Toward a concept of derivation for Prolog

Marija Kulaš

FernUniversität Hagen, Wissensbasierte Systeme, 58084 Hagen, Germany
kulas.marija@online.de

Abstract. The traditional concept of derivation in logic programming is adapted to meet some needs of implemented systems, such as explicit backtracking and aggregation of steps. Also, a way to enrich Horn clause logic with utilities like explicit unification and disjunction, but also cut or catch/throw, is suggested.

Keywords: built-in predicate, backtracking, big-step derivation, compositionality, cut

1 Introduction

This work is motivated by the wish to prove that an operational semantics of (slightly enriched) pure Prolog is adequate, in the sense of reflecting pure Prolog computation in a sound and complete way. Faced with such a task, the authors of an aspiring operational semantics X for (a subset of) Prolog have three ready options. They may deem X so straightforward that its adequacy seems obvious (e.g. for pure Prolog with cut [6], for full Prolog both [2] and our previous approach [9]). If they feel uncomfortable with that, they can prove something, but what would be "enough"? For example, it may be proved that any logical consequence of the program, and nothing else, can be derived in X [18]. This leaves out some operational aspects like the order of answers. Since the advent of ISO Standard Prolog, there is a third option: to brave its complex definition [5] and hopefully show correspondence with X, hereby contributing X to the pool of trusted semantics [17]. What we now suggest is a fourth option: to revisit the traditional concept of derivation in logic programming, *SLD-derivation* [11] [1], and try to make it a bit more accomodating to the needs of implemented logic programming. On this basis, correspondence to a formal model may be easier to prove.

SLD-derivations are suitable for claims about Horn clause logic (*HCL*), but not if its implementations are concerned, since it cannot express backtracking, and hence "spent" variables, or the order of answers, or the effect of cut. Also, it lacks abstraction facilities needed for meta-reasoning, like big step (grouping of steps) or compositionality with respect to conjunction. Yet, it has been commonly used as an operational model not only of HCL but of Prolog as well, e.g. in program analysis. Not surprisingly, a major impediment to proving properties was the abundance of freedom: there is freedom of variables (choice of mgu, and

of program clause renaming), and freedom of search (choice of an atom from a query, and of a program clause from a predicate definition). This has been approached by some complex notions like "equivalence modulo enhanced variance" [3]. As an alternative, we suggest a more pragmatic way: to acknowledge that, in an implemented system, many choices are fixed by algorithms. This way, we still may not know how the variables are going to be chosen, but we know that it is in one of a fairly small number of practical ways, obeying certain properties. For example, classical unification algorithms are renaming-compatible, which allows constructive versions of the variant lemma suitable for implemented systems. Such formal claims about logic programming systems or their compilation are still few and far between, a notable exception being [14].

Last but not least, it would be good to also have a way of accomodating utilities, usually called *built-in predicates*, in order to experiment with increasing the subset of Prolog (beyond HCL) that could be handled in a relatively intuitive yet formal way. For example, disjunction, implicit unification and truth values are obvious candidates, and utilities affecting search like "cut" could profit from a concise definition. Ideally, a refurbished concept of derivation should be reasonably obvious, to be trustworthy without the need for a proof.

1.1 Overview of the paper

For the most part we concentrate on enriched pure Prolog called Pure*. We start with a subset of HCL called HCL*, its syntax and proof method (Section 2). Next we consider restrictions due to implementation, turning HCL* into a programming language, Pure*, and propose a small-step concept of top-level derivation for Pure*, including backtracking (Section 3).

Next we attempt a big-step concept, based on *query context* (Section 4). In Subsection 4.1, backtracking is captured by means of *residual*, giving n-th answer. In Subsection 4.2, the missing transitivity of the path relation and the missing compositionality of answers are alleviated using *preface* of spent variables, so that computing a conjunction can be reduced to computing the conjuncts (Theorem 26). This bridges the gap between a traditional SLD-derivation that is rather devoid of structure, and a compositional formal semantics. A big-step version of variant lemma is sugested (Theorem 28).

Finally, in Section 5 it is shown how further utilities like cut or catch/throw could be added.

1.2 Notation

As a visual aid, the first defining mention of a symbol or concept shall be shown in blue. In Prolog, everything is a term, and so shall *term* be here the topmost syntactic concept. Terms are built in the usual way starting from two disjoint sets: a countably infinite set $\boldsymbol{V}$ of *variable*s and a set $\boldsymbol{Fun}$ of *functor*s. Associated with every functor f is one or more natural numbers n denoting its possible number of arguments, *arity*; for disambiguation, the notation f/n may be used, which is also the *outline* of the term $f(t_1, ..., t_n)$. If s occurs within t, we write

$s \mathring{\in} t$. A *list of n elements* $t_1, ..., t_n$ is written as $[t_1, ..., t_n]$. The *empty list* is written as $[]$. The set of variables in t is $Vars(t)$. A term without variables is a *ground term*. If r is obtained from s by replacing its variables in order of appearance with $t_1, ..., t_n$, we write $r = s[t_1, ..., t_n]$. A *substitution* is a mapping of variables on terms which is identity almost everywhere. If everywhere, it is denoted ε. A substitution σ is represented by its finite set of non-identity bindings as $\left(\begin{smallmatrix} x_1 & \cdots & x_n \\ \sigma(x_1) & \cdots & \sigma(x_n) \end{smallmatrix}\right)$, and $\sigma(t)$ is an *instance* of t. A *renaming* ρ is a substitution represented by a permutation; $\rho(t)$ is a *variant* of t. The set of most general unifiers (*mgu*s) of an equation E is denoted $MguSet(E)$. Finally, *nothing* (or *void*) is denoted as $\square$, *anything* as $\_$ and *impossibility* as $\bot$.

2 Modifying HCL toward HCL* and beyond

2.1 The syntax

In the course of enriching Prolog with ever more utilities (by means of *reserved functors*), like disjunction, unification or cut, the original syntax of HCL was stretched to accomodate those as well. As a result, mathematical meaning of "predicate" (logical relation) and "predication" (atomic sentence) has been blurred to a large extent, and logical connectives have been lost. This happened in two ways. First, by considering every reserved functor as a predicate, even a conjunction [5]. Second, by defining predication to be anything but variable or number, so it may even be a clause [4].

Indeed, there does not seem to be an easy way to merge utilities into logic. As a compromise, we suggest to consider only conjunction and clause-building as structural functors, and all other reserved functors as built-in predicates.

More formally, assume a set $\boldsymbol{P} \subseteq \boldsymbol{Fun}$ whose members shall be called *predicate*s. A term whose outline is a predicate shall be called a *predication*. Apart from predicates, ***Fun*** includes *structural outlines*, consisting of conjunction and reversed implication, $\boldsymbol{Str} := \{\texttt{','}/2,\ \texttt{':-'}/2\}$. A term with outline in ***Str*** is a *structured logical term*, which makes a predication *unstructured*, a mere *atom*. Thus, only conjunction and reversed implication are deemed connectives. Any utility we might add, like disjunction, shall be seen as restricting the choice of user-definable atom, since reserved functors cannot be redefined. So, extending the language with reserved functors can be seen as restricting Horn clauses.

Definition 1 (HCL*). $\boldsymbol{BiP^*} := \{\texttt{true}/0,\ \texttt{fail}/0,\ \texttt{true}/1,\ \texttt{'='}/2,\ \texttt{';'}/2\}$ *are built-in predicates of HCL*. A user-definable atom, for short uatom, is an atom whose outline is not in* ***BiP*****. A built-in atom is any other atom. A HCL* program is a set of program clauses, defined in Figure 1.*

Alongside `true`/0, there is a version with one dummy argument, `true`/1 (from an old Quintus Prolog package `true.pl`). A use for `true(t)` is to hold variables of the term t, similarly to the epsoid ε_t in [8]. Following [1], we leave out the negation prefix of queries. A fixed HCL* program is assumed.

$$
\begin{aligned}
\langle \mathit{HCL}^*\ \mathit{clause}\rangle &::= \langle \mathit{program\ clause}\rangle \mid \langle \mathit{query}\rangle \\
\langle \mathit{program\ clause}\rangle &::= \langle \mathit{uatom}\rangle\,\texttt{.} \mid \langle \mathit{uatom}\rangle\ \texttt{:-}\ \langle \mathit{qf}\rangle\,\texttt{.} \\
\langle \mathit{query}\rangle &::= \langle \mathit{qf}\rangle \\
\langle \mathit{qf}\rangle &::= \langle \mathit{atom}\rangle \mid \langle \mathit{atom}\rangle\,\texttt{,}\,\langle \mathit{qf}\rangle \\
\langle \mathit{atom}\rangle &::= \langle \mathit{uatom}\rangle \mid \texttt{true} \mid \texttt{fail} \mid \texttt{true(}\langle \mathit{qf}\rangle\texttt{)} \mid \langle \mathit{term}\rangle\ \texttt{=}\ \langle \mathit{term}\rangle \mid \langle \mathit{qf}\rangle\,\texttt{;}\,\langle \mathit{qf}\rangle
\end{aligned}
$$

Fig. 1. Syntax of HCL*

In a pragmatical deviation from the original meaning of clauses, we treat program clauses as *fixed terms*, having their variables chosen by the programmer, so a renamed program clause is not a program clause any more, but a *variant* of one. As a visual help, program clauses shall be written with a hat, like $\hat{\mathcal{K}}$.

2.2 The proof method

Can utilities be handled with the proof method of HCL, consisting of the resolution rule [15]? One idea is, if we can define a predicate in HCL, we can use it to extend HCL as well. Resolution rule derives from $\forall(\neg(A' \wedge B))$ and $\forall(A'' \vee \neg C)$ the formula $\forall(\neg(\sigma(C) \wedge \sigma(B)))$, where $\sigma \in \mathit{MguSet}(A'{=}A'')$ and A', A'', B, C are arbitrary. In HCL A', A'' are uatoms and B, C are conjunctions of uatoms, but there is nothing to preclude built-in atoms (like disjunction). Thus, we can *build* the predicate *in* HCL. As shown in [19], every computable function can be defined in HCL. So at least in theory many experiments are imaginable. Some of them are easy, like emulating truth values, unification and disjunction, as shown in Listing 1.1. Clearly, `fail` is emulated by omission.

```
true. true(_). X=X. X;Y :- X. X;Y :- Y.
```

Listing 1.1. Embedding HCL* in HCL

Emulating cut (and other utilities which affect backtracking) in HCL is bound to be less readable, here we resort to a meta-functor ***Back***. Thus, the proposed proof method for HCL* consists in resolution customized for utilities, plus customized rules for ***Back*** where backtracking has been affected.

Technically, to resolve built-in atoms, we shall separate the two parts of resolution: the replacing of an uatom A' with $\sigma(C)$, and the joining of the replacement with the (now suitably instantiated) rest of the original query, $\sigma(B)$. Let us call the former step *reduction*.

Definition 2 (reduction of uatom). *We say that a uatom A can be reduced to $\sigma(B)$ with score $\mathcal{K}{:}\sigma$, written as $A \rhd_{\mathcal{K}:\sigma} \sigma(B)$, if $\mathcal{K} = (H\ \texttt{:-}\ B)$ is a variant of a program clause $\hat{\mathcal{K}}$, such that $\sigma \in \mathit{MguSet}(H{=}A)$. A score can be applied on a term by $(\_ : \sigma)(t) := \sigma(t)$.*

For built-in atoms, we define reduction by the following rules using a ground term $\boldsymbol{k}$ called the *pseudo program clause*, and the *pseudo score* $\mathtt{o} := \boldsymbol{k}{:}\varepsilon$.

Definition 3 (reduction of built-in atom). *For HCL* the following reduction rules shall be added:*

- $\texttt{true} \rhd_{\mathsf{o}} \square$, *as well as* $\texttt{true}(\_) \rhd_{\mathsf{o}} \square$
- $X\texttt{=}Y \rhd_{\boldsymbol{k}:\sigma} \square$, *if* $\sigma \in \mathit{MguSet}(X{=}Y)$
- $X\,\texttt{;}\,Y \rhd_{\mathsf{o}} X$ *and* $X\,\texttt{;}\,Y \rhd_{\mathsf{o}} Y$

Based on reduction, we shall now define SLD*-resolution as an extension of SLD-resolution. Assumed is a rule for *selecting* an atom from a sequence of atoms, possibly relying on the previous course of computation [16, p. 62].

Definition 4 (SLD*-resolution and derivation). *Let A be an atom and M, N queries. If A is selected from M, A, N and for some s, B holds $A \rhd_{\mathsf{s}} B$, then we say $\mathsf{s}(M), B, \mathsf{s}(N)$ is an SLD*-resolvent of M, A, N with score s, and $M, A, N \hookrightarrow\!\rhd_{\mathsf{s}} \mathsf{s}(M), B, \mathsf{s}(N)$ is an SLD*-derivation step.*

An SLD-derivation $\boldsymbol{D}$ of a query G is a sequence of SLD*-derivation steps $G \hookrightarrow\!\rhd_{\mathsf{s}_1} G_1 \hookrightarrow\!\rhd_{\mathsf{s}_2} \ldots$. In each $\mathsf{s}_n = \mathcal{K}_n{:}\sigma_n$, $\mathcal{K}_n$ must be standardized apart (i.e., variable-disjoint) from the current history, $\boldsymbol{D}\restriction_{n-1} := G \hookrightarrow\!\rhd_{\mathsf{s}_1} \ldots \hookrightarrow\!\rhd_{\mathsf{s}_{n-1}} G_{n-1}$.*

If $\boldsymbol{D}$ has n steps, it can be abbreviated as $G \hookrightarrow\!\rhd^{n}_{\mathsf{s}_1+\ldots+\mathsf{s}_n} G_n$. Here $\mathsf{S} := \mathsf{s}_1 + \ldots + \mathsf{s}_n$ is the (cumulative) score of $\boldsymbol{D}$, denoted $\mathit{Score}(\boldsymbol{D})$, from which the list of input clauses, $\mathit{CList}(\mathsf{S}) := [\mathcal{K}_n, \ldots, \mathcal{K}_1]$, and the list of mgus, $\mathit{SList}(\mathsf{S}) := [\sigma_n, \ldots, \sigma_1]$, can be extracted. By composing the mgus we obtain the partial answer for G by $\boldsymbol{D}$, $\mathit{Subst}(\mathsf{S}) := \sigma_n \cdot \ldots \cdot \sigma_1$. A final partial answer, whenever $G_n = \square$, shall be called a complete answer for G. In that case G is said to succeed. The effect of a score S on a term t is given by $\mathsf{S}(t) := \mathit{Subst}(\mathsf{S})(t)$.

Thus, SLD*-derivation is defined almost like SLD-derivation. The main differences are the factoring-out of reduction, which allows utilities, and reviving *unrestricted* composition of mgus instead of the usual restriction on query variables (as in *c.a.s.*, computed answer substitution). Using input clauses (instead of program clauses) in scores enables a simple definition of derivation variables: $\mathit{Vars}(\boldsymbol{D}\restriction_{n-1}) := \mathit{Vars}((G, G_1, \ldots, G_{n-1}, \mathsf{s}_1, \ldots, \mathsf{s}_{n-1})) = \mathit{Vars}((G, \mathsf{s}_1, \ldots, \mathsf{s}_{n-1}))$. These variables are considered *spent* by step n.

Remark 5 (dual meaning of "derivation"). To save some space, "derivation" is here understood both as a *sequence of steps*, i.e. queries interspersed with scores, and as a *relation* between two queries.

Hence, we may say "let $\boldsymbol{D} := A \hookrightarrow\!\rhd^* B$ with no C within" as well as "let $A \hookrightarrow\!\rhd^* B$ hold". Similarly for all later kinds of derivation.

To visualize the search for a successful SLD*-derivation of a query, *SLD*-trees* are used, defined analogously to SLD-trees. Each node with more than one descendant is a *choice point*. Any node labeled $\square$ is a *success node*. If one SLD*-tree for a query is successful, then any one is [1, p. 70]. Hence, if one finite SLD*-tree for G is unsuccessful, then any one is, so we say G is *(finitely) failed*.

To systematically search for a success node, *backtracking* is used: If the search reaches a node without descendants, then it goes back to the last choice point, where a new choice shall be tried.

3 From HCL* to Pure*: Restrictions by implementation

Despite Prolog standard, there seems to be no consensus on a definition of pure Prolog [7], but since we need some kind of understanding, let us try this:

Definition 6 (pure Prolog, Pure*). *Pure Prolog is a programming language implementing HCL and obeying the following restrictions:*

(R-var) *variables are fixed: mgu is calculated by a fixed algorithm* $\boldsymbol{U}$*, and standardizing apart is calculated by a fixed algorithm* $\boldsymbol{S}$

(R-ord) *reduction order is fixed: it respects the ordering of atoms in a query, and the ordering of program clauses within a predicate definition.*

Similarly, Pure implements HCL* and obeys the same restrictions, plus* **(R-ord*)***: reduction rules for each utility are ordered.*

Every implementation iHCL* of HCL* obeying (R-var), hence any Prolog, is parametrized by two algorithms, $\boldsymbol{U}$ and $\boldsymbol{S}$. That can be made explicit by $\mathsf{iHCL}^*_{\boldsymbol{U},\boldsymbol{S}}$. To denote all implementations with the same $\boldsymbol{U}$, we write $\mathsf{iHCL}^*_{\boldsymbol{U}}$. For $\mathsf{iHCL}^*_{\boldsymbol{U}}$, a propagation claim leading to an appropriate variant-lemma can be proved as in [8, Sec. 6].

The examples in this text use $\boldsymbol{U} := \mathsf{MM}$ and $\boldsymbol{S} := \mathsf{NT}$, where MM is a deterministic version of Martelli-Montanari algorithm (with sequences instead of sets and picking the leftmost equation eligible for a rule application) and NT is defined below, after some preliminaries.

Standardizing-apart algorithms $\boldsymbol{S}$ can be seen as a special case of algorithms producing (relatively) fresh terms:

Definition 7 (fresh renaming). *A binary function New is a fresh-renaming algorithm, if* $New_s(t)$ *is a variant of* t *variable-disjoint with* s*, for any* s, t*.*

To enable accumulation of spent variables, we allow an optional parameter in superscript, so New^{+r} *is defined as* $New^{+r}_s(t) := New_{r,s}(t)$ *for any* s, t*.*

In theoretical work, the existence of $\boldsymbol{S}$ is usually enough, established by the well-known renaming device of Lloyd [11, p. 41]: For the n-th derivation step, the original program clause variables are indexed with n, assuming that top-level queries may not contain indices. This assumption, however, rules out *resuming* of a derivation, i.e. starting from a resolvent, which is needed for proofs involving modularity. As a remedy, instead of the whole $\boldsymbol{D}$ solely the *variables* of $\boldsymbol{D}$ can be taken as a parameter of the algorithm.

So we shall say that a standardizing-apart algorithm is *flat*, if it depends only on the spent variables, i.e. $\boldsymbol{S}_{\boldsymbol{D}}(\hat{\mathcal{K}}) = \boldsymbol{S}_{Vars(\boldsymbol{D})}\hat{\mathcal{K}}$. Such is the following fresh renaming algorithm, based on the idea to rename only where necessary:

Definition 8 (thrifty fresh renaming, NT). *The algorithm NT renames any variable* x *apart from* s *as follows. If* x *appears in* s*, together with its indexed versions* $x_1, ..., x_k$*, but* x_{k+1} *does not, then* x_{k+1} *shall replace* x*, i.e.* $\mathsf{NT}_s(x) := x_{k+1}$*. Otherwise,* x *remains unchanged.*

Adding the restrictions (R-ord) and (R-ord*) means that the order of resolvents for a query is determined. To formalize this, we need a new concept of derivation step, based on *ordered reduction*, to supersede $\hookrightarrow$.

Definition 9 (i-th program clause, i-th utility reduction). *Assume the predicate p/n is defined by m program clauses enumerated in the order of their appearance as $Clause(p/n, i)$ for $i = 1, ..., m$. If G has outline p/n, we also use $Clause(G, i)$. Similarly, in case of more than one reduction rule for a utility, they are ordered as $\triangleright^1, \triangleright^2, \ldots$.*

So for disjunction we would write:

$$X; Y \triangleright_{\mathtt{o}}{}^1 X$$
$$X; Y \triangleright_{\mathtt{o}}{}^2 X$$

In case of uatom, we may need some fresh variables, hence reduction has to avoid re-using already spent variables (expressed as a term P).

Definition 10 (i-th uatom reduction sparing P). *Let i be a number, P a term or void, and A an uatom. Sparing P, we say that A has i-th reduction to C with score $\mathtt{s} := \mathcal{K}{:}\sigma$, written as $A \triangleright_{\mathtt{s}}{}^{i,P} C$, if $\mathcal{K} = (H \mathtt{:-} B) := \boldsymbol{S}_P(Clause(A, i))$ and exists $\sigma := \boldsymbol{U}(H{=}A)$ and $C = \sigma(B)$.*

In terms of SLD*-trees, the restrictions amount to having only one SLD*-tree per query and searching the tree in depth-first manner. A single tree merits a name.

Definition 11 (LD^*-tree sparing P). *Let G be a query or void, and P be a term or void. The derivation tree for G sparing P, denoted as $LD^*{}_P(G)$, is the SLD*-tree for G identified by (R-var) with $\boldsymbol{S}^{+P}$, (R-ord) and (R-ord*).*

Observe that here again we allow some variables (P) to be already spent. This is to enable handling of sub-queries. Clearly, G is finitely failed in HCL* iff $LD^*{}_\_(G)$ is finite and without a success node.

Example 12. Assume $\boldsymbol{U} := \mathsf{MM}$, $\boldsymbol{S} := \mathsf{NT}$ and the program

```
alt(X) :- p(X), q(X).          % K̂a1
alt(X) :- s(X).                % K̂a2
p(a). p(b). q(b). t(1). t(3). % K̂p1, K̂p2, K̂q, K̂t1, K̂t2
```

The tree $LD^*{}_\square(\texttt{alt(X),r(X)})$ is shown in Figure 2. Here $\mathtt{s}_{a1} = \hat{\mathcal{K}}_{a1}[\mathtt{X}_1]{:}\binom{\mathtt{X}_1}{\mathtt{X}}$, $\mathtt{s}_{a2} = \hat{\mathcal{K}}_{a2}[\mathtt{X}_2]{:}\binom{\mathtt{X}_2}{\mathtt{X}}$, $\mathtt{s}_{p1} = \hat{\mathcal{K}}_{p1}{:}\binom{\mathtt{X}}{\mathtt{a}}$, $\mathtt{s}_{p2} = \hat{\mathcal{K}}_{p2}{:}\binom{\mathtt{X}}{\mathtt{b}}$ and $\mathtt{s}_q = \hat{\mathcal{K}}_q{:}\varepsilon$. Observe that we wrote $\hat{\mathcal{K}}_{p1}$ instead of $\hat{\mathcal{K}}_{p1}[]$ in the score $\mathtt{s}_{p1}$; in general, if the renaming does not change the program clause, we leave out the bracketed part.

On the basis of i-th reduction, we now define top-level Pure* computation of G. Intuitively, it is the left-to-right, depth-first path from the root of $LD^*(G)$ until possibly the first success node is encountered. Choices for G now being ordered, the computation must be deterministic. Its linear rendering shall be achieved with three devices:

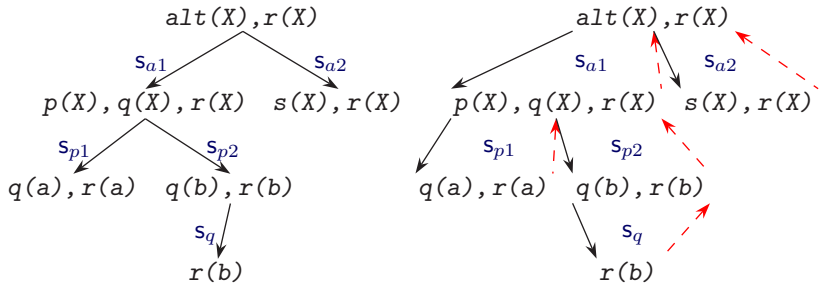

Fig. 2. LD^*-tree, on the right with backtracking

- negative score, written as $-\mathsf{s}$, serving to cancel a choice with score s
- meta-functor ***Back***, where ***Back*** G means looking for the next choice for G
- meta-functor ***Top***, serving as an artificial starting node, and thus ensuring that every query shall have a parent node

Notation 13 (zero score, net-value). Due to cancellation of reduction scores, we shall need *zero score*, denoted as $\emptyset$. A derivation for G shall start as ***Top*** $\rightarrow_\emptyset G$. As it proceeds, scores accumulate in S, some of them cancelled. Even those are important if we want to know which variables are already spent, hence $CList(\mathsf{S})$ is again defined to gather all input clauses from S, disregarding cancellation.

However, for the partial answer only *non-cancelled* scores are important, i.e. the *net-value* of S, $Net(\mathsf{S})$. It obeys $Net(\mathsf{T}+\mathsf{s}+(-\mathsf{s})+\mathsf{U}) = Net(\mathsf{T}+\emptyset+\mathsf{U})$, where s is a reduction score and T,U are sub-sums of a cumulative score S. Zero-score is neutral and obeys $Net(\emptyset) := \emptyset$, $SList(\emptyset) := []$, $CList(\emptyset) := []$ and $Subst(\emptyset) := \bot$. If we need only input clauses of $\boldsymbol{D}$ not belonging to cancelled scores, i.e. the list of *promising clause*s, we use $CList(Net(Score(\boldsymbol{D})))$.

Definition 14 (node). *If G is a query, then* ***Back*** *G is a revisited query. A node is a query or a revisited query. To provide for the limits, a node can also be $\Box$ (void) or* ***Top*** *(top-level).*

G or ***Back*** G shall be abbreviated as $(\textbf{\textit{Back}})G$, and amount to one node in the corresponding LD^*-tree (Figure 2).

