Appendices for the Paper "Planning for an Efficient Implementation of Hypothetical Bousi∼Prolog"

Pascual Julián-Iranzo∗
Dept. of Information Technologies and Systems, University of Castilla-La Mancha, Spain
(*e-mail:* `Pascual.Julian@uclm.es`)

Fernando Sáenz-Pérez
Dept. of Software Engineering and Artificial Intelligence, Universidad Complutense de Madrid, Spain
(*e-mail:* `fernan@sip.ucm.es`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

∗ This work was supported by the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2019-104735RB-C42 (SAFER), by the Spanish Ministry of Economy and Competitiveness, under the grants TIN2016-76843-C4-2-R (MERINET), TIN2017-86217-R (CAVI-ART-2), and by the Comunidad de Madrid, under the grant S2018/TCS-4339 (BLOQUES-CM), co-funded by EIE Funds of the European Union.

Appendix A Performance comparison: meta-interpreted vs. interpreted

This section analyses the performance of the different solving alternatives explained above: the meta-interpreter and the compiler-based solver interpreter. They are also compared, when applicable, to the native implementation and the solving in SWI-Prolog of the program under test. Two alternatives are first described for the meta-interpreter, and then the performance analysis is given, comparing them to the compiled approach in Section 3. The Prolog interpreter for non-hypothetical programs is also included in the comparison as a reference for the overhead caused by the compiled approach.

A.1 Hypothetical Meta-interpreters

Figure A 1 illustrates the hypothetical propositional meta-interpreter, which has been enlarged to deal with disjunctive rules and built-in calls, where a fact is a rule with a *true* body. The predicate *builtin*/1 checks whether its argument represents a call to a built-in predicate (different from $\wedge$, $\vee$ and $\Rightarrow$).

$$solve((\phi_1 \wedge \phi_2),\Pi) \leftarrow solve(\phi_1,\Pi) \wedge solve(\phi_2,\Pi).$$
$$solve((\phi_1 \vee \phi_2),\Pi) \leftarrow solve(\phi_1,\Pi) \vee solve(\phi_2,\Pi).$$
$$solve(\phi,\Pi) \leftarrow builtin(\phi) \wedge call(\phi).$$
$$solve((R \Rightarrow \phi),\Pi) \leftarrow solve(\phi,[R|\Pi]).$$
$$solve(\phi,\Pi) \leftarrow member((\phi \leftarrow \phi'),\Pi) \wedge solve(\phi',\Pi).$$

Fig. A 1. Meta-interpreter for hypothetical propositional logic programs

The meta-interpreter for hypothetical propositional logic programs depicted in Figure A 1 is not applicable to predicate logic programs. For example, the goal $\leftarrow p$ for the program $\{p \leftarrow q(1) \wedge q(2), q(X)\}$ should succeed, but it does not because solving $\leftarrow q(1)$ creates the substitution $\{X/1\}$, which is not compatible with the second call $\leftarrow q(2)$. Nonetheless, it can be easily adapted to the non-propositional case by modifying the last clause to:

$$solve(\phi,\Pi) \leftarrow copy\_term(\Pi,\Pi') \wedge member((\phi \leftarrow \phi'),\Pi') \wedge solve(\phi',\Pi).$$

However, copying the entire program each time the unification of a rule or fact with the goal is sought, is hugely resource consuming. A more convenient approach is to look for a matching clause and copy only this clause, as follows:

$$solve(\phi,\Pi) \leftarrow unif\_member((\phi \leftarrow \_),\Pi,R) \wedge copy\_term(R,(\phi' \leftarrow \phi'')) \wedge \phi = \phi' \wedge solve(\phi'',\Pi).$$

where $unif\_member(X,L,Y)$ stands for: Y is a member of L that is unifiable with X.

This can be slightly enhanced by using an ordered list for the program predicates (though preserving rule user ordering in each predicate), therefore reducing the serial access time complexity by a factor of 2, on average. Also, adding cuts for selecting the appropriate meta-interpreter clause will prune choice points in advance, and will also save some tests. Thus, an actual implementation in Prolog could be:

```
solve((Goal1, Goal2), Program) :-
  !, solve(Goal1, Program), solve(Goal2, Program).
solve((Goal1; Goal2), Program) :-
  !, (solve(Goal1, Program) ; solve(Goal2, Program)).
solve((Hyp => Goal), Program) :-
  !, insert_rule(Hyp, Program, NProgram), solve(Goal, NProgram).
solve(Goal, _Program) :-
  builtin(Goal), !, Goal.
solve(Goal, Program) :-
  unif_member((Goal :- _Body), Program, (UGoal :- UBody)),
  copy_term((UGoal :- UBody), (CGoal :- CBody)),
  CGoal=Goal, solve(CBody, Program).
```

Note that, in particular, the last clause will not be selected uselessly, as opposed to the skeleton shown in Figure A 1. The predicate `insert_rule`/3 inserts a rule in the place corresponding to its ordering.

As suggested by an anonymous referee, there are more alternatives that may be considered. In particular, this meta-interpreter can be further enhanced by passing only assumed rules to `solve`/2, instead of the whole program. Then, the static user program can be accessed by the built-in predicate `clause`/2. Moreover, the assumed rules can be better represented by a tree, whose nodes are predicates (key of the tree), with their rules in a list, thereby improving access when many predicates are assumed. To implement this approach, the last clause of the former meta-interpreter is replaced by:

```
% Lookup in the consulted (static) program:
solve(Goal, Program) :-
  clause(Goal, Body), solve(Body, Program).
% Lookup in the augmented program:
solve(Goal, Program) :-
  functor(Goal, Functor, Arity), atomic_concat(Functor, Arity, Key),
  get_assoc(Key, Program, Rules),
  unif_member((Goal :- _Body), Rules, (UGoal :- UBody)),
  copy_term((UGoal :- UBody), (CGoal :- CBody)),
  CGoal=Goal, solve(CBody, Program).
```

Association lists are implemented with AVL trees, with $\mathcal{O}(\log n)$ worst-case (and expected) time operations, where n denotes the number of elements in the association list (Wielemaker et al. 2012). This meta-interpreter uses `get_assoc`/3 in `solve`/2 to retrieve nodes in the tree and `put_assoc`/4 in `insert_rule`/3 (cf. its implementation in `meta2.pl` at the URL mentioned in the next subsection).

A.2 Performance Analysis

All experiments were run on an Intel Xeon CPU E3-1505M v5 with 4 physical cores at 2.8 GHz, 16GiB RAM, with the Windows 10 64-bit operating system. Benchmarks are run on the last stable version of SWI-Prolog 64-bit 8.2.4-1 at the time of writing this. Times in the Tables are given in seconds and are the result of averaging 10 runs (and discarding the first), and individual times were measured with the built-in predicate `statistics/2`. We have collected the time for the total run-time measurement (key cputime for SWI-Prolog), which returns the (user) CPU time, and the run-time (key runtime) which returns the total run-time, eliding the time for memory management (garbage collection) and system calls. Tables include the column CPUtime (for the total run-time), and Diftime (for the total run-time minus the run-time, primarily to reflect the cost of garbage collection). Garbage collection and stack trimming are carried out before each trial is run; then the time measurement can start. On completion of the trial, these operations are performed again before taking the elapsed time, in order to account for housekeeping tasks due to running the tests. Also, `inferