Operational and Denotational Semantics of Rewrite Programs

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

In this paper we present a new operational and denotational semantics for rewrite systems as logic programs. The main feature of our rewrite programs is that they allow us to define predicates not only by implications as in Prolog, but also by equivalences. We give a few examples showing how rewrite programs may turn out to be more effective than Prolog programs in capturing the user's intended semantics, because of this additional feature. Rewrite programs are interpreted by linear completion. Our definition of linear completion differs from the previous one [10, 9], because we allow simplification of goals by their ancestors. If simplification is allowed, programs given by equivalences and programs given by implications show a different behaviour. In the second part of the paper we give a fixpoint characterization of rewrite programs and we show that it captures both the proof theoretic and the model theoretic semantics. Rewrite programs and Prolog programs for the same predicates have the same fixpoint, i.e. the same set of ground true facts, although a rewrite program may give fewer answers. We show that this happens 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 explains why a smaller set of answers covers the same set of ground true facts.

1 Rewrite programs

Term rewriting systems have been widely applied in functional programming [5, 12, 13, 15] and to a less extent in logic programming [8, 10, 9, 11, 18]. 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 perform 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* [9]. A computation consists in generating an answer substitution

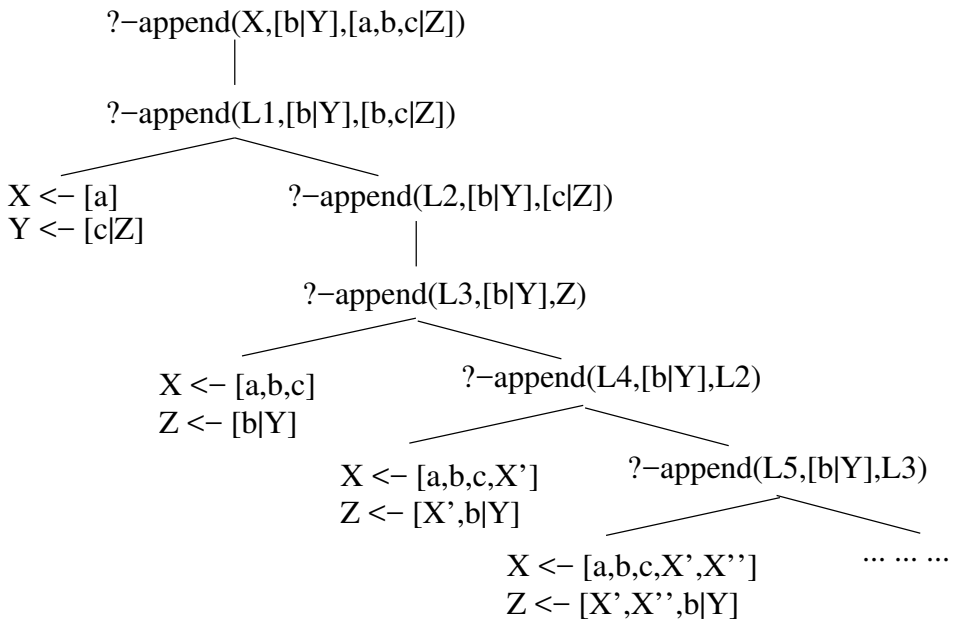
for a non ground query as it is done in Prolog.

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 this paper we present a more precise operational and denotational semantics and show how rewrite programs can avoid certain infinite loops which occur in similar Prolog 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 following Prolog program:

$append([\,], L, L).$
$append([X|L1], Y, [X|L2]) :- append(L1, Y, L2).$
with the query
$?- append(X, [b|Y], [a, b, c|Z]).$