Definition 15 (Pure^* derivation, top level). *Let G be a Pure^* query. The (top-level) Pure^* derivation for G in 0 steps is* ***Top*** $\rightarrow_\emptyset G$. *Assume $n \geq 1$ and let $\boldsymbol{D}$ be the derivation for G in $n-1$ steps ending with G_{n-1}, where $G_0 := G$.*

If G_{n-1} was of the form $\Box$ or ***Back Top****, no further step is possible. Otherwise, G_{n-1} is of the form $(\textbf{Back})A, R$ where A is an atom and R may be void. Let the most recent reduction of A starting a node $(\textbf{Back})A, R$ in $\boldsymbol{D}$ was m-th (in case of none, set $m := 0$). The n-th step is as follows:*

1. *If there is a next reduction for A, i.e. for some minimal $k > 0$ exists $m+k$-th reduction sparing $\boldsymbol{D}$ as $A \rhd_\mathsf{s}{}^{m+k,\boldsymbol{D}} C$, then $(\textbf{Back})A, R \rightarrow_\mathsf{s} C, \mathsf{s}(R)$.*

2. *Otherwise,* $(\textbf{\textit{Back}})A, R \to_{-\mathsf{s}} \textbf{\textit{Back}}\ B$, *where* $B/\mathsf{s} = \mathit{Parent}_n\{(\textbf{\textit{Back}})A, R\}$.

Here $\mathit{Parent}_n\{A\} = B/\mathsf{s}$, *if* $(\textbf{\textit{Back}})B \to_{\mathsf{s}} A$ *is the most recent forward step (i.e. step with a non-negative score) ending in* A *before step* n.

As usual, the *(top-level) Pure* computation* for G is the maximal Pure* derivation for G. The computation of `alt(X),r(X)` for the program in Example 12 is given in Figure 3. It is a linear rendering of $LD^*{}_\Box$(`alt(X),r(X)`) from Figure 2.

derivation step nr.	score	query *Top*	aspects of cumulative score S: promising input clauses $\mathit{CList}(\mathit{Net}(\mathsf{S}))$	part. answer $\mathit{Subst}(\mathit{Net}(\mathsf{S}))$
0.	$\to_{\emptyset}$	`alt(X),r(X)`	$[]$	$\bot$
1.	$\to_{\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}}$	`p(X),q(X),r(X)`	$[\hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X_1}{X}$
2.	$\to_{\hat{\mathcal{K}}_{p1}:\binom{X}{a}}$	`q(a),r(a)`	$[\hat{\mathcal{K}}_{p1}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X}{a} \cdot \binom{X_1}{X}$
3.	$\to_{-\hat{\mathcal{K}}_{p1}:\binom{X}{a}}$	***Back*** `p(X),q(X),r(X)`	$[\hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X_1}{X}$
4.	$\to_{\hat{\mathcal{K}}_{p2}:\binom{X}{b}}$	`q(b),r(b)`	$[\hat{\mathcal{K}}_{p2}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X}{b} \cdot \binom{X_1}{X}$
5.	$\to_{\hat{\mathcal{K}}_{q}:\varepsilon}$	`r(b)`	$[\hat{\mathcal{K}}_{q}, \hat{\mathcal{K}}_{p2}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\varepsilon \cdot \binom{X}{b} \cdot \binom{X_1}{X}$
6.	$\to_{-\hat{\mathcal{K}}_{q}:\varepsilon}$	***Back*** `q(b),r(b)`	$[\hat{\mathcal{K}}_{p2}, \hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X}{b} \cdot \binom{X_1}{X}$
7.	$\to_{-\hat{\mathcal{K}}_{p2}:\binom{X}{b}}$	***Back*** `p(X),q(X),r(X)`	$[\hat{\mathcal{K}}_{a1}[X_1]]$	$\binom{X_1}{X}$
8.	$\to_{-\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}}$	***Back*** `alt(X),r(X)`	$[]$	$\bot$
9.	$\to_{\hat{\mathcal{K}}_{a2}[X_2]:\binom{X_2}{X}}$	`s(X),r(X)`	$[\hat{\mathcal{K}}_{a2}[X_2]]$	$\binom{X_2}{X}$
10.	$\to_{-\hat{\mathcal{K}}_{a2}[X_2]:\binom{X_2}{X}}$	***Back*** `alt(X),r(X)`	$[]$	$\bot$
11.	$\to_{-\emptyset}$	***Back Top***	$[]$	$\bot$

Fig. 3. Linear rendering of a LD*-tree

Assuming the computation of G is finite, how could it have ended? By definition, if $G \to^* \Box$, the computation must stop. Also, if there is no choice for resolving G, then we readily obtain $\textbf{\textit{Top}} \to_{\emptyset} G \to_{-\emptyset} \textbf{\textit{Back Top}}$. But it can also happen that there are choices for G, yet none leads to $\Box$. In that case those choices shall be unraveled until their parent, G, is revisited, obtaining eventually $\textbf{\textit{Top}} \to_{\emptyset} G \to_{\mathsf{s}} H \ \ldots\ \to_{-\mathsf{s}} \textbf{\textit{Back}}\ G \to\ \ldots\ \to_{-\emptyset} \textbf{\textit{Back Top}}$.

Let us call a subderivation of the form $(\textbf{\textit{Back}})G \to_{\mathsf{s}} \ldots \to_{-\mathsf{s}} \textbf{\textit{Back}}\ G$ a *dead-end branch*. In Figure 3, steps 2-3, 5-6, 4-7, 1-8 and 9-10 are such branches.

The next claim states that backtracking happens stackwise: before we backtrack on a parent, we backtrack on all of its children.

Lemma 16 (zero score). *The score of a dead-end branch has net-value* $\emptyset$.

Knowing the boundaries of a computation enables us to define subnodes: If $A \to^* B$ such that no ***Back*** $\mathit{Parent}\{A\}$ is within, we say B is a *subnode* of A.

Now let us see how to recognize success or finite failure (as defined in HCL*) in a Pure*derivation. By stripping of dead-end branches, we obtain

Lemma 17 (success/failure). *If* $G \rightarrow^*_{\mathsf{S}} \square$*, then* $G \hookrightarrow\!\triangleright^*_{Net(\mathsf{S})} \square$*, i.e. G succeeds in HCL***. If* $G \rightarrow^*$ ***Back Top****, then G fails finitely in HCL***.*

Remark 18 (past is now included). In contrast to SLD*-derivations, a Pure* derivation does not consist only of "promising" resolvents, but also of "dead-end" resolvents: anything that was tried (and possibly failed) along the way to the current query. Clearly, this must be taken into account for proofs involving Pure* derivations.

An appropriate variant lemma for Pure* derivations would be similar to the variant lemma for $\mathsf{iHCL}_{\boldsymbol{U}}$ [8, Sec. 6] except that (due to backtracking) the step variance β_n now cannot map the whole $\boldsymbol{D}\restriction_n$ on its counterpart $\boldsymbol{D'}\restriction_n$, but just the current query G_n and cumulative score $\mathsf{s}_1 + ... + \mathsf{s}_n$.

4 Toward big-step derivation

As seen in Section 3, implementing a logic programming language removes many sources of non-determinism. Must we now see Pure* computation of a query as unique ("the computation"), or still as one of many? We advocate the latter view, justified by a somewhat neglected further source of non-determinism: the *context*. At first glance, computing a Pure* query G should be "the same" independent of what preceded it and what is about to follow. Yet this is not quite so:

- Without queries to follow G, no backtracking on G shall be attempted. Otherwise, more than one answer for G is possible, giving shorter or longer computations for G.
- Standardizing apart depends on a pool of available variables, so if there were queries preceding (or following) G, their variables are not available: they are "spent". Thus, computations of G may differ in fresh variables.

4.1 Backtracking cause: residual

The top-level computation of G (Definition 15) stops in case of finite failure, or the first answer. To try to capture the search for *next* answer, i.e. backtracking on G, we have to consider G not alone but as part of some conjunction G, R. In other words, we need a "residual" query R coming after G. Without any such, there would be no need to ever[1] go back on G. Residual is nothing else than the *success continuation* for a Prolog query. It is used, with or without a name, in [10], [9], [4]. In [3], residual is modeled by "conjunction of two derivations".

Normally, a *residual* is just a query, but to include top-level computation we allow it to be void. Unlike the (first and only) answer for G in a top-level derivation, the n-th answer for G upon R is computed intermittently, in a sequence of "spells": after answering G, the residual is going to be computed, and if it fails, another answer for G shall be sought, and so on.

[1] In a Prolog top-level loop, the user's reaction can be thought of as the residual.

Definition 19 (initial spell). *Let G be a query and R a query or void. The initial spell of G upon R is the maximal derivation starting with G, R and not having inside it an instance of R, or* ***Back*** *Parent$\{G, R\}$.*

Assume the initial spell of G upon R was finite, with score S. By definition of resolution, if the spell ends with an instance of R, then it must be $\mathsf{S}(R)$. Otherwise the spell ends in ***Back*** *Parent*$\{G, R\}$ and $Net(\mathsf{S}) = \emptyset$.

