# Reversible Debugging in Logic Programming[*]

Germán Vidal[0000−0002−1857−6951]

MiST, VRAIN, Universitat Politècnica de València, Spain
`gvidal@dsic.upv.es`

**Abstract.** Reversible debugging is becoming increasingly popular for locating the source of errors. This technique proposes a more natural approach to debugging, where one can explore a computation from the observable misbehaviour *backwards* to the source of the error. In this work, we propose a reversible debugging scheme for logic programs. For this purpose, we define an appropriate instrumented semantics (a so-called Landauer embedding) that makes SLD resolution reversible. An implementation of a reversible debugger for Prolog, rever, has been developed and is publicly available.

*This paper is dedicated to Manuel Hermenegildo on his 60th birthday, for his many contributions to logic programming as well as his energetic leadership within the community. I wish him many springs more to come.*

## 1 Introduction

Reversible debugging allows one to explore a program execution back and forth. In particular, if one observes a misbehaviour in some execution (e.g., a variable that takes a wrong value or an unexpected exception), reversible debugging allows us to analyse the execution backwards from this point. This feature is particularly useful for long executions, where a step-by-step forward inspection from the beginning of the execution would take too much time, or be even impractical.

One can already find a number of tools for reversible debugging in different programming languages, like Undo [12], rr [9] or CauDEr [6], to name a few.

---

In this work, we consider reversible debugging in *logic programming* [7]. In this context, one has to deal with two specific features that are not common in other programming languages: nondetermism and a bidirectional parameter passing mechanism (unification).

Typically, the *reversibilization* of a (reduction) semantics can be obtained by instrumenting the states with an appropriate *Landauer embedding* [5], i.e., by introducing a *history* where the information required to undo the computation steps is stored. Defining a Landauer embedding for logic programming is a challenging task because of nondetermism and unification. On the one hand, in order to undo backtracking steps, a deterministic semantics that models the complete traversal of an SLD tree is required (like the linear operational semantics introduced in [10]). On the other hand, unification is an irreversible operation: given two terms, $s$ and $t$, with most general unifier $\sigma$, we cannot obtain $s$ from $t$ and $\sigma$ (nor $t$ from $s$ and $\sigma$).

Let us note that, in this work, we aim at reversibility in the sense of being able to *deterministically* undo the steps of a computation. In general, (pure) logic programs are *invertible* (e.g., the same relation can be used for both addition and subtraction), but they are not reversible in the above sense.

This paper extends the preliminary results reported in the short paper [13]. In particular, our main contributions are the following:

- First, we define a reversible operational semantics for logic programs that deals explicitly with backtracking steps. In particular, we define both a *forward* and a *backward* transition relation that model forward and backward computations, respectively.
- Moreover, we state and prove some formal properties for our reversible semantics, including the fact that it is indeed a conservative extension of the standard semantics, that it is deterministic, and that any forward computation can be undone.
- Finally, we present the design of a reversible debugger for Prolog that is based on our reversible semantics, and discuss some aspects of the implemented tool, the reversible debugger rever.

We consider that our work can be useful in the context of existing techniques for program validation in logic programming, like run-time verification (e.g., [11]) or concolic testing (e.g., [8]), in order to help locating the bugs of a program.

The paper is organised as follows. After introducing some preliminaries in the next section, we introduce our reversible operational semantics in Section 3. Then, Section 4 presents the design of a reversible debugger based on the previous semantics. Finally, Section 5 compares our approach with some related work and Section 6 concludes and points out some directions for further research.

## 2    Preliminaries

In this section, we briefly recall some basic notions from logic programming (see, e.g., [7,1] for more details).

In this work, we consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by $\Pi$, $\Sigma$ and $\mathcal{V}$, respectively, with $\Sigma \cap \Pi = \emptyset$ and $(\Sigma \cup \Pi) \cap \mathcal{V} = \emptyset$. Every element of $\Sigma \cup \Pi$ has an *arity* which is the number of its arguments. We write $f/n \in \Sigma$ (resp. $p/n \in \Pi$) to denote that $f$ (resp. $p$) is an element of $\Sigma$ (resp. $\Pi$) whose arity is $n \geq 0$. A *constant symbol* is an element of $\Sigma$ whose arity is 0. We let $\mathcal{T}(\Sigma, \mathcal{V})$ denote the set of *terms* constructed using symbols from $\Sigma$ and variables from $\mathcal{V}$.

An *atom* has the form $p(t_1, \ldots, t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ for $i = 1, \ldots, n$. A *query* is a finite conjunction of atoms which is denoted by a sequence of the form $A_1, \ldots, A_n$, where the *empty query* is denoted by `true`. A *clause* has the form $H \leftarrow B_1, \ldots, B_n$, where $H$ (the *head*) and $B_1, \ldots, B_n$ (the *body*) are atoms, $n \geq 0$ (thus we only consider *definite* logic programs, i.e., logic programs without negated atoms in the body of the clauses). Clauses with an empty body, $H \leftarrow$ `true`, are called *facts*, and are typically denoted by $H$. In the following, atoms are ranged over by $A, B, C, H, \ldots$ while queries (possibly empty sequences of atoms) are ranged over by $\mathcal{A}, \mathcal{B}, \ldots$

$\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object $s$ (i.e., $s$ can be a term, an atom, a query, or a clause). A syntactic object $s$ is *ground* if $\mathcal{V}ar(s) = \emptyset$. In this work, we only consider *finite* ground terms.

Substitutions and their operations are defined as usual; they are typically denoted by (finite) sets of bindings like, e.g., $\{x_1/s_1, \ldots, x_n/s_n\}$. We let *id* denote the identity substitution. Substitutions are ranged over by $\sigma, \theta, \ldots$ In particular, the set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of a substitution $\sigma$. Composition of substitutions is denoted by juxtaposition, i.e., $\sigma\theta$ denotes a substitution $\gamma$ such that $\gamma(x) = \theta(\sigma(x))$ for all $x \in \mathcal{V}$. We follow a postfix notation for substitution application: given a syntactic object $s$ and a substitution $\sigma$ the application $\sigma(s)$ is denoted by $s\sigma$. The *restriction* $\theta\restriction_V$ of a substitution $\theta$ to a set of variables $V$ is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$ and $x\theta\restriction_V = x$ otherwise. We say that $\theta = \sigma\ [V]$ if $\theta\restriction_V = \sigma\restriction_V$.

A syntactic object $s_1$ is *more general* than a syntactic object $s_2$, denoted $s_1 \leqslant s_2$, if there exists a substitution $\theta$ such that $s_2 = s_1\theta$. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. Two syntactic objects $t_1$ and $t_2$ are *variants* (or equal up to variable renaming), denoted $t_1 \approx t_2$, if $t_1 = t_2\rho$ for some variable renaming $\rho$. A substitution $\theta$ is a unifier of two syntactic objects $t_1$ and $t_2$ iff $t_1\theta = t_2\theta$; furthermore, $\theta$ is the *most general unifier* of $t_1$ and $t_2$, denoted by $\mathsf{mgu}(t_1, t_2)$ if, for every other unifier $\sigma$ of $t_1$ and $t_2$, we have that $\theta \leqslant \sigma$.

A logic *program* is a finite sequence of clauses. Given a program $P$, we say that $A, \mathcal{B}' \leadsto_{P,\sigma} (\mathcal{B}, \mathcal{B}')\sigma$ is an *SLD resolution step*[1] if $H \leftarrow \mathcal{B}$ is a renamed apart clause (i.e., with fresh variables) of program $P$, in symbols, $H \leftarrow \mathcal{B} \ll P$, and $\sigma = \mathsf{mgu}(A, H)$. The subscript $P$ will often be omitted when the program is clear from the context. An *SLD derivation* is a (finite or infinite) sequence of SLD resolution steps. As is common, $\leadsto^*$ denotes the reflexive and transitive

---

[1] In this paper, we only consider Prolog's *computation rule*, so that the selected atom in a query is always the leftmost one.

closure of $\rightsquigarrow$. In particular, we denote by $\mathcal{A}_0 \rightsquigarrow^*_\theta \mathcal{A}_n$ a derivation

$$\mathcal{A}_0 \rightsquigarrow_{\theta_1} \mathcal{A}_1 \rightsquigarrow_{\theta_2} \ldots \rightsquigarrow_{\theta_n} \mathcal{A}_n$$

where $\theta = \theta_1 \ldots \theta_n$ if $n > 0$ (and $\theta = id$ otherwise).

An SLD derivation is called *successful* if it ends with the query true, and it is called *failed* if it ends in a query where the leftmost atom does not unify with the head of any clause. Given a successful SLD derivation $\mathcal{A} \rightsquigarrow^*_\theta$ true, the associated *computed answer*, $\theta\big|_{\mathcal{V}ar(\mathcal{A})}$, is the restriction of $\theta$ to the variables of the initial query $\mathcal{A}$. SLD derivations are represented by a (possibly infinite) finitely branching tree, which is called *SLD tree*. Here, *choice points* (queries with more than one child) correspond to queries where the leftmost atom unifies with the head of more than one program clause.

*Example 1.* Consider the following (labelled) logic program:[2]

$\ell_1 : \mathtt{p(X,Y)\ :-\ q(X), r(X,Y)}.$
$\ell_2 : \mathtt{q(a)}.$　　　　　　　　　$\ell_5 : \mathtt{r(b,b)}.$
$\ell_3 : \mathtt{q(b)}.$　　　　　　　　　$\ell_6 : \mathtt{r(b,c)}.$
$\ell_4 : \mathtt{q(c)}.$　　　　　　　　　$\ell_7 : \mathtt{r(c,c)}.$

Given the query $\mathtt{p(X,Y)}$, we have, e.g., the following (successful) SLD derivation:

$$\mathtt{p(A,B)} \rightsquigarrow_{\{X/A,Y/B\}} \mathtt{q(A), r(A,B)}$$
$$\rightsquigarrow_{\{A/b\}} \quad \mathtt{r(b,B)}$$
$$\rightsquigarrow_{\{B/c\}} \quad \mathtt{true}$$

with computer answer $\{\mathtt{A/b, B/c}\}$.

## 3   A Reversible Semantics for Logic Programs

In this section, we present a reversible semantics for logic programs that constitutes a good basis to implement a reversible debugger for Prolog (cf. Section 4). In principle, one of the main challenges for defining a reversible version of SLD resolution is dealing with unification, since it is an irreversible operation. E.g., given the SLD resolution step

$$\mathtt{p(X,a), q(a)} \rightsquigarrow_{\{X/a,Y/a\}} \mathtt{q(a), q(a)}$$

using clause $\mathtt{p(a,Y)\ :-\ q(Y)}$, there is no deterministic way to get back the query $\mathtt{p(X,a), q(a)}$ from the query $\mathtt{q(a), q(a)}$, the computed mgu $\{\mathtt{X/a, Y/a}\}$, and the applied clause. For instance, one could obtain the query $\mathtt{p(X,X), q(X)}$ since the following SLD resolution step

$$\mathtt{p(X,X), q(X)} \rightsquigarrow_{\{X/a,Y/a\}} \mathtt{q(a), q(a)}$$

is also possible using the same clause and computing the same mgu.

In order to overcome this problem, [13] proposed a reversible semantics where

---

[2] We consider Prolog notation in examples (so variables start with an uppercase letter). Clauses are labelled with a unique identifier of the form $\ell_i$.

– computed mgu's are not applied to the atoms of the query, and
– the selected call at each SLD resolution step is also stored.

Queries are represented as pairs $\langle \mathcal{A}; [(A_n, H_n, m_n), \ldots, (A_1, H_1, m_1)] \rangle$, where the first component is a sequence of atoms (a query), and the second component stores, for each SLD resolution step performed so far, the selected atom $(A_i)$, the head of the selected clause $(H_i)$, and the number of atoms in the body of this clause $(m_i)$. Here, mgu's are not stored explicitly but can be inferred from the pairs $(A_i, H_i)$. The number $m_i$ is used to determine the number of atoms in the current query that must be removed when performing a backward step. A reversible SLD resolution step has then the form[3]

$$\langle A, \mathcal{B}; \mathcal{H} \rangle \rightharpoonup \langle B_1, \ldots, B_m, \mathcal{B}; (A, H, m) : \mathcal{H} \rangle$$

if there exists a clause $H \leftarrow B_1, \ldots, B_m \ll P$ and $\mathsf{mgu}(A\sigma, H) \neq \mathsf{fail}$, where $\sigma$ is the substitution obtained from $\mathcal{H}$ by computing the mgu's associated to each triple $(A_i, H_i, m)$ in $\mathcal{H}$ and, then, composing them. A simple proof-of-concept implementation that follows this scheme can be found at https://github.com/mistupv/rever/tree/rc2020.

The proposal in [13], however, suffers from several drawbacks:

– First, it is very inefficient, since one should compute the mgu's of each SLD resolution step once and again. This representation was chosen in [13] for clarity and, especially, because it allowed us to easily implement it without using a ground representation for queries and programs, so that there was no need to reimplement all basic operations (mgu, substitution application and composition, etc).
– The second drawback is that the above definition of reversible SLD resolution cannot be used to undo a *backtracking* step, since the structure of the SLD tree is not explicit in the considered semantics.

In the following, we introduce a reversible operational semantics for logic programs that overcomes the above shortcomings.

### 3.1   A Deterministic Operational Semantics

First, we present a deterministic semantics (inspired by the linear operational semantics of [10]) that deals explicitly with backtracking.

Our semantics is defined as a transition relation on states. In the following, queries are represented as pairs $\langle \mathcal{A}; \theta \rangle$ instead of $\mathcal{A}\theta$, where $\theta$ is the composition of the mgu's computed so far in the derivation. This is needed in order to avoid undoing the application of mgu's, which is an irreversible operation.

**Definition 1 (state).** *A* state *is denoted by a sequence $Q_1 | Q_2 | \ldots | Q_n$, where each $Q_i$ is a (possibly labelled) query of the form $\langle \mathcal{B}; \theta \rangle$. In some cases, a query $Q$ is labelled with a clause label, e.g., $Q^\ell$, which will be used to denote that the query $Q$ can be unfolded with the clause labelled with $\ell$ (see below).*

---

[3] Here, $(A, H, m) : \mathcal{H}$ denotes a list with head element $(A, H, m)$ and tail $\mathcal{H}$.

$$\text{(backtrack)}\ \frac{S \neq \epsilon}{\langle \mathsf{fail}, \mathcal{B}; \theta \rangle \,|\, S \to S} \qquad \text{(next)}\ \frac{S \neq \epsilon}{\langle \mathtt{true}; \theta \rangle \,|\, S \to S}$$

$$\text{(choice)}\ \frac{A \neq \mathsf{fail} \wedge A \neq \mathtt{true} \wedge \mathsf{clauses}(A\theta, P) = \{\ell_1, \ldots, \ell_m\} \wedge m > 0}{\langle A, \mathcal{B}; \theta \rangle \,|\, S \to \langle A, \mathcal{B}; \theta \rangle^{\ell_1} \,|\, \ldots \,|\, \langle A, \mathcal{B}; \theta \rangle^{\ell_m} \,|\, S}$$

$$\text{(choice\_fail)}\ \frac{A \neq \mathsf{fail} \wedge A \neq \mathtt{true} \wedge \mathsf{clauses}(A\theta, P) = \emptyset}{\langle A, \mathcal{B}; \theta \rangle \,|\, S \to \langle \mathsf{fail}, \mathcal{B}; \theta \rangle \,|\, S}$$

$$\text{(unfold)}\ \frac{\mathsf{cl}(\ell, P) = H \leftarrow B_1, \ldots, B_n \wedge \mathsf{mgu}(A\theta, H) = \sigma}{\langle A, \mathcal{B}; \theta \rangle^{\ell} \,|\, S \to \langle B_1, \ldots, B_n, \mathcal{B}; \theta\sigma \rangle \,|\, S}$$

**Fig. 1.** A deterministic operational semantics

A state will often be denoted by $\langle \mathcal{B}; \theta \rangle \,|\, S$ so that $\langle \mathcal{B}; \theta \rangle$ is the first query of the sequence and $S$ denotes a (possibly empty) sequence of queries. In the following, an empty sequence is denoted by $\epsilon$.

In this paper, we consider that program clauses are labelled, so that each label uniquely identifies a program clause. Here, we use the auxiliary function $\mathsf{clauses}(A, P)$ to obtain the labels of those clauses in program $P$ whose heads unify with atom $A$, i.e.,

$$\mathsf{clauses}(A, P) = \{\ell \mid \ell : H \leftarrow \mathcal{B} \ll P \wedge \mathsf{mgu}(A, H) \neq \mathsf{fail}\}$$

and $\mathsf{cl}(\ell, P)$ to get a renamed apart variant of the clause labelled with $\ell$, i.e., $\mathsf{cl}(\ell, P) = (H \leftarrow \mathcal{B})\vartheta$ if $\ell : H \leftarrow \mathcal{B} \in P$ and $\vartheta$ is a variable renaming with fresh variables.

The rules of the semantics can be found in Figure 1. An *initial state* has the form $\langle A, \mathcal{B}; id \rangle$, where $A$ is an atom, $\mathcal{B}$ is a (possibly empty) sequence of atoms, and $id$ is the identity substitution. Initially, one can either apply rule choice or choice_fail. Let us assume that $A$ unifies with the head of some clauses, say $\ell_1, \ldots, \ell_m$. Then, rule choice derives a new state by replacing $\langle A, \mathcal{B}; id \rangle$ with $m$ copies labelled with $\ell_1, \ldots, \ell_m$:

$$\langle A, \mathcal{B}; id \rangle \to \langle A, \mathcal{B}; id \rangle^{\ell_1} \,|\, \ldots \,|\, \langle A, \mathcal{B}; id \rangle^{\ell_m}$$