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



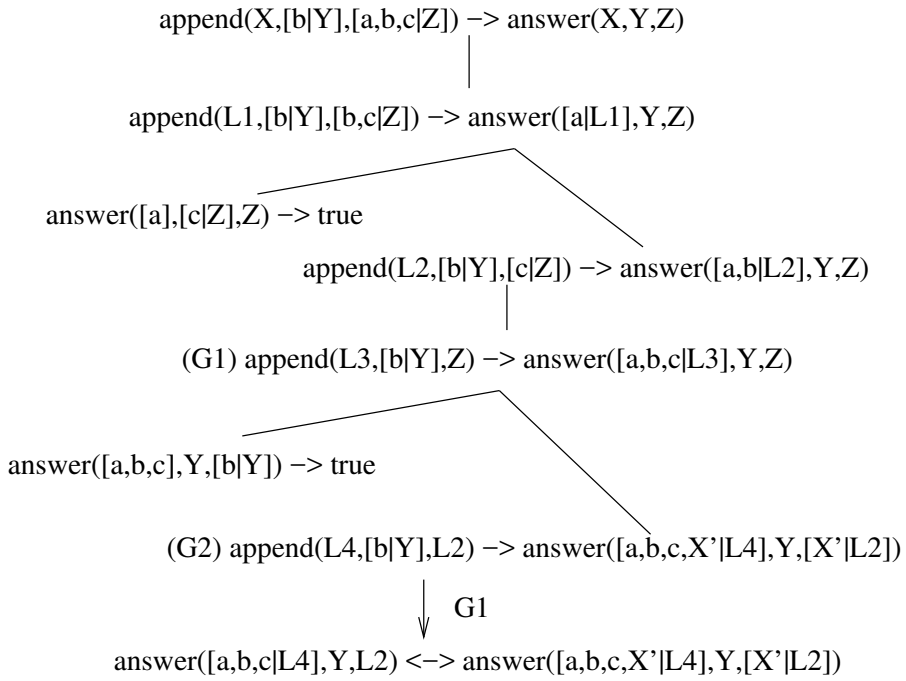
$\{X \leftarrow [a],\ Y \leftarrow [c|Z]\}$
$\{X \leftarrow [a, b, c],\ Z \leftarrow [b|Y]\}$
$\{X \leftarrow [a, b, c, X'],\ Z \leftarrow [X', b|Y]\}$
$\{X \leftarrow [a, b, c, X', X''],\ Z \leftarrow [X', X'', b|Y]\}$
. . . . . .
$\{X \leftarrow [a, b, c, X', X'', \ldots, X^n],\ Z \leftarrow [X', X'',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

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.



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

Figure 2: Linear completion generates only two answers

The last step, labeled by $\downarrow$, is a simplification step where an ancestor goal, labeled (G1) in Fig. 2, rewrites the current goal (G2) to a trivial goal in the form $answer(\ ) \simeq answer(\ )$. Since no inference can be applied to this goal, the execution halts with just two answers.

The reason for this different behaviour is that the Prolog program and the rewrite program give two different definitions of *append*. The "program units" in a rewrite program and in a Prolog program are interpreted in two different ways. In Prolog, each unit is a clause, where "$:-$" indicates the logical *if*. In rewrite programs, the logical connective "$\rightarrow$" 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 of 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 using 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 simplification is applied, i.e. if *append* is defined by equivalences. All those answers are equivalent to the second one with respect to the rewrite program so that they are 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 system is closer to the intended meaning of the definition 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 clause:

$P(X,Y,Z) :- append(X, [b|Y], [a,b,c|Z]), non-member(a,X).$
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 evaluates 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:

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

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 $Q(X,Y,Z) \rightarrow answer(X,Y,Z)$ is given as the query, the execution generates first the two solutions to the *append* subgoal

$size([a]) > 3 \rightarrow answer([a], [c|Z], Z)$

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

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

$append(L1, [b|Y], L2), size([a,b,c,X'|L1]) > 3 \rightarrow answer([a,b,c,X'|L1], Y, [X'|L2])$.

Assuming that *size* is defined as desired, $size([a,b,c,X'|L1]) > 3$ simplifies to *true* and the goal is reduced to

$append(L1, [b|Y], L2) \rightarrow answer([a,b,c,X'|L1], Y, [X'|L2])$ (G2).

Note that this is the same goal (G2) generated in the previous execution for the *append* query. In that execution this goal is rewritten by its ancestor (G1) and it does not yield any answer. In the execution for the Q query all the ancestors contain a *size* literal too, 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

$append(L1, [b|Y], L3) \rightarrow answer([a,b,c,X',X''|L1], Y, [X',X''|L3])$

is reduced by its ancestor (G2) to

$answer([a,b,c,X'|L1], Y, [X'|L3]) \simeq answer([a,b,c,X',X''|L1], Y, [X', X''|L3])$.

