVR, EIR and EVR loop checks give both solutions like linear completion and SIR, but they are less efficient, since they prune the computation much later. It is remarkable that linear completion saves exactly those paths which lead to a solution.

24

$a(0) \rightarrow true.$
$a(Y), a(0), c(Y) \rightarrow a(0), c(Y).$
$b(1) \rightarrow true.$
$c(Z) \rightarrow b(Z), a(W).$
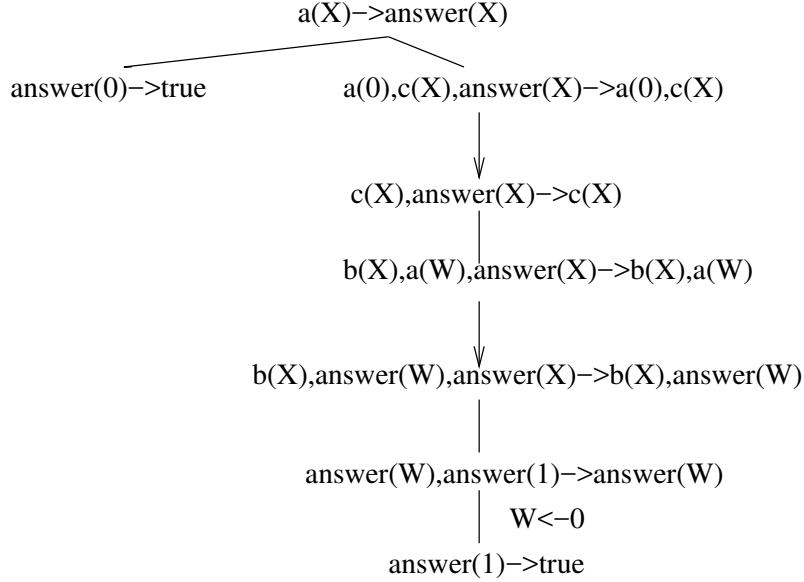$a(X) \rightarrow answer(X).$

Figure 7: Linear completion generates $\{X \leftarrow 0\}$, $\{X \leftarrow 1\}$, and halts

We do not have a result establishing an exact relationship between the pruning power of simplification in linear completion and the pruning power of the SIR loop check in Prolog. It seems to us that the significance of any comparison between linear completion and Prolog augmented with these loop checks is limited by the observation that the two approaches are very different in nature. The approach in [2, 3] consists in restricting the inference system of Prolog by adding loop checks. The concept of a loop check is purely procedural: the inference system is not modified, but just affected by the addition of an external control. This approach seems therefore somewhat artificial.

In our approach, simplification is a natural consequence of writing program units as equations. Simplification is not a pruning mechanism added to the inference rules, but an inference rule itself, with the capability of reducing the search space. Therefore, if simplification applies, the derivation is not interrupted, but it continues with the reduced goal. For instance, in our basic *append* example, Prolog with EIR or SIR does not loop, because of the external loop checking mechanism, whereas linear completion does not loop because *append* is defined by equivalences rather than by implications. In all the examples proposed in [2, 3], linear completion behaves as desired. Furthermore, simplification in linear completion uses uniformly program rules, answer rules and ancestor goals as simplifiers, and it allows us to optimize executions where no infinite derivation occurs, as shown by the *ancestor* example in Section 2.

Inference rules which reduce the search space have been recently called *contraction* inference rules [12]. Simplification is one such rule. Inference rules of this kind have proved to be extremely valuable in theorem proving and our work suggests that they may be considered in logic programming as well.

9 Discussion

In this paper we have given the operational and denotational semantics of a notion of rewrite programs. We have shown that its operational semantics, via linear completion, is both sound and complete with respect to the denotational semantics.

The main difference between rewrite programs and Prolog programs is that rewrite programs allow one to differentiate between predicates which are mutually exclusively defined and those which are not. A predicate is mutually exclusively defined if the head of each of its clauses is logically equivalent to its body. A typical such example is the usual definition of *append*. Rewrite programs, with their simplification power, can take advantage of these definitions to prevent certain infinite loops which are otherwise unavoidable in pure Prolog.

Rewrite programs have the curious property of being denotationally equivalent to Prolog on the ground level while yielding fewer answers in general. This is because certain answers equivalent under the equivalence relation defined by the program "collapse" into one. However, rewrite programs are also guaranteed not to lose any necessary answers. That is, they will indeed generate answers where there are some, as we have shown in both the examples and the theorems. One weakness of these results is that they use the E-equivalence relation $\leftrightarrow_E^*$. Any two answers are both equivalent to *true* and hence to each other. We have proved that if there are answers at least one will be generated, but we have not characterized exactly which answers are generated. This would require to find some refinement of $\leftrightarrow_E^*$ that partitions the class of all answers into finer subclasses. Still, we feel that when a predicate symbol is supposed to be mutually exclusively defined, our semantics is more desirable than Prolog since it captures the intended meaning more accurately.