Now, let assume that $\mathsf{cl}(\ell_1, P)$ returns $H \leftarrow B_1, \ldots, B_n$. Then, rule unfold applies so that the following state is derived:

$$\langle B_1, \ldots, B_n, \mathcal{B}; \sigma \rangle \,|\, \langle A, \mathcal{B}; id \rangle^{\ell_2} \,|\, \ldots \,|\, \langle A, \mathcal{B}; id \rangle^{\ell_m}$$

Let us consider now that $B_1\sigma$ does not match any program clause, i.e., we have $\mathsf{clauses}(B_1\sigma, P) = \emptyset$. Then, rule choice_fail applies and the following state is derived:

$$\langle \mathsf{fail}, B_2, \ldots, B_n, \mathcal{B}; \sigma \rangle \,|\, \langle A, \mathcal{B}; id \rangle^{\ell_2} \,|\, \ldots \,|\, \langle A, \mathcal{B}; id \rangle^{\ell_m}$$

Then, rule backtrack applies so that we jump to a choice point with some pending alternative (if any). In this case, we derive the state

$$\langle A, \mathcal{B}; id \rangle^{\ell_2} \,|\, \ldots \,|\, \langle A, \mathcal{B}; id \rangle^{\ell_m}$$

so that unfolding with clause $\ell_2$ is tried now, and so forth.

Here, we say that a derivation is *successful* if the last state has the form $\langle \mathtt{true}; \theta \rangle \,|\, S$. We have also included a rule called next to be able to reach all solutions of an SLD tree (which has a similar effect as rule backtrack). Therefore, $\theta$ is not necessarily the first computed answer, but an arbitrary one (as long as it is reachable from the initial state after a finite number of steps).

A computation is *failed* if it ends with a state of the form $\langle \mathtt{fail}, \mathcal{B}; \theta \rangle$, so no rule is applicable (note that rule backtrack is not applicable when there is a single query in the state).

*Example 2.* Consider the program of Example 1 and the same initial query: $\langle \mathtt{p(X, Y)}; id \rangle$. In order to reach the same computed answer, $\{\mathtt{A/b, B/c}\}$, we now perform the following (deterministic) derivation:[4]

$$
\begin{aligned}
\langle \mathtt{p(A, B)}; id \rangle \rightarrow_{\mathsf{choice}} \quad & \langle \mathtt{p(A, B)}; id \rangle^{\ell_1} \\
\rightarrow_{\mathsf{unfold}} \quad & \langle \mathtt{q(A), r(A, B)}; id \rangle \\
\rightarrow_{\mathsf{choice}} \quad & \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_2} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_3} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{unfold}} \quad & \langle \mathtt{r(A, B)}; \{\mathtt{A/a}\} \rangle \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_3} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{choice\_fail}} \quad & \langle \mathtt{fail}; \{\mathtt{A/a}\} \rangle \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_3} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{backtrack}} \quad & \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_3} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{unfold}} \quad & \langle \mathtt{r(A, B)}; \{\mathtt{A/b}\} \rangle \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{choice}} \quad & \langle \mathtt{r(A, B)}; \{\mathtt{A/b}\} \rangle^{\ell_5} \,|\, \langle \mathtt{r(A, B)}; \{\mathtt{A/b}\} \rangle^{\ell_6} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{unfold}} \quad & \langle \mathtt{true}; \{\mathtt{A/b, B/b}\} \rangle \,|\, \langle \mathtt{r(A, B)}; \{\mathtt{A/b}\} \rangle^{\ell_6} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{next}} \quad & \langle \mathtt{r(A, B)}; \{\mathtt{A/b}\} \rangle^{\ell_6} \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4} \\
\rightarrow_{\mathsf{unfold}} \quad & \langle \mathtt{true}; \{\mathtt{A/b, B/c}\} \rangle \,|\, \langle \mathtt{q(A), r(A, B)}; id \rangle^{\ell_4}
\end{aligned}
$$

with computer answer $\{\mathtt{A/b, B/c}\}$.

Clearly, the semantics in Figure 1 is deterministic. In the following, we assume that a fixed program $P$ is considered for stating formal properties.

**Theorem 1.** *Let $S$ be a state. Then, at most one rule from the semantics in Figure 1 is applicable.*

*Proof.* The proof is straightforward since the conditions of the rules do not overlap:

- If the leftmost query is not headed by $\mathtt{true}$ nor $\mathtt{fail}$ and the query is not labelled, only rule choice and choice_fail are applicable, and the conditions trivially do not overlap.

---

[4] For clarity, we only show the bindings for the variables in the initial query. Moreover, the steps are labelled with the applied rule.

- If the leftmost query is labelled, only rule unfold is applicable.
- Finally, if the leftmost query is headed by fail (resp. true) then only rule backtrack (resp. next) is applicable.

Now, we prove that the deterministic operational semantics is sound in the sense that it explores the SLD tree of a query following Prolog's depth-first search strategy:

**Theorem 2.** *Let $\langle \mathcal{A}; id \rangle$ be an initial state. If $\langle \mathcal{A}; id \rangle \rightarrow^* \langle \mathtt{true}; \theta \rangle \mid S$, then $\mathcal{A} \leadsto_\theta^* \mathtt{true}$, up to variable renaming.*

*Proof.* Here, we prove a more general claim. Let us consider an arbitrary query, $\langle \mathcal{A}; \sigma \rangle$ with $\langle \mathcal{A}; \sigma \rangle \rightarrow^* Q_1 \mid \ldots \mid Q_m$, where $Q_i$ is either $\langle \mathcal{B}_i; \sigma\theta_i \rangle$ or $\langle \mathcal{B}_i; \sigma\theta_i \rangle^{\ell_i}$, $i = 1, \ldots, m$. Then, we have $\mathcal{A}\sigma \leadsto_{\theta_i}^* \mathcal{B}_i\sigma\theta_i$ for all $i = 1, \ldots, m$ such that $\mathcal{B}_i \neq (\mathsf{fail}, \mathcal{B}')$ for some $\mathcal{B}'$, up to variable renaming. We exclude the queries with fail since failures are not made explicit in the definition of SLD resolution (this is just a device of our deterministic semantics to point out that either a backtracking step should be performed next or the derivation is failed).