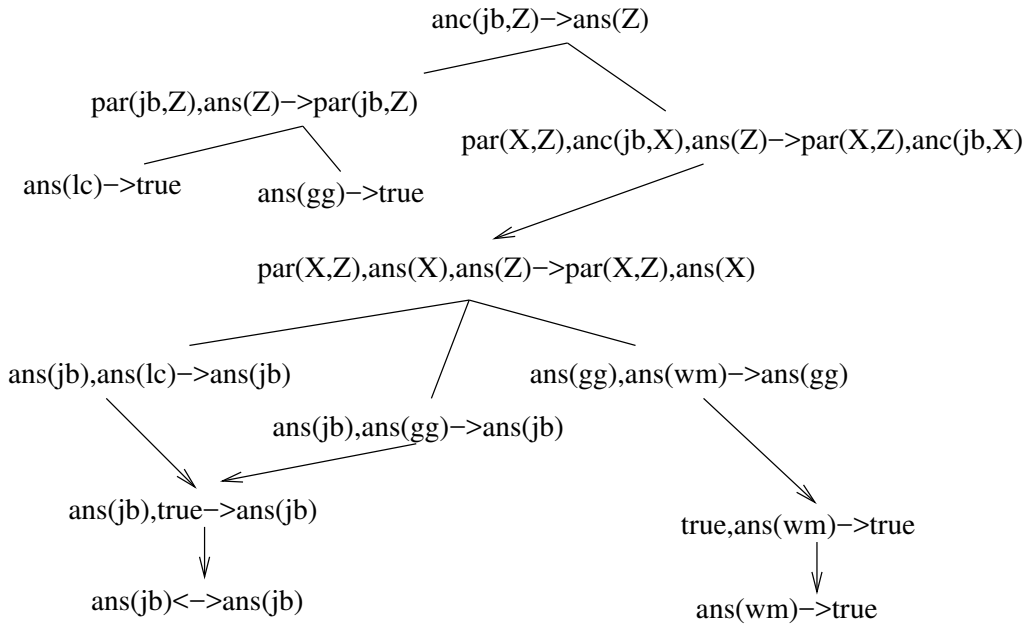
So far we have seen examples where it is desirable to define predicates by equivalences. However, not all relations are meant to be defined by equivalences. Still, rewrite programs allow us to define predicates by implications. For example, for the *ancestor* relation

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

both clauses are implications. As it was already pointed out in [9], implications can be written as bi-implications by recalling that $P :- Q$ is equivalent to $P \wedge Q \rightarrow Q$. If we add some facts and a query we get:

$parent(jb, lc) \rightarrow true$
$parent(jb, gg) \rightarrow true$
$parent(gg, wm) \rightarrow true$
$ancestor(X,Y),\ parent(X,Y) \rightarrow parent(X,Y)$
$ancestor(X,Y),\ parent(Z,Y),\ ancestor(X,Z) \rightarrow parent(Z,Y), ancestor(X,Z)$
$ancestor(jb, Z) \rightarrow answer(Z).$

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



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

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 $parent(X, Z), ancestor(jb, X)$, 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 the search space.

2 Linear completion

As we have seen in the previous section, rewrite programs allow us to distinguish between mutually exclusive definitions and non mutually exclusive definitions. A predicate is *mutually exclusively defined* if it is defined by a set of clauses such that no two heads unify[1]. If a predicate A is defined by a set of clauses

[1] A more general notion of mutually exclusive clauses appears in [6].

$A(\bar{t}_1).$
$\ldots$
$A(\bar{t}_m).$
$A(\bar{t}_{m+1}) :- B_{11} \ldots B_{1p_1}.$
$\ldots$
$A(\bar{t}_{m+n}) :- B_{n1} \ldots B_{np_n}.$

its rewrite program contains the rules

$A(\bar{t}_1) \to true$
$\ldots$
$A(\bar{t}_m) \to true$
$A(\bar{t}_{m+1}) \to B_{11} \ldots B_{1p_1}$
$\ldots$
$A(\bar{t}_{m+n}) \to B_{n1} \ldots B_{np_n}$

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 [2]. Otherwise A is transformed into

$A(\bar{t}_1) \to true$
$\ldots$
$A(\bar{t}_m) \to true$
$A(\bar{t}_{m+1})B_{11} \ldots B_{1p_1} \to B_{11} \ldots B_{1p_1}$
$\ldots$
$A(\bar{t}_{m+n})B_{n1} \ldots B_{np_n} \to B_{n1} \ldots B_{np_n}$

We call the rewrite rules representing facts, implications and bi-implications *fact rules*, *if-rules* and *iff-rules*. 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*. 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.