It should not be too difficult to incorporate our treatment of mutually exclusively defined predicates into a Prolog interpreter. It requires eliminating a few backtracking points and keeping the ancestor goals around for simplification purposes. However, the cleaner semantics and the prevention of certain loops may justify the extra effort spent. We are also interested to see whether *negation* can be incorporated into our framework, since a negative fact $\neg A$ simply means a rule $A \rightarrow false$.

10 Acknowledgements

References

[1] Apt, K.R. and van Emden, M.H., Contributions to the Theory of Logic Programming, *J. ACM*, 29:841–863 (1982).

[2] Apt, K.R., Bol, R.N. and Klop, J.W., On the Safe Termination of Prolog programs, in G.Levi and M.Martelli (eds.), *Proceedings of the Sixth International Conference on Logic Programming*, 353–368, MIT Press, Cambridge MA, 1989.

[3] Bol, R.N., Apt, K.R. and Klop, J.W., On the Power of Subsumption and Context Checks, in A.Miola (ed.), *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, LNCS 429, 131–140, Capri, Italy, April 1990.

[4] Burstall, R.M., MacQueen, D.B. and Sannella, D.T., HOPE: an experimental applicative language, in *Conference Record of the 1980 LISP Conference*, 136–143, Stanford, California, 1980.

[5] Chang, C.L. and Lee, R.C.T., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[6] Debray, S.K. and Warren, D.S., Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems*, 11:451–481 (1989).

[7] Dershowitz, N., Orderings for term rewriting systems, *J. of Theoretical Computer Science*, 17:279–301 (1982).

[8] Dershowitz, N., Hsiang, J., Josephson, N.A. and Plaisted, D.A., Associative-commutative rewriting, in A.Bundy (ed.), *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 940–944, Karlsrue, Germany, 1983.

[9] Dershowitz, N., Computing with Rewrite Systems, *Information and Control*, 65:122–157 (1985).

[10] Dershowitz, N. and Josephson, N.A., Logic Programming by Completion, in *Proceedings of the Second International Conference on Logic Programming*, 313–320, Uppsala, Sweden, 1984.

[11] Dershowitz, N. and Plaisted, D.A., Equational Programming, in J.E.Hayes, D.Michie and J.Richards (eds.), *Machine Intelligence 11: The logic and acquisition of knowledge*, Chapter 2, 21-56, Oxford University Press, 1988.

[12] Dershowitz, N., A Maximal-Literal Unit Strategy for Horn Clauses, in M.Okada, S.Kaplan (eds.), *Proceedings of the Second International Workshop on Conditional and Typed Rewriting Systems*, Montréal, Canada, June 1990, LNCS 516, 14–25, 1991.

[13] Futatsugi, K., Goguen, J.A., Jouannaud, J.P. and Meseguer, J., Principles of OBJ2, in B.Reid (ed.) *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, 1985.

[14] Goguen, J.A. and Meseguer, J., Equality, types, modules and (why not?) generics for logic programming, *J. of Logic Programming*, 1:179–210 (1984).

[15] Green, C.C., The Application of Theorem-proving to Question-answering, Ph.D. Dissertation, Stanford University, Stanford, California, 1969.

[16] Hoffmann, C.M. and O'Donnell, M.J., Programming with Equations, *ACM Transactions on Programming Languages and Systems*, 4:83–112 (1982).

[17] Kowalski, R.A. and Van Emden, M.H., The Semantics of Predicate Logic as a Programming Language, *J. ACM*, 4:733–742 (1976).

[18] Lankford, D.S., Canonical inference, Memo ATP-32, Automatic Theorem Proving Project, University of Texas, Austin, Texas, December 1975.

[19] Lloyd, J.W., *Foundations of Logic Programming*, second edition, Springer Verlag, Berlin, 1987.

[20] Rety, P., Kirchner, C., Kirchner, H. and Lescanne, P., NARROWER: a new algorithm for unification and its application to logic programming, in J.P.Jouannaud (ed.), *Proceedings of the First International Conference on Rewrite Techniques and Applications*, LNCS 202, Dijon, France, May 1985.

[21] Slagle, J.R., Automated Theorem Proving with Simplifiers, Commutativity and Associativity, *J. ACM*, 21:622–642 (1974).