We prove the claim by induction on the number $n$ of steps in the former derivation: $\langle \mathcal{A}; \sigma \rangle \rightarrow^* Q_1 \mid \ldots \mid Q_m$. Since the base case ($n = 0$) is trivial, let us consider the inductive case ($n > 0$). Here, we assume a derivation of $n + 1$ steps from $\langle \mathcal{A}; \sigma \rangle$. By the induction hypothesis, we have $\mathcal{A}\sigma \leadsto_{\theta_i}^* \mathcal{B}_i\sigma\theta_i$ for all $i = 1, \ldots, m$ such that $\mathcal{B}_i \neq (\mathsf{fail}, \mathcal{B}')$ for some $\mathcal{B}'$. We now distinguish several possibilities depending on the applied rule to the state $Q_1 \mid \ldots \mid Q_m$:

- If the applied rule is backtrack or next, we have

  $$Q_1 \mid Q_2 \mid \ldots \mid Q_m \rightarrow Q_2 \mid \ldots \mid Q_m$$

  and the claim trivially holds by the induction hypothesis.
- If the applied rule is choice, we have

  $$Q_1 \mid \ldots \mid Q_m \rightarrow Q_1^{\ell_1} \mid \ldots \mid Q_1^{\ell_k} \mid Q_2 \mid \ldots \mid Q_m$$

  for some $k > 0$, and the claim also follows trivially from the induction hypothesis.
- If the applied rule is choice_fail, the claim follows immediately by the induction hypothesis since a query of the form $(\mathsf{fail}, \mathcal{B}')$ is not considered.
- Finally, let us consider that the applied rule is unfold. Let $Q_1 = \langle A, \mathcal{B}; \sigma\theta_1 \rangle^{\ell_1}$. Then, we have

  $$\langle A, \mathcal{B}; \sigma\theta_1 \rangle^{\ell_1} \mid Q_2 \mid \ldots \mid Q_m \rightarrow \langle \mathcal{B}', \mathcal{B}; \sigma\theta_1\theta' \rangle \mid Q_2 \mid \ldots \mid Q_m$$

  if $\mathsf{cl}(\ell_1, P) = H \leftarrow \mathcal{B}'$ and $\mathsf{mgu}(A\sigma\theta_1, H) = \theta'$. Then, we also have an SLD resolution step of the form $(A, \mathcal{B})\sigma\theta_1 \leadsto_{\theta'} (\mathcal{B}', \mathcal{B})\sigma\theta_1\theta'$ using the same clause[5] and computing the same mgu and, thus, the claim follows from the induction hypothesis.

Note that the deterministic semantics is sound but *incomplete* in general since it implements a depth-first search strategy.

---

[5] For simplicity, we assume that the same renamed clauses are considered in both derivations.

$$(\text{backtrack}) \; \frac{}{\langle \text{fail}, \mathcal{B}; \theta \rangle \,|\, \langle A, \mathcal{B}'; \theta' \rangle \,|\, S \bullet \Pi \rightharpoonup \langle A, \mathcal{B}'; \theta' \rangle \,|\, S \bullet \text{bck}(\mathcal{B}, \theta) : \Pi_{\text{redo}(A\theta')}}$$

$$(\text{next}) \; \frac{S \neq \epsilon}{\langle \text{true}; \theta \rangle \,|\, S \bullet \Pi_{\text{answer}(\theta)} \rightharpoonup S \bullet \text{next}(\theta) : \Pi}$$

$$(\text{choice}) \; \frac{A \neq \text{true} \wedge A \neq \text{fail} \wedge A \neq \text{ret}(A') \wedge \text{clauses}(A\theta, P) = \{\ell_1, \ldots, \ell_m\} \wedge m > 0}{\langle A, \mathcal{B}; \theta \rangle \,|\, S \bullet \Pi_{\text{call}(A\theta)} \rightharpoonup \langle A, \mathcal{B}; \theta \rangle^{\ell_1} \,|\, \ldots \,|\, \langle A, \mathcal{B}; \theta \rangle^{\ell_m} \,|\, S \bullet \text{ch}(m) : \Pi}$$

$$(\text{choice\_fail}) \; \frac{A \neq \text{true} \wedge A \neq \text{fail} \wedge A \neq \text{ret}(A') \wedge \text{clauses}(A\theta, P) = \emptyset}{\langle A, \mathcal{B}; \theta \rangle \,|\, S \bullet \Pi_{\text{call}(A\theta)} \rightharpoonup \langle \text{fail}, \mathcal{B}; \theta \rangle \,|\, S \bullet \text{fail}(A) : \Pi_{\text{fail}(A\theta)}}$$

$$(\text{unfold}) \; \frac{A \neq \text{ret}(A') \wedge \text{cl}(\ell, P) = H \leftarrow B_1, \ldots, B_n \wedge \text{mgu}(A\theta, H) = \sigma}{\langle A, \mathcal{B}; \theta \rangle^{\ell} \,|\, S \bullet \Pi \rightharpoonup \langle B_1, \ldots, B_n, \text{ret}(A), \mathcal{B}; \theta\sigma \rangle \,|\, S \bullet \text{unf}(A, \theta, \ell) : \Pi}$$

$$(\text{exit}) \; \frac{}{\langle \text{ret}(A), \mathcal{B}; \theta \rangle \,|\, S \bullet \Pi_{\text{exit}(A\theta)} \rightharpoonup \langle \mathcal{B}; \theta \rangle \,|\, S \bullet \text{exit}(A) : \Pi}$$

**Fig. 2.** Forward reversible semantics

### 3.2  A Reversible Semantics

Now, we extend the deterministic operational semantics of Figure 1 in order to make it reversible. Our reversible semantics is defined on *configurations*:

**Definition 2 (configuration).** *A* configuration *is defined as a pair $S \bullet \Pi$ where $S$ is a state (as defined in Definition 1) and $\Pi$ is a list representing the history of the configuration. Here, we consider the following history events:*