Definition 20 ($n+1$-th spell). *Let $n \geq 1$. Assume the n-th spell of G upon R was finite, with $Net(\mathsf{S}_n) \neq \emptyset$. If $\mathsf{S}_n(R)$ fails, then for some $\mathtt{s}$ holds $G, R \rightarrow^* (\textbf{\textit{Back}})\mathsf{S}_n(R) \rightarrow_{-\mathtt{s}}$ **Back** Parent$\{\mathsf{S}_n(R)\}$.*

*In that case, there is $n+1$-th spell of G upon R. It is analogously defined as the initial spell, but starting with $(\textbf{\textit{Back}})\mathsf{S}_n(R) \rightarrow_{-\mathtt{s}}$ **Back** Parent$\{\mathsf{S}_n(R)\}$.*

A Pure* *computation* of G is a chronological sequence of all spells of G upon R, for some residual R. A *derivation* is an initial fragment of a computation, as usual. In Figure 3, there are two success spells of `p(X)` upon `q(X),r(X)` before the final spell (exhaustion): step 2, steps 3-4 and steps 7-8.

Definition 21 (n-th answer). *Assume $n \geq 1$ and the n-th spell of G upon R is finite with score S_n. If $Net(\mathsf{S}_n) \neq \emptyset$, we say G succeeded upon residual R with n-th complete answer S_n, written as $G \setminus_{\mathsf{S}}^n R$.*

The justification for calling S_n an "answer" for G lies in

Lemma 22 (all answers). *If $G \setminus_{\mathsf{S}}^n \mathtt{fail}$, then $G \hookrightarrow\!\triangleright^*_{Net(\mathsf{S})} \Box$. Conversely, if $G \hookrightarrow\!\triangleright^*_{\mathsf{T}} \Box$, then for some n, S holds $G \setminus_{\mathsf{S}}^n \mathtt{fail}$ and $\mathsf{T} = Net(\mathsf{S})$.*

Just the fresh variables in the scores may vary, depending on R: If $G \setminus_{\mathsf{S}_1}^n R_1$ and $G \setminus_{\mathsf{S}_2}^n R_2$, then $\mathsf{S}_1(G) = \mathsf{S}_2(G)$.

Traditionally, universal termination of G is defined as finiteness of $LD^*(G)$. An equivalent definition would be: G terminates universally if and only if $G, \mathtt{fail}$ terminates [12]. This is another nice use for residuals.

Definition 23 (termination, n-finiteness). *If G (finitely) fails or succeeds, we say it terminates. We say that G is n-finite, if there is a maximal n such that for some R holds $G \setminus^n R$. If G fails, it shall be called 0-finite. Any n-finite query is said to terminate universally, or to be exhaustible.*

In Example 12, `alt(X),r(X)` finitely failed (0-finite), but `alt(X)` upon `r(X)` is 1-finite with complete answer $\binom{X_1\ X}{b\ b}$, and `p(X)` upon `q(X),r(X)` is 2-finite with complete answers $\binom{X}{a}$ and $\binom{X}{b}$.

4.2 Modularity: impact of spent variables

For proving properties of a formal model of Pure*, it would be good to have a kind of modularity of Pure* derivations. It rests upon two questions: first,

could a derivation be resumed, and second, could a conjunction be computed in a piecemeal fashion?

Even in HCL*, the path relation is not transitive: If $A \hookrightarrow\!\triangleright^*_{\mathsf{S}} B$ and $B \hookrightarrow\!\triangleright^*_{\mathsf{T}} C$, it would be expected that also $A \hookrightarrow\!\triangleright^*_{\mathsf{S}+\mathsf{T}} C$, or at least $A \hookrightarrow\!\triangleright^* C$. But regretfully this does not always hold, as the following program shows.

```
p(X) :- q(Y).  % K̂1
q(Y) :- r(X).  % K̂2
```

Here $\texttt{p(X)} \hookrightarrow\!\triangleright_{\hat{\mathcal{K}}_1[X_1,Y]:\binom{X_1}{X}} \texttt{q(Y)}$ and $\texttt{q(Y)} \hookrightarrow\!\triangleright_{\hat{\mathcal{K}}_2[X,Y_1]:\binom{Y_1}{Y}} \texttt{r(X)}$, although $\texttt{p(X)} \hookrightarrow\!\triangleright^* \texttt{r(X)}$ is not possible, due to standardizing-apart.

Clearly, the reason for non-transitivity is that a derivation step has an *implicit parameter*, the current history. By making it explicit, and choosing $\boldsymbol{S}$ to be flat, we obtain an ersatz for transitivity:

Lemma 24 (resumability). *Let $\boldsymbol{S}$ be flat. For any queries A, B, C and any term or void P satisfying $Vars((P, B)) = Vars((A, \mathsf{S}, B))$ holds: If $A \to^*_{\mathsf{S}} B$ and $\texttt{true(P)}, B \to^+_{\mathsf{T}} C$, then $A \to^*_{\mathsf{S}+\mathsf{T}} C$.*

Regarding compositionality, for equations there is an iteration property [1] enabling us to compute an mgu ϕ of E', E'' by computing an mgu σ for E', then an mgu θ for $\sigma(E'')$, giving $\phi := \theta \cdot \sigma$ (this also holds for "the" mgu by MM). In case of general queries, such an iteration property does not hold for either complete answer or c.a.s.. By accomodating history again, it can hold for complete answer, as stated below (Theorem 26).

Notation 25 (preface). Let P be a term or void. We abbreviate $\texttt{true(P)}, G \setminus^n_{\mathsf{S}} R$ as $P \setminus\!\setminus G \setminus^n_{\mathsf{S}} R$, and say that P is a *preface* for computing G.

Theorem 26 (iteration for complete answer). *Let $\boldsymbol{S}$ be flat. Let A, B, R be queries, and P be a term or void. Then:*

1. *If $P \setminus\!\setminus A \setminus_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \setminus\!\setminus \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$, then also $P \setminus\!\setminus A, B \setminus_{\mathsf{S}+\mathsf{T}} R$.*
2. *Conversely, if $P \setminus\!\setminus A, B \setminus_{\mathsf{U}} R$, then there are S, T such that $P \setminus\!\setminus A \setminus_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \setminus\!\setminus \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$ and $\mathsf{U} = \mathsf{S} + \mathsf{T}$.*

Using big-step, the order of answers for a conjunction can also be expressed in a succinct yet still readable way (without proof).

Lemma 27 (n-th complete answer). *Let $\boldsymbol{S}$ be flat. The query A, B has at least n complete answers upon R, written as $A, B \setminus^n_{\mathsf{U}_n} R$, iff there are natural numbers p, $m_1, ..., m_p$ and $n_1, ..., n_p$ such that $0 = m_0 < m_1 < ... < m_p$ and $n_1 + ... + n_p = n$, and also*

1. *For $j = 1, ..., m_p$ and some S_j holds $A \setminus^j_{\mathsf{S}_j} B, R$.*
2. *For $i = 1, ..., p$ holds: $\mathsf{S}_j(B)$ fails whenever $m_{i-1} < j < m_i$, but for $k = 1, ..., n_i$ and some $\mathsf{T}_{i,k}$ holds $A, \mathsf{S}_{m_i} \setminus\!\setminus \mathsf{S}_{m_i}(B) \setminus^k_{\mathsf{T}_{i,k}} \mathsf{S}_{m_i}(R)$.*
3. *$\mathsf{S}_{m_i}(B)$ is n_i-finite for $i = 1, ..., p-1$.*

Finally, for $i = 1, ..., p$ *and* $k = 1, ..., n_i$ *holds* $\mathsf{U}_{n_1+...+n_{i-1}+k} = \mathsf{S}_{m_i} + \mathsf{T}_{i,k}$.

To conclude, an appropriate variant lemma for Pure* derivations, outlined at the close of the previous section, can be reformulated with big-step as follows.

Theorem 28 (variant, big-step). *Assume a renaming-compatible* $\mathbf{U}$ *producing relevant mgus, i.e. mgus without extraneous variables. Let* α *be a prenaming*[2] *with* $C(\alpha) = Vars((P, Q, R))$. *If* $P \setminus\!\setminus Q \setminus_{\mathsf{S}} R$, *then* $\alpha(P) \setminus\!\setminus \alpha(Q) \setminus_{(\alpha \uplus \lambda)(\mathsf{S})} \alpha(R)$, *where* λ *is mapping the promising input clauses of the underlying small-step derivations.*

5 Adding utilities that modify backtracking

Tampering with backtracking as in the case of the "cut" functor `!`/0 or the pair `catch`/3 and `throw`/1 can also be handled by customized reduction, plus one or two special-case rules for ***Back***. Compared to [5], the (re-)execution of cut is now more readable but still precise.