Rewrite programs are interpreted by *linear completion*.

A query $\exists \bar{x} Q_1 \ldots Q_m$ is negated into $Q_1 \ldots Q_m \to false$ and written as a *query rule* $Q_1 \ldots Q_m \to answer(\bar{x})$, where $\bar{x}$ contains all the free variables in $Q_1 \ldots Q_m$. When a rule in the form $answer(\bar{x})\sigma \to true$ is deduced, $\bar{x}\sigma$ is a solution to the query. The *answer* literal was used by Dershowitz in [9], who refers to [14] for its introduction. The *state* of a computation is defined by a triple

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

where E is the program, $L_1 \ldots L_l \to R_1 \ldots R_r$ is the current goal and S is the set of *goal simplifiers*. The set S contains all the ancestors of the current

[2] 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), $c[s] \succ s$ (subterm property) and $\succ$ is well founded [7].

goal plus all the previously generated *answer rules*, i.e. the goal rules in the form $answer(\bar{x})\sigma \to true$. 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

$$(E; Q_1 \dots Q_m \to answer(\bar{x}); \emptyset)$$

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

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

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

$E \cup \{Q_1 \dots Q_m \to answer(\bar{x})\} \vdash_{LC} answer(\bar{x})\sigma \to true$

We also denote it by $E \vdash_{LC} Q_1 \dots Q_m\sigma$, meaning that $Q_1 \dots Q_m\sigma$ is proved from program E by linear completion. If a ground query is given, the answer substitution is empty and we write $E \vdash_{LC} Q_1 \dots Q_m$.

A computation step transforms the current state in one of the following ways:

Simplify:

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

Overlap:

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

Answer:

$$\frac{(E; answer(\bar{x})\sigma \to true; S)}{(E; —; S \cup \{answer(\bar{x})\sigma \to true\})}$$

Delete:

$$\frac{(E; L_1 \dots L_l \simeq L_1 \dots L_l; S)}{(E; —; S)}$$

where — means "backtrack".

In *Simplify*, $L_1 \dots L_l \to R_1 \dots R_r$ is simplified into a new goal $L'_1 \dots L'_n \to R'_1 \dots R'_s$ using $E \cup S$. The rewritten goal is discarded. Simplification of goals by goals is the basic difference between our definition of linear completion and the one in [9]. 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 added to the simplifiers set. In *Answer* an answer is found and the answer rule is put into the simplifier set. Then the interpreter backtracks if possible. Similarly, in *Delete* a goal which is an identity is deleted and then the interpreter backtracks.

Note that 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.

The simplifiers set is a global variable: its modifications on one path of the computation tree affect the other paths since a current goal at some node can be simplified by an answer rule reached on another path. A linear completion interpreter builds the tree sequentially following some search strategy. For instance a depth first strategy with backtracking as in Prolog can be adopted. If the goal rule in the current state is such that none of the above inference rule applies, the interpreter backtracks. This is the case for instance when the goal rule has the form $answer(\bar{t}) \simeq answer(\bar{s})$. Here we focus on inference rules only, abstracting from the search strategy. In the following we assume that a fair strategy is adopted whenever needed.

The overlap steps in LC are similar to the resolution steps in Prolog. Given a goal rule $A(\bar{u})L_1 \dots L_l \rightarrow R_1 \dots R_r$ 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:

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

Overlap with an iff-rule:

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

Overlap with a fact rule:

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

where σ is the most general unifier of $A(\bar{s})$ and $A(\bar{u})$. In the last case the generated goal is oriented according to a simplification ordering. Here and in the following we assume that the selected literal in a goal $Q_1 \dots Q_m$ is Q_1. There is no loss of generality, because if the selected literal is Q_j with $j \neq 1$, we can permute the indices so that $j = 1$. The generated goals are oriented by a simplification ordering such that goal rules in the form $B_1 \dots B_n answer(\bar{t}) \rightarrow B_1 \dots B_n$ and $B_1 \dots B_n \rightarrow answer(\bar{t})$ are oriented from left to right. 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.