- $\text{ch}(n)$*: denotes a choice step with $n$ branches;*
- $\text{unf}(A, \theta, \ell)$*: represents an unfolding step where the selected atom is $A$, the answer computed so far is $\theta$, and the selected clause is labelled with $\ell$;*
- $\text{fail}(A)$*: is associated to rule* choice_fail *and denotes that the selected atom $A$ matches no rule;*
- $\text{exit}(A)$*: denotes that the execution of atom $A$ has been completed (see below);*
- $\text{bck}(\mathcal{B}, \theta)$*: represents a backtracking step, where $\langle \text{fail}, \mathcal{B}; \theta \rangle$ is the query that failed;*
- $\text{next}(\theta)$*: denotes an application of rule* next *after an answer $\theta$ is obtained.*

*We use Haskell's notation for lists and denote by $s : \Pi$ a history with first element $s$ and tail $\Pi$; an empty history is denoted by $[\,]$.*

The reversible (forward) semantics is shown in Figure 2.[6] The rules of the reversible semantics are basically self-explanatory. They are essentially the same as in the standard deterministic semantics of Figure 1 except for the following differences:

---

[6] The subscripts of some configurations: call, exit, fail, redo, and answer, can be ignored for now. They will become useful in the next section.

- First, configurations now keep a *history* with enough information for undoing the steps of a computation.
- And, secondly, unfolding an atom $A$ now adds a new call of the form $\mathsf{ret}(A)$ after the atoms of the body (if any) of the considered program clause. This is then used in rule $\mathsf{exit}$ in order to determine when the call has been completed successfully ($\mathsf{ret}(A)$ marks the exit of a program clause). This extension is not introduced for reversibility, but it is part of the design of our reversible debugger (see Section 4, where the reversible debugger $\mathsf{rever}$ is presented). Here, and in the following, we assume that programs contain no predicate named $\mathsf{ret}/1$.

We note that extending our developments to SLD resolution with an arbitrary computation rule (i.e., different from Prolog's rule, which always selects the leftmost atom) is not difficult. Basically, one would only need to extend the $\mathsf{unf}$ elements as follows: $\mathsf{unf}(A, \theta, i, \ell)$, where $i$ is the position of the selected atom in the current query.

*Example 3.* Consider again the program of Example 1 and the initial query: $\langle \mathsf{p}(\mathsf{X}, \mathsf{Y}); id \rangle \bullet [\,]$. In order to reach the first computed answer, $\{\mathsf{A}/\mathsf{b}, \mathsf{B}/\mathsf{b}\}$, we perform the derivation shown in Figure 3.

It is worthwhile to observe that the drawbacks of [13] mentioned before are now overcome by using substitutions with the answer computed so far, together with a deterministic semantics where backtracking is dealt with explicitly.

Trivially, the instrumented semantics of Figure 2 is a conservative extension of the deterministic semantics of Figure 1 since the rules impose no additional condition. The only difference is the addition of atoms $\mathsf{ret}(A)$ that mark the exit of a program clause. In the following, given two states, $S, S'$, we let $S \sim S'$ if they are equal after removing all atoms of the form $\mathsf{ret}(A)$.

**Theorem 3.** *Let $Q$ be an initial state. Then, $Q \rightarrow^* S$ iff $Q \bullet [\,] \rightharpoonup^* S' \bullet \Pi$ such that $S \sim S'$ for some history $\Pi$, up to variable renaming.*

Let us now consider backward steps. Here, our goal is to be able to explore a given derivation backwards. For this purpose, we introduce a backward operational semantics that is essentially obtained by switching the configurations in each rule of the forward semantics, and removing all unnecessary premises. The resulting backward semantics is shown in Figure 4. Let us just add that, in rule $\overline{\mathsf{unfold}}$, we use the auxiliary function $\mathsf{body}(\ell, P)$ to denote the body of clause labelled with $\ell$ in program $P$, and, thus, $|\mathsf{body}(\ell, P)|$ represents the number of atoms in the body of this clause.[7] This information was stored explicitly in our previous approach [13].

*Example 4.* If we consider the configurations of Figure 3 from bottom to top, they constitute a backward derivation using the rules of Figure 4.

The following result states the reversibility of our semantics:

---

[7] As is common, $|S|$ denotes the cardinality of the set or sequence $S$.

$\langle \mathtt{p(A,B)}; id \rangle \bullet [\,]$

$\longrightarrow_{\mathsf{choice}} \langle \mathtt{p(A,B)}; id \rangle^{\ell_1} \bullet [\mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{unfold}} \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle \bullet [\mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{choice}} \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_2} \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_3}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4} \bullet [\mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{unfold}} \langle \mathtt{ret(q(A))}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/a}\} \rangle \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_3}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{exit}} \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/a}\} \rangle \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_3}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{choice\_fail}} \langle \mathtt{fail}, \mathtt{ret(p(A,B))}; \{\mathtt{A/a}\} \rangle \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_3}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{backtrack}} \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_3} \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3),$
$\qquad \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{unfold}} \langle \mathtt{ret(q(A))}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{unf}(\mathtt{q(A)}, id, \ell_3), \mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}),$
$\qquad \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{exit}} \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_3), \mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}),$
$\qquad \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{choice}} \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle^{\ell_5} \,|\, \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle^{\ell_6}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{ch}(2), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_3), \mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}),$
$\qquad \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{unfold}} \langle \mathtt{ret(r(A,B))}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}, \mathtt{B/b}\} \rangle \,|\, \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle^{\ell_6}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{unf}(\mathtt{r(A,B)}, \{\mathtt{A/b}\}, \ell_5), \mathsf{ch}(2), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_3),$
$\qquad \mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3),$
$\qquad \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{exit}} \langle \mathtt{ret(p(A,B))}; \{\mathtt{A/b}, \mathtt{B/b}\} \rangle \,|\, \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle^{\ell_6}$
$\qquad |\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{exit}(\mathtt{r(A,B)}), \mathsf{unf}(\mathtt{r(A,B)}, \{\mathtt{A/b}\}, \ell_5), \mathsf{ch}(2), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_3),$
$\qquad \mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}), \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3),$
$\qquad \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