Observe: when adding a utility, care must be taken to ensure Lemma 16 still holds, in order to have correct scores.

5.1 Emulating cut

Upon execution, the cut succeeds like `true`/0, but upon re-execution it prunes the LD*-tree. This can be emulated by its reduction to nothing and an additional rule for ***Back***. The rule shall handle the special case of revisiting a query starting with cut, and shall in that case replace the default rules 1 and 2 from Definition 15.

$$\frac{\texttt{!} \rhd_{\mathsf{o}} \square}{\textbf{\textit{Back}}\ \texttt{!},R \to_{-\mathsf{S}} \textbf{\textit{Back}}\ \mathit{Parent}\{\mathit{CutParent}\{\texttt{!},R\}\}}$$

Here the *cut-parent* of a node is the most recent node whose resolution produced this occurence of cut (i.e., the node that "issued" this cut).

Finally, S is the score of the subderivation between $\mathit{Parent}\{\mathit{CutParent}\{\texttt{!},R\}\}$ and ***Back*** `!`$,R$. The purpose of the rule is to discard any alternatives between ***Back*** `!`$,R$ and the parent of the cut-parent.

Example 29 (cut). If we change $\hat{\mathcal{K}}_{a1}$ in Example 12 to

```
alt(X) :- p(X), !, q(X).   % new K̂a1
```

we obtain the computation in Figure 4, with $\mathsf{S} = \hat{\mathcal{K}}_{t1}\colon \binom{\mathtt{Y}}{1} + \hat{\mathcal{K}}_{a1}[\mathtt{X}_1]\colon \binom{\mathtt{X}_1}{\mathtt{X}} + \hat{\mathcal{K}}_{p1}\colon \binom{\mathtt{X}}{\mathtt{a}} + \mathsf{o} - \mathsf{o}$. Notice that the second clause for `alt`/1 shall not be tried.

[2] A *prenaming* α is a variable-pure substitution in a so-called *relaxed core representation*, i.e. some pairs x/x are allowed, with mutually distinct variables not only in its core $C(\alpha)$ but in the range $\alpha(C(\alpha))$ as well [8].

```
                          t(Y), alt(X), r(X)
→_{K̂_{t1}:(Y 1)}          alt(X), r(X)
→_{K̂_{a1}[X1]:(X1 X)}     p(X), !, q(X), r(X)
→_{K̂_{p1}:(X a)}          !, q(a), r(a)
→_o                       q(a), r(a)
→_{-o}                    Back !, q(a), r(a) % cut-parent: alt(X), r(X)
→_{-S}                    Back t(Y), alt(X), r(X)
→_{K̂_{t2}:(Y 3)}          ...
```

Fig. 4. Emulating cut

Having cut enables us to emulate several more utilities:

$$\begin{aligned}
\texttt{once(}G\texttt{)} &\rhd_o G\texttt{,!}\\
\mathit{If}\texttt{->}\mathit{Then}\texttt{;}\mathit{Else} &\rhd_o \texttt{call(}\mathit{If}\texttt{),!,}\mathit{Then}\texttt{;}\mathit{Else}\\
\mathit{If}\texttt{->}\mathit{Then} &\rhd_o \texttt{call(}\mathit{If}\texttt{),!,}\mathit{Then}\texttt{;fail}\\
\backslash\texttt{+}G &\rhd_o \texttt{call(}G\texttt{),!,fail;true}
\end{aligned}$$

Standard-abiding opacity for cut ([13, pp. 100-101], [5]) must be ensured for the reduction of any utility based on cut, like if-then-else[3]. So *Then* or *Else* must be "transparent" for cut, and *If* must be "opaque" for cut. This is ensured in our reduction of *If*`->`*Then*`;`*Else* by means of `call`/1. The utility `call`/1 is emulated simply as

$$\texttt{call(}G\texttt{)} \rhd_o G$$

By enveloping its argument, `call`/1 makes it opaque for cut. Hence, for possible cut in *Then* or *Else* the cut-parent must be `(`*If*`->`*Then*`;`*Else*`),`*R*, but for cut in *If* the cut-parent must be more recent, `call(`*If*`),!,`*Then*`,`*R*.

5.2 Emulating catch/throw

For `catch`/3 and `throw`/1, the following reduction rules can be used:

$$\begin{aligned}
\texttt{catch(}G\texttt{,}\mathit{Catcher}\texttt{,}\mathit{Recovery}\texttt{)} &\rhd_o G\\
\texttt{throw(}\mathit{Ball}\texttt{)} &\rhd_o \texttt{fail}
\end{aligned}$$

Reducing `throw`/1 to `fail`/0 is not the same as default failure by lack of program clauses: it enables backtracking on `throw`/1. On backtracking, `throw`/1 springs to life:

$$\begin{aligned}
&\textbf{\textit{Back}}\ \texttt{throw(}B\texttt{)}, R \rightarrow_o \sigma((\mathit{Rec}, R'))\\
&\textbf{\textit{Back}}\ \texttt{throw(}B\texttt{)}, R \rightarrow_{-S} \textbf{\textit{Back}}\ \mathit{Parent}\{\texttt{catch(}G\texttt{,}C\texttt{,}\mathit{Rec}\texttt{)}, R'\}
\end{aligned}$$

[3] For backtracking reasons, if-then-else is here not treated as a special case of disjunction, but as a ternary functor.

Here we have not one but two ***Back*** rules for revisiting query `throw(`B`)`, R, to replace respectively the default rules 1 ("next choice") and 2 ("no more choices") from Definition 15. The latter serves to discard any alternatives between the current `throw` and the parent of the issuing `catch`.

To fill in the missing variables G, C, Rec, R' as well as σ, observe: If the utility pair `catch`/`throw` is used in a standard-abiding way [4], then `throw(`B`)` was issued by G from some `catch(`G`,`C`,`Rec`)`, and B is unifiable with C. In that case, `throw(`B`),`R must be a subnode of the most recent `catch(`G`,`C`,`Rec`),`R', with existing $\sigma := \boldsymbol{U}(\boldsymbol{S}_C(B), C)$. Also, it must be a subnode of `,`G`,`R'.

Finally, S is the score to be discarded, belonging to the subderivation between *Parent*{`catch(`G`,`C`,`Rec`),`R'} and the current ***Back*** `throw(`B`),`R.

Example 30 (catch/throw). By changing $\hat{\mathcal{K}}_q$ in Example 12 to

```
q(b) :- throw(xb).   % new K̂q
```

the computation in Figure 5 is obtained, with $\mathsf{S} = \hat{\mathcal{K}}_{t1}{:}\binom{Y}{1} + \ \ldots\ - \mathsf{o}$.

	`t(Y), catch(alt(X),B,rx(B)), r(X)`
$\rightarrow_{\hat{\mathcal{K}}_{t1}:\binom{Y}{1}}$	`catch(alt(X),B,rx(B)), r(X)`
$\rightarrow_{\mathsf{o}}$	`alt(X), r(X)`
$\rightarrow_{\hat{\mathcal{K}}_{a1}[X_1]:\binom{X_1}{X}}$	`p(X), q(X), r(X)`
$\rightarrow_{\hat{\mathcal{K}}_{p1}:\binom{X}{a}}$	`q(a), r(a)`
$\rightarrow_{-\hat{\mathcal{K}}_{p1}:\binom{X}{a}}$	***Back*** `p(X), q(X), r(X)`
$\rightarrow_{\hat{\mathcal{K}}_{p2}:\binom{X}{b}}$	`q(b), r(b)`
$\rightarrow_{\hat{\mathcal{K}}_{q}:\varepsilon}$	`throw(xb), r(b)`
$\rightarrow_{\mathsf{o}}$	`fail, r(b)`
$\rightarrow_{-\mathsf{o}}$	***Back*** `throw(xb), r(b)`
$\rightarrow_{\mathsf{o}}$	`rx(xb), r(X)` *% =* σ*((*`rx(B),r(X)`*)) with* $\sigma = \{$`B/xb`$\}$
$\rightarrow_{-\mathsf{o}}$	***Back*** `throw(xb), r(b)` *% due to lack of* `rx`/1
$\rightarrow_{-\mathsf{S}}$	***Back*** `t(Y), catch(alt(X),B,rx(B)), r(X)`
$\rightarrow_{\hat{\mathcal{K}}_{t2}:\binom{Y}{3}}$	...

Fig. 5. Emulating catch/throw

6 Summary

A way to add utilities to pure Prolog and still remain in resolution logic is outlined. To this aim, the concept of structural functor is suitably restricted, so that "atom" (unstructured logical term) can be a utility, and *reduction* of an atom is factored out of resolution, to be extended with utilities. The programming language obtained by adding truth values, explicit unification and explicit disjunction is called *Pure**. To represent Pure* computation, we start from the traditional concept of SLD-derivation and add backtracking steps necessary for

implemented logic. A big-step concept of derivation is also suggested. It is built around *query context*, which is two-fold, consisting of the prequel to the query (*preface*) and its sequel (*residual*). *Residual* is the cause of backtracking. *Preface* provides history of computation. With context, not only *top-level* but also *sub*-computation can be expressed. As an aside, the concept of backtracking steps can accomodate "cut" and other utilities that affect backtracking.