Note that no overlap on an atom different from the head of a rule needs to be considered. If the atom $A(\bar{u})$ in the goal rule $A(\bar{u})\bar{L} \rightarrow \bar{R}$ unifies with an atom $A(\bar{v})$ in an if-rule $CA(\bar{v})\bar{B} \rightarrow A(\bar{v})\bar{B}$ the overlap step generates the goal

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

which is reduced by its predecessor 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 will lead to the identity $A(\bar{v})\bar{B}\bar{R}\sigma \simeq A(\bar{v})\bar{B}\bar{R}\sigma$.

The key feature of rewrite programs is simplification of the current goal by program rules, ancestor goal rules and answer rules. If an if-rule $AB_1 \dots B_n \to B_1 \dots B_n$ or a fact $A \to true$ simplifies a query $Q_1 \dots Q_m$, then the atom Q_j such that $A\sigma = Q_j$ is deleted. If an iff-rule $A \to B_1 \dots B_n$ applies, the atom Q_j is replaced by $(B_1 \dots B_n)\sigma$. The rewrite rules $x \cdot true \to x$ and $x \cdot x \to x$ are also implicitly applied. They allow us to delete any repeated atom and any occurrence of *true* in a conjunction. Notice that since the product is associative, commutative and idempotent, we regard conjunctions of atoms as *sets* of atoms: the left side $\bar{L}$ of an if-rule or of an ancestor goal rule matches a side $\bar{R}$ of the current goal rule if there is a subset of $\bar{R}$ which is an instance of $\bar{L}$. Simplification is given higher priority than overlap. The current goal rule is always fully simplified before the next overlap step is performed.

3 A fixpoint characterization of rewrite programs

Rewrite programs have operational, model theoretic and fixpoint semantics like Prolog programs. Let E be a rewrite program, $\mathcal{B}$ be its Herbrand base and $\mathcal{P}(\mathcal{B})$ be 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 $E^* = E \cup \{x \cdot x \to x,\ x \cdot true \to x\}$. If $E \vdash_{LC} Q_1 \dots Q_m\sigma$, we say that σ is a *correct answer substitution* for the query $Q_1 \dots Q_m$ if $E^* \models Q_1 \dots Q_m\sigma = true$.

We define a *least fixpoint semantics* of rewrite programs on the lattice $\mathbb{B} = \{I' \mid I' = I \cup \{true\}, I \subseteq \mathcal{B}\}$. $\mathbb{B}$ is $\mathcal{P}(\mathcal{B})$ where the element *true* is added to each subset of $\mathcal{B}$. 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 3.1 *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 $A_1 \dots A_n \simeq B_1 \dots B_m$ $(n \geq 1, m \geq 1)$ such that $P = A_i\sigma$ and $\{A_1\sigma, \dots, A_{i-1}\sigma, A_{i+1}\sigma, \dots, A_n\sigma, B_1\sigma, \dots, B_m\sigma\} \subseteq I$ for some i, $1 \leq i \leq n$, and some ground substitution σ. (The double arrow $\simeq$ means there is no distinction between the left hand side and the right hand side.)*

Lemma 3.1 *Given a rewrite program E, T_E is continuous, that is for every non decreasing chain $X_1 \subseteq X_2 \subseteq \ldots$ of elements in $\mathbb{B}$, $T_E(\bigcup\{X_i \mid i < \omega\}) = \bigcup\{T_E(X_i) \mid i < \omega\}$.*

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:

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

4 Equivalence of proof theoretic, model theoretic and fixpoint semantics

We show now that $lfp(T_E)$ is a fixpoint characterization of both the operational and model theoretic semantics of a program E. These results are obtained through a few steps. The proofs are omitted due to their length and can be found in the full version of the paper [4].