$\longrightarrow_{\mathsf{exit}} \langle \mathtt{true}; \{\mathtt{A/b}, \mathtt{B/b}\} \rangle \,|\, \langle \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; \{\mathtt{A/b}\} \rangle^{\ell_6} \,|\, \langle \mathtt{q(A)}, \mathtt{r(A,B)}, \mathtt{ret(p(A,B))}; id \rangle^{\ell_4}$
$\qquad \bullet [\mathsf{exit}(\mathtt{p(A,B)}), \mathsf{exit}(\mathtt{r(A,B)}), \mathsf{unf}(\mathtt{r(A,B)}, \{\mathtt{A/b}\}, \ell_5), \mathsf{ch}(2), \mathsf{exit}(\mathtt{q(A)}),$
$\qquad \mathsf{unf}(\mathtt{q(A)}, id, \ell_3), \mathsf{bck}(\mathtt{ret(p(A,B))}, \{\mathtt{A/a}\}), \mathsf{fail}(\mathtt{r(A,B)}), \mathsf{exit}(\mathtt{q(A)}),$
$\qquad \mathsf{unf}(\mathtt{q(A)}, id, \ell_2), \mathsf{ch}(3), \mathsf{unf}(\mathtt{p(A,B)}, id, \ell_1), \mathsf{ch}(1)]$

**Fig. 3.** Example derivation with the reversible (forward) semantics.

$(\overline{\mathsf{backtrack}})$ $\quad S \bullet \mathsf{bck}(\mathcal{B},\theta) \colon \varPi \leftharpoonup \langle \mathsf{fail}, \mathcal{B}; \theta \rangle \,|\, S \bullet \varPi$

$(\overline{\mathsf{next}})$ $\quad S \bullet \mathsf{next}(\theta) \colon \varPi \leftharpoonup \langle \mathtt{true}; \theta \rangle \,|\, S \bullet \varPi$

$(\underline{\mathsf{choice}})$ $\quad \langle A, \mathcal{B}; \theta \rangle^{\ell_1} \,|\, \ldots \,|\, \langle A, \mathcal{B}; \theta \rangle^{\ell_m} \,|\, S \bullet \mathsf{ch}(m) \colon \varPi \leftharpoonup \langle A, \mathcal{B}; \theta \rangle \,|\, S \bullet \varPi$

$(\underline{\mathsf{choice\_fail}})$ $\quad \langle \mathsf{fail}, \mathcal{B}; \theta \rangle \,|\, S \bullet \mathsf{fail}(A) \colon \varPi \leftharpoonup \langle A, \mathcal{B}; \theta \rangle \,|\, S \bullet \varPi$

$(\underline{\mathsf{unfold}})$ $\quad \langle B_1, \ldots, B_n, \mathsf{ret}(A), \mathcal{B}; \theta\sigma \rangle \,|\, S \bullet \mathsf{unf}(A,\theta,\ell) \colon \varPi \leftharpoonup \langle A, \mathcal{B}; \theta \rangle^{\ell} \,|\, S \bullet \varPi$
$\quad\qquad$ where $|\mathsf{body}(\ell, P)| = n$

$(\underline{\mathsf{exit}})$ $\quad \langle \mathcal{B}; \theta \rangle \,|\, S \bullet \mathsf{exit}(A) \colon \varPi \leftharpoonup \langle \mathsf{ret}(A), \mathcal{B}; \theta \rangle \,|\, S \bullet \varPi$

**Fig. 4.** Backward reversible semantics

**Lemma 1.** *Let $\mathcal{C}, \mathcal{C}'$ be configurations. If $\mathcal{C} \rightharpoonup \mathcal{C}'$, then $\mathcal{C}' \leftharpoonup \mathcal{C}$, up to variable renaming.*

*Proof.* The claim follows by a simple case distinction on the applied rule and the fact that the backward semantics of Figure 4 is trivially deterministic since each rule requires a different element on the top of the history.

In principle, one could also prove the opposite direction, i.e., that $\mathcal{C}' \leftharpoonup \mathcal{C}$ implies $\mathcal{C} \rightharpoonup \mathcal{C}'$, by requiring that $\mathcal{C}'$ is not an arbitrary configuration but a "legal" one, i.e., a configuration that is reachable by a forward derivation starting from some initial configuration.

The above result could be straightforwardly extended to derivations as follows:

**Theorem 4.** *Let $\mathcal{C}, \mathcal{C}'$ be configurations. If $\mathcal{C} \rightharpoonup^* \mathcal{C}'$, then $\mathcal{C}' \leftharpoonup^* \mathcal{C}$, up to variable renaming.*

## 4  A Reversible Debugger for Prolog

In this section, we present the design of a reversible debugger for Prolog. It is based on the standard 4-port tracer introduced by Byrd [2,3]. The ports are call (an atom is called), exit (a call is successfully completed), redo (backtracking requires trying again some call), and fail (an atom matches no clause). In contrast to standard debuggers that can only explore a computation forward, our reversible debugger allows us to move back and forth.

The implemented debugger, rever, is publicly available from https://github.com/mistupv/rever. It can be used in two modes:

- *Debug mode.* In this case, execution proceeds silently (no information is shown) until the execution of a special predicate rtrace/0 is reached (if any). The user can include a call to this predicate in the source program

```
Call: p(A,B)                    ^Exit: p(b,b)
Call: q(A)                      ^Exit: r(b,b)
Exit: q(a)                      ^Call: r(b,B)
Call: r(a,B)                    ^Exit: q(b)
Fail: r(a,B)                    ^Redo: q(A)
Redo: q(A)                      ^Fail: r(a,B)
Exit: q(b)                      ^Call: r(a,B)
Call: r(b,B)                    ^Exit: q(a)
Exit: r(b,b)                    ^Call: q(A)
Exit: p(b,b)                    ^Call: p(A,B)
**Answer: A = b, B = b
```

              (a)                             (b)

**Fig. 5.** Trace Example with rever

in order to start tracing the computation (i.e., it behaves as `trace/0` in most Prolog systems). Tracing also starts if an *exception* is produced during the evaluation of a query. This mode is invoked with a call of the form `rdebug(query)`, where `query` is the initial query whose execution we want to explore.