Acknowledgement

Many thanks are due to anonymous referees.

References

1. Apt, K.R.: From logic programming to Prolog. Prentice Hall (1997)
2. Börger, E., Rosenzweig, D.: A mathematical definition of full Prolog. Sci. of Computer Programming 24(3), 249–286 (1995)
3. Comini, M., Meo, M.C.: Compositionality properties of SLD-derivations. Theor. Comp. Sci 211, 275–309 (1999)
4. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog: The standard (reference manual). Springer-Verlag (1996)
5. ISO: Information technology - Programming languages - Prolog - Part 1: General core. ISO/IEC JTC 1/SC 22 (1995), iSO/IEC 13211-1-1995. `https://www.iso.org/standard/21413.html`
6. Jones, N.D., Mycroft, A.: Stepwise development of operational and denotational semantics for Prolog. In: Proc. of the ISLP'84. pp. 281–288. Atlantic City (1984)
7. Kifer, M.: (2005), `https://www.w3.org/2005/rules/wg/wiki/Pure_Prolog.html`
8. Kulaš, M.: A practical view on renaming. In: Schwarz, S., Voigtländer, J. (eds.) Proc. WLP'15/'16 and WFLP'16. EPTCS, vol. 234, pp. 27–41 (2017)
9. Kulaš, M., Beierle, C.: Defining Standard Prolog in rewriting logic. In: Futatsugi, K. (ed.) Proc. of WRLA'00. ENTCS, vol. 36, pp. 158–174. Elsevier (2001)
10. Lindgren, T.: A continuation-passing style for Prolog. In: Proc. SLP'94. pp. 603–617 (1994)
11. Lloyd, J.W.: Foundations of logic programming. Springer-Verlag, 2. edn. (1987)
12. Neumerkel, U.: Teaching Prolog and CLP. In: Tutorial notes of ICLP'97 (1997)
13. O'Keefe, R.A.: The Craft of Prolog. The MIT Press (1990)
14. Pusch, C.: Verification of compiler correctness for the WAM. In: Proc. TPHOLs. LNCS, vol. 1125, pp. 347–361. Springer Berlin Heidelberg (1996)
15. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. of ACM 12(1), 23–41 (1965)
16. Shepherdson, J.C.: Negation as failure: A comparison of Clark's completed data base and Reiter's closed world assumption. J. Logic Programming 1, 51–79 (1984)
17. Ströder, T., Emmes, F., Schneider-Kamp, P., Giesl, J., Fuhs, C.: A linear operational semantics for termination and complexity analysis of ISO Prolog. In: Proc. LOPSTR'11. LNCS, vol. 7225, pp. 237–252 (2012), proofs in: Technical Report AIB-2011-08, RWTH Aachen
18. Tobermann, G., Beckstein, C.: What's in a trace: The box model revisited. In: Proc. of AADEBUG'93. LNCS, vol. 749, pp. 171–187. Springer-Verlag (1993)
19. Šebelík, J., Štěpánek, P.: Horn clause programs for recursive functions. In: Clark, K.L., Tärnlund, S.A. (eds.) Logic Programming. Academic Press (1982)

A Proofs

A.1 Auxiliary claims

We start with a simple variable-conservation lemma. By applying a substitution on a term, the term may lose some or win some variables, but any changes are balanced by the substitution.

Lemma 31 (no change). *For any term t and any substitution σ holds*

$$\mathit{Vars}(t) \cup \mathit{Vars}(\sigma) = \mathit{Vars}(\sigma(t)) \cup \mathit{Vars}(\sigma)$$

Proof. Let $x \mathrel{\mathring{\in}} t$. If $x \notin \mathit{Core}(\sigma)$, then $x = \sigma(x) \mathrel{\mathring{\in}} \sigma(t)$. In other words, any variables from t that are missing in $\sigma(t)$ can be found in $\mathit{Core}(\sigma)$. Analogously for the possible win. ♢

In the following we shall need a symbol for a reduction of A that is actually *reachable* with the given residual R. We also assume a preface P.

Due to zero score of a dead-end branch, the score of the reduction is equal to (the net-value of) the score of a backtracking derivation leading to it.

Notation 32 (reduction in context). Given is a query G, a term P and a query (or void) R. If for some C, S holds $\mathtt{true}(P), G, R \to_{\mathsf{S}}^{+} C, \mathsf{S}(R)$ and $G \rhd_{Net(\mathsf{S})} C$, that shall be shorter written as $P \lll G \rhd C \setminus_{\mathsf{S}} R$.

In Theorem 26, we show an attempt to uncover an iteration property for complete answer. But first we need to handle the case of one resolution step.

The direct part of the base case is straightforward, it just rephrases a resolution step by means of "$\setminus$".

Lemma 33 (split). *Let A be an atom. If $P \lll A, B \setminus_{\mathsf{S}} R$, then for some $C, \mathsf{T}, \mathsf{U}$ holds $P \lll A \rhd C \setminus_{\mathsf{T}} B, R$ and $P, A, \mathsf{T} \lll C, \mathsf{T}(B) \setminus_{\mathsf{U}} \mathsf{T}(R)$ and $\mathsf{S} = \mathsf{T} + \mathsf{U}$.*

The interesting part is the converse, claiming that a derivation starting from a resolvent and sparing the original query and the score of the resolution is indistinguishable from the original derivation. In other words, the outcome of a resolution step should be determined by the algorithms $\boldsymbol{U}$, $\boldsymbol{S}$ and the spent variables, and by nothing else. Therefore, $\boldsymbol{S}$ must be flat.

Lemma 34 (assemble). *Let $\boldsymbol{S}$ be flat. Let A be an atom. If for some $C, \mathsf{S}, \mathsf{T}$ holds*

$$P \lll A \rhd C \setminus_{\mathsf{S}} B, R \quad \textit{and} \quad P, A, \mathsf{S} \lll C, \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$$

then also holds $P \lll A, B \setminus_{\mathsf{S}+\mathsf{T}} R$.

Proof. From $P \lll A \rhd C \setminus_{\mathsf{S}} B, R$ follows

$$\mathtt{true}(P), A, B, R \to_{\mathsf{S}}^{+} C, \mathsf{S}(B), \mathsf{S}(R) \tag{1}$$

From $P, A, \mathsf{S} \setminus\!\setminus C, \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$ follows

$$\texttt{true}((P, A, \mathsf{S})), C, \mathsf{S}(B), \mathsf{S}(R) \to^{+}_{\mathsf{T}} \mathsf{T}(\mathsf{S}(R)) \tag{2}$$

Before Lemma 24 could be applied, the variables of (2) must obey

$$\mathit{Vars}((P, A, \mathsf{S}, C, \mathsf{S}(B), \mathsf{S}(R))) = \mathit{Vars}((P, A, B, R, \mathsf{S}))$$

By Lemma 31 we know $\mathit{Vars}(B) \cup \mathit{Vars}(\mathsf{S}) = \mathit{Vars}(\mathsf{S}(B)) \cup \mathit{Vars}(\mathsf{S})$, and similarly for R. Also, $\mathit{Vars}(C) \subseteq \mathit{Vars}((A, \mathsf{S}))$.

Therefore, Lemma 24 may be applied on (1) and (2), giving

$$\texttt{true}(P), A, B, R \to^{+}_{\mathsf{S}+\mathsf{T}} \mathsf{T}(\mathsf{S}(R))$$

The necessary border condition is ensured by the two original derivations. ♢

A.2 Some claims from the main text

Lemma 16 (zero score). The score of a dead-end branch has net-value $\emptyset$.

Proof. For the branch $\boldsymbol{Top} \to_{\emptyset} G \to_{-\emptyset} \boldsymbol{Back}\ \boldsymbol{Top}$ the claim is obvious. Let $\boldsymbol{D}$ be $(\boldsymbol{Back})G \to_{\mathsf{s}} H \ \ldots \ (\boldsymbol{Back})H \to_{-\mathsf{s}} \boldsymbol{Back}\ G$, where $(\boldsymbol{Back})G$ is an abbreviation of "G or $\boldsymbol{Back}\ G$". We shall use induction on the number n of inner nodes. If $n = 1$, the derivation is $(\boldsymbol{Back})G \to_{\mathsf{s}} H \to_{-\mathsf{s}} \boldsymbol{Back}\ G$, and it satisfies the claim.