Lemma 4.1 *For all conjunctions of atoms $Q_1 \ldots Q_m$, $Q_1 \ldots Q_m \leftrightarrow_{E^*}^* true$ if and only if $\forall j, 1 \leq j \leq m$, $Q_j \leftrightarrow_{E^*}^* true$.*

The following theorem establishes the soundness of linear completion, i.e. that all the answers given by linear completion are correct answer substitutions:

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

We now restrict our attention to ground queries. First we prove a lemma which allows us to split the problem of proving a ground conjunction into the subproblems of proving its single atoms:

Lemma 4.2 *$\forall Q_1 \ldots Q_m \in \mathcal{B}$, $E \vdash_{LC} Q_1 \ldots Q_m$ if and only if $\forall j$, $1 \leq j \leq m$, $E \vdash_{LC} Q_j$.*

Intuitively this result holds because since the query is ground, there is no computation of an answer substitution and the processes of reducing to *true* the literals in the query are independent processes. Next we show that all ground queries proved by linear completion from E are equivalent to *true*:

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

These results allow us to relate the fixpoint semantics of a rewrite program E to its operational semantics:

Theorem 4.3 *$\forall Q_1 \dots Q_m \in \mathcal{B}$, $Q_1 \dots Q_m \leftrightarrow^*_{E^*} true$ if and only if $\forall i$, $1 \leq i \leq m$, $Q_i \in lfp(T_E)$.*

Therefore, the fixpoint semantics $lfp(T_E)$ captures both the operational and model theoretic semantics of the rewrite program E:

Theorem 4.4 *$\forall G \in \mathcal{B}$, $G \in lfp(T_E)$ if and only if $E \vdash_{LC} G$.*

Theorem 4.5 *$\forall G \in \mathcal{B}$, $G \in lfp(T_E)$ if and only if $E^* \models G = true$.*

5 Denotational semantics of rewrite programs

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 $A' :- B_1 \dots B_m$ $(m \geq 0)$ such that $A = A'\sigma$ and $\{B_1\sigma \dots B_m\sigma\} \subseteq I$ for some ground substitution σ [16, 1]. The following theorem shows that the two semantics are indeed the same.

Theorem 5.1 *If $E \equiv P$, then $lfp(T_E) = lfp(T_P)$.*

This theorem shows 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. However, the behaviour of the two programs P and E may be different. We show in the following that even if a rewrite programs E may generates less answers than the corresponding Prolog program P, for all answers given by P there is an answer given by E which is E-equivalent.

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 5.2 *If $E \equiv P$, if $E \vdash_{LC} Q_1 \dots Q_m\sigma$, then there exists an answer θ, $P \vdash_{Prolog} Q_1 \dots Q_m\theta$, such that $\sigma = \theta\rho$ for some substitution ρ.*

We now prove that linear completion is complete as well 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 $Q_1 \dots Q_m \to answer(\bar{x})$ can be written as a single literal query by introducing a new predicate symbol N, a new program rule $N \to Q_1 \dots Q_m$ or $NQ_1 \dots Q_m \to Q_1 \dots Q_m$ and the query $N \to answer(\bar{x})$.

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 5.3 *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.*

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

Lemma 5.1 *If linear completion generates a computation path $(E; G \to answer(\bar{x}); \emptyset) \vdash^*_{LC} (E; \bar{W} \to \bar{V}; \_) \vdash^*_{LC} (E; H; \_) \vdash_{LC} (E; \bar{R} \simeq \bar{R}; \_)$, where the goal H is simplified to an identity by its predecessor $\bar{W} \to \bar{V}$, 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.)*

Lemma 5.2 *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. $E \vdash_{LC} G\sigma$.*

We can finally state our completeness result:

Theorem 5.4 *Let E be an iff-program. 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.*

Theorem 5.5 *If $E \equiv P$, if $P \vdash_{Prolog} G\theta$, then there exists an answer σ given by linear completion, $E \vdash_{LC} G\sigma$ and a substitution ρ, such that $G\theta \leftrightarrow^*_{E^*} G\sigma\rho$.*

6 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 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 and linear completion have already been discussed in [10, 9, 11, 18]. 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 [18], although they did not study the case where a predicate is defined as an implication and not as a logical equivalence. Neither did they explain the semantics of their method. Techniques for loop detection in the execution of Prolog programs have received considerable attention. In [2, 3] loop checking mechanisms based on subsumption are studied. The basic idea in these loop checks is to eliminate the current goal if it is subsumed by one of its ancestors. The pruning effect of this kind of loop checks can turn out to be similar to the effect of our simplification of goals by ancestors. However, simplification as it is done in linear completion is more powerful in general, since it also includes simplification by program rules and by answer rules. More importantly, we believe that our approach is more natural and easier to understand, since simplification is a natural consequence of writing program units as equations.