- *Trace mode.* In this mode, port information is shown from the beginning. One can invoke the trace mode with `rtrace(query)`. Note that it is equivalent to calling `rdebug((rtrace, query))`.

Our reversible debugger essentially implements the transition rules in Figures 2 and 4. As the reader may have noticed, some configurations in Figure 2 are labeled with a subscript: it denotes the output of a given port. Moreover, there is an additional label in rule next which denotes that, at this point, an answer must be shown to the user.

In tracing mode, every time that a configuration with a subscript is reached, the execution stops, shows the corresponding port information, and waits for the user to press some key. We basically consider the following keys: ↓ (or Enter) proceeds with the next (forward) step; ↑ performs a backward step; $s$ (for skip) shows the port information without waiting for any user interaction; $t$ enters the tracing mode; $q$ quits the debugging session.

For instance, given the initial call `rtrace(p(A, B))`, and according to the forward derivation shown in Figure 3, our debugger displays the sequence shown in Figure 5 (a). Now, if one presses "↑" repeatedly, the sequence displayed in Figure 5 (b) is shown. Note that ports are prefixed by the symbol "^" in backward derivations. Of course, the user can move freely back and forth.

Reversible debugging might be especially useful when we have an execution that produces some exception at the end. With our tool, one can easily inspect the execution backwards from the final state that produced the error.

Let us mention that, in order to avoid the use of a ground representation and having to implement all basic operations (mgu, substitution application and composition, etc), substitutions are represented in its equational form. E.g.,

substitution $\{A/a, B/b\}$ is represented by $A = a, B = b$. This equational representation of a mgu can be easily obtained by using the predefined predicate unify/3. This representation is much more efficient than storing pairs of atoms (as in [13]), that must be unified once and again at each execution step.

Finally, let us mention that, despite the simplicity of the implemented system (some 500 lines of code in SWI Prolog), our debugger is able to deal with medium-sized programs (e.g., it has been used to debug the debugger itself).

## 5   Related Work

The closest approach is clearly the preliminary version of this work in [13]. There are, however, several significant differences: [13] presents a reversible version of the usual, nondeterministic SLD resolution. Therefore, backtracking steps cannot be undone. This is improved in this paper by considering a deterministic semantics that models the traversal of the complete SLD tree. Moreover, [13] considers a simple but very inefficient representation for the history, which is greatly improved in this paper. Finally, we provide proofs of some formal properties for our reversible semantics, as well as a publicly available implementation of the debugger, the system rever.

Another close approach we are aware of is that of Opium [4], which introduces a trace query language for inspecting and analyzing trace histories. In this tool, the trace history of the considered execution is stored in a database, which is then used for trace querying. Several analysis can then be defined in Prolog itself by using a set of given primitives to explore the trace elements. In contrast to our approach, Opium is basically a so-called "post-mortem" debugger that allows one to analyze the trace of an execution. Therefore, the goal is different from that of this paper.

## 6   Concluding Remarks and Future Work

We have proposed a novel reversible debugging scheme for logic programs by defining an appropriate Landauer embedding for a deterministic operational semantics. Essentially, the states of the semantics are extended with a *history* that keeps track of all the information which is needed to be able to undo the steps of a computation. We have proved a number of formal properties for our reversible semantics. Moreover, the ideas have been put into practice in the reversible debugger rever, which is publicly available from https://github.com/mistupv/rever. Our preliminary experiments with the debugger have shown promising results.

As for future work, we are currently working on extending the debugger in order to cope with negation and the cut. Also, we plan to define a more compact representation for the history, so that it can scale up better to larger programs and derivations.

**Acknowledgements**

## References

1. Apt, K.R.: From logic programming to Prolog. Prentice Hall International series in computer science, Prentice Hall (1997)
2. Byrd, L.: Understanding the Control Flow of Prolog Programs. In: Tarnlund, S.A. (ed.) Proceedings of the 1980 Logic Programming Workshop. pp. 127–138 (1980)
3. Clocksin, W.F., Mellish, C.S.: Programming in Prolog (4. ed.). Springer (1994)
4. Ducassé, M.: Opium: An Extendable Trace Analyzer for Prolog. J. Log. Program. **39**(1-3), 177–223 (1999). https://doi.org/10.1016/S0743-1066(98)10036-5
5. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**, 183–191 (1961)
6. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. In: Pérez, J.A., Yoshida, N. (eds.) Proceedings of the 39th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019). Lecture Notes in Computer Science, vol. 11535, pp. 167–184. Springer (2019). https://doi.org/10.1007/978-3-030-21759-4_10
7. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987). https://doi.org/10.1007/978-3-642-83189-8
8. Mesnard, F., Payet, É., Vidal, G.: Concolic testing in logic programming. Theory Pract. Log. Program. **15**(4-5), 711–725 (2015). https://doi.org/10.1017/S1471068415000332
9. O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., Partush, N.: Engineering record and replay for deployability: Extended technical report. CoRR **abs/1705.05937** (2017), http://arxiv.org/abs/1705.05937
10. Ströder, T., Emmes, F., Schneider-Kamp, P., Giesl, J., Fuhs, C.: A linear operational semantics for termination and complexity analysis of ISO prolog. In: Vidal, G. (ed.) Proceedings of the 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011). Lecture Notes in Computer Science, vol. 7225, pp. 237–252. Springer (2011). https://doi.org/10.1007/978-3-642-32211-2_16
11. Stulova, N., Morales, J.F., Hermenegildo, M.V.: Assertion-based debugging of higher-order (C)LP programs. In: Chitil, O., King, A., Danvy, O. (eds.) Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014). pp. 225–235. ACM (2014). https://doi.org/10.1145/2643135.2643148
12. Undo Software: Increasing software development productivity with reversible debugging (2014), https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf
13. Vidal, G.: Reversible computations in logic programming. In: Lanese, I., Rawski, M. (eds.) Proceedings of the 12th International Conference on Reversible Computation (RC 2020). Lecture Notes in Computer Science, vol. 12227, pp. 246–254. Springer (2020). https://doi.org/10.1007/978-3-030-52482-1_15