Assume the claim holds for up to $n \geq 1$ inner nodes and consider a derivation with $n+1$ inner nodes. To cater for more than one inner node, there must have been choices for H, finitely many (guaranteed by $\boldsymbol{Back}\ H \to_{-\mathsf{s}} \boldsymbol{Back}\ G$). So let H have m choices. Then the derivation looks like

$$\begin{aligned}
&(\boldsymbol{Back})G \to_{\mathsf{s}} H \\
&\to_{\mathsf{t}_1} F_1 \ \ldots \ (\boldsymbol{Back})F_1 \to_{-\mathsf{t}_1} \boldsymbol{Back}\ H \\
&\ \ldots \\
&\to_{\mathsf{t}_m} F_m \ \ldots \ (\boldsymbol{Back})F_m \to_{-\mathsf{t}_m} \boldsymbol{Back}\ H \\
&\to_{-\mathsf{s}} \boldsymbol{Back}\ G
\end{aligned}$$

By applying the inductive hypothesis on the m subtrees of the form $\to_{\mathsf{t}_i} \ldots \to_{-\mathsf{t}_i} \boldsymbol{Back}\ H$ (each having less than n inner nodes), sum total of $\emptyset$ is obtained. ♢

Lemma 24 (resumability). Let $\boldsymbol{S}$ be flat. For any queries A, B, C and any term or void P satisfying $\mathit{Vars}((P, B)) = \mathit{Vars}((A, \mathsf{S}, B))$ holds: If $A \to^{*}_{\mathsf{S}} B$ and $\texttt{true}(P), B \to^{+}_{\mathsf{T}} C$, then $A \to^{*}_{\mathsf{S}+\mathsf{T}} C$.

Proof. At the start of the second derivation, the variables of A, S, B (and no others) are spent, the same situation as at the end of the first derivation.

Since $\boldsymbol{S}$ is assumed to be flat, solely the spent variables determine standardizing apart. So the first derivation can be continued indistinguishably from the second one (after getting rid of $\texttt{true}(P)$). ♢

Theorem 26 (iteration for complete answer). Let $\boldsymbol{S}$ be flat. Let A, B, R be queries, and P be a term or void. Then:

1. If $P \Vdash A \setminus_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \Vdash \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$, then also $P \Vdash A, B \setminus_{\mathsf{S}+\mathsf{T}} R$.
2. Conversely, if $P \Vdash A, B \setminus_{\mathsf{U}} R$, then there are S, T such that $P \Vdash A \setminus_{\mathsf{S}} B, R$ and $P, A, \mathsf{S} \Vdash \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R)$ and $\mathsf{U} = \mathsf{S} + \mathsf{T}$.

Proof. By induction on the number k of conjuncts in A. We start with the base case $k = 1$, i.e. A is an atom.

Direct part By induction on the length n of derivation of A. For $n = 1$, A can be `true`, `true(_)` or a unification. The claim follows from the definition of resolvent. Now assume the claim holds for derivations of length $\leq n$ and consider a derivation for A of length $n + 1$. Assumptions are

$$P \Vdash A \quad \setminus_{\mathsf{S}} B, R \tag{A.3}$$
$$P, A, \mathsf{S} \Vdash \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R) \tag{A.4}$$

From Lemma 33 applied on (A.3), A had to have been resolved, so for some $C, \mathsf{V}, \mathsf{U}$ with $\mathsf{S} = \mathsf{V} + \mathsf{U}$ we have

$$P \Vdash A \rhd C \setminus_{\mathsf{V}} B, R \tag{A.5}$$
$$P, A, \mathsf{V} \Vdash C \setminus_{\mathsf{U}} \mathsf{V}(B), \mathsf{V}(R) \tag{A.6}$$

Then $\mathsf{S}(\_) = \mathsf{U}(\mathsf{V}(\_))$, so by (A.4) holds $P, A, \mathsf{V} + \mathsf{U} \Vdash \mathsf{U}(\mathsf{V}(B)) \setminus_{\mathsf{T}} \mathsf{U}(\mathsf{V}(R))$ and hence, by $\mathit{Vars}(C) \subseteq \mathit{Vars}((A, \mathsf{V}))$,

$$P, A, \mathsf{V}, C, \mathsf{U} \Vdash \mathsf{U}(\mathsf{V}(B)) \setminus_{\mathsf{T}} \mathsf{U}(\mathsf{V}(R)) \tag{A.7}$$

Applying the inductive hypothesis on (A.6) and (A.7), we obtain

$$P, A, \mathsf{V} \Vdash C, \mathsf{V}(B) \setminus_{\mathsf{U}+\mathsf{T}} \mathsf{V}(R)$$

This and (A.5) give, by Lemma 34, $P \Vdash A, B \setminus_{\mathsf{V}+\mathsf{U}+\mathsf{T}} R$ with $\mathsf{V} + \mathsf{U} + \mathsf{T} = \mathsf{S} + \mathsf{T}$, which proves the direct part.

Converse part By induction on the length n of derivation of A, B. For $n = 2$, A and B can be `true`, `true(_)` or a unification. The claim follows from the definition of resolvent. Now assume the claim holds for derivations for A, B of length $\leq n$ and consider a derivation of length $n + 1$.

The assumption is $P \Vdash A, B \setminus_{\mathsf{U}} R$. By Lemma 33, for some $C, \mathsf{W}, \mathsf{V}$

$$P \Vdash A \rhd C \setminus_{\mathsf{W}} B, R \tag{A.8}$$
$$P, A, \mathsf{W} \Vdash C, \mathsf{W}(B) \setminus_{\mathsf{V}} \mathsf{W}(R), \text{ and } \mathsf{U} = \mathsf{W} + \mathsf{V} \tag{A.9}$$

By induction hypothesis on (A.9), V can be split as $\mathsf{V} = \mathsf{V}_1 + \mathsf{V}_2$ such that

$$P, A, \mathsf{W} \Vdash C \quad \setminus_{\mathsf{V}_1} \mathsf{W}(B), \mathsf{W}(R) \tag{A.10}$$
$$P, A, \mathsf{W}, C, \mathsf{V}_1 \Vdash \mathsf{V}_1(\mathsf{W}(B)) \setminus_{\mathsf{V}_2} \mathsf{V}_1(\mathsf{W}(R)) \tag{A.11}$$

By Lemma 34 applied on (A.8) and (A.10),

$$P \Vdash A \setminus_{\mathsf{W}+\mathsf{V}_1} B, R$$

By (A.11) and $\mathit{Vars}(C) \subseteq \mathit{Vars}((A, \mathsf{W}))$, we obtain

$$P, A, \mathsf{W} + \mathsf{V}_1 \Vdash \mathsf{V}_1(\mathsf{W}(B)) \setminus_{\mathsf{V}_2} \mathsf{V}_1(\mathsf{W}(R))$$

So U can be split as $\mathsf{U} = \mathsf{W} + \mathsf{V}_1 + \mathsf{V}_2$ with required properties.

Base case is thus proved. Now assume the claim holds for conjunctions of up to k conjuncts. Let $A \coloneqq Q, Q'$ where Q is an atom and Q' is a conjunction of length k.

Inductive case, direct part The assumptions are

$$P \Vdash Q, Q' \setminus_{\mathsf{S}} B, R \tag{A.12}$$

$$P, Q, Q', \mathsf{S} \Vdash \mathsf{S}(B) \setminus_{\mathsf{T}} \mathsf{S}(R) \tag{A.13}$$

We wish to obtain $P \Vdash Q, Q', B \setminus_{\mathsf{S}+\mathsf{T}} R$. By applying the converse part of the base case on (A.12), there are U, V with $\mathsf{U} + \mathsf{V} = \mathsf{S}$ and

$$P \Vdash Q \quad \setminus_{\mathsf{U}} Q', B, R \tag{A.14}$$

$$P, Q, \mathsf{U} \Vdash \mathsf{U}(Q') \setminus_{\mathsf{V}} \mathsf{U}(B, R) \tag{A.15}$$

By Lemma 31, $\mathit{Vars}(Q') \cup \mathit{Vars}(\mathsf{U}) = \mathit{Vars}(\mathsf{U}(Q')) \cup \mathit{Vars}(\mathsf{U})$. Hence, (A.13) can be rearranged as

$$P, Q, \mathsf{U}, \mathsf{U}(Q'), \mathsf{V} \Vdash \mathsf{V}(\mathsf{U}(B)) \setminus_{\mathsf{T}} \mathsf{V}(\mathsf{U}(R)) \tag{A.16}$$

By induction hypothesis, from (A.15) and (A.16) follows

$$P, Q, \mathsf{U} \Vdash \mathsf{U}(Q'), \mathsf{U}(B) \setminus_{\mathsf{V}+\mathsf{T}} \mathsf{U}(R) \tag{A.17}$$

By applying the direct part of the base case on (A.14) and (A.17),

$$P \Vdash Q, Q', B \setminus_{\mathsf{U}+\mathsf{V}+\mathsf{T}} R$$

Inductive case, converse claim This can be proved similarly. ♢