Rewrite programs have the curious property of being denotationally equivalent to Prolog on the ground level while yield less 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 both in the examples and in the theorems. 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 $A \rightarrow false$.

Acknowledgements

This research was supported in part by grants CCR-8805734, INT-8715231 and CCR-8901322, funded by the National Science Foundation. The first author is also supported by Dottorato di ricerca in Informatica, Universitá degli Studi di Milano, Italy.

References

[1] K.R.Apt and M.H.Van Emden. Contributions to the Theory of Logic Programming. *J. ACM*, 29,3:841–862, July 1982.

[2] K.R.Apt, R.N.Bol and J.W.Klop. On the Safe Termination of PROLOG programs. *Proc. of the Sixth Int. Conf. on Logic Programming*, G.Levi and M.Martelli eds., 353–368, MIT Press, Cambridge MA, 1989.

[3] R.N.Bol, K.R.Apt and J.W.Klop. On the Power of Subsumption and Context Checks. *Proc. Int. Symp. on Design and Implementation of Systems for Symbolic Computation*, A.Miola ed., 131–140, Capri, Italy, April 1990.

[4] M.P.Bonacina, J.Hsiang. Operational and Denotational Semantics of Rewrite Programs. Technical report, Dept. of Computer Science, SUNY, Stony Brook, NY, March 1990.

[5] R.M.Burstall, D.B.MacQueen and D.T.Sannella. HOPE: An experimental applicative language. *Conf. Record of the 1980 LISP Conf.*, 136–143, Stanford, CA, 1980.

[6] S.K.Debray and D.S.Warren. Functional Computations in Logic Programs. *ACM Trans. on Programming Languages and Systems*, 11,3:451–481, July 1989.

[7] N.Dershowitz. Orderings for term rewriting systems. *J. of Theoret. Comp. Sci.*, 17,3:279–301, 1982.

[8] N.Dershowitz, J.Hsiang, N.A.Josephson and D.A.Plaisted. Associative-commutative rewriting. *Proc. of the Eighth Int. Joint Conf. on Artificial Intelligence*, A.Bundy ed., 940–944, Karlsrue, Germany, 1983.

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

[10] N.Dershowitz and N.A.Josephson. Logic Programming by Completion. *Proc. of the Second Int. Conf. on Logic Programming*, 313–320, Uppsala, Sweden, 1984.

[11] N.Dershowitz and D.A.Plaisted. Logic Programming Cum Applicative Programming. *Proc. IEEE Symp. on Logic Programming*, 54–66, Boston MA, 1985.

[12] K.Futatsugi, J.A.Goguen, J.P.Jouannaud and J.Meseguer. Principles of OBJ2. *Conf. Record of the 12th Annual ACM Symp. on Principles of Programming Languages*, New Orleans, LA, 1985.

[13] J.A.Goguen and J.Meseguer. Equality, types, modules and (why not?) generics for logic programming. *J. Logic Programming*, 1,2:179–210, 1984.

[14] C.C.Green. The Application of Theorem-proving to Question-answering. Ph.D. Thesis, Dept. of Computer Science, Stanford University, Stanford, CA, 1969.

[15] C.M.Hoffmann and M.J.O'Donnell. Programming with Equations. *ACM Trans. on Programming Languages and Systems*, 4,1:83–112, Jan. 1982.

[16] R.A.Kowalski and M.H.Van Emden. Predicate Logic as a Programming Language. *J. ACM*, 4,23, 1976.

[17] J.W.Lloyd. Foundations of Logic Programming. Second edition, Springer Verlag, Berlin, 1987.

[18] P.Réty, C.Kirchner, H.Kirchner and P.Lescanne. NARROWER: A new Algorithm for Unification and its Application to Logic Programming *Proc. of the First Int. Conf. on Rewrite Techniques and Applications*, J.P.Jouannaud ed., Dijon, France, May 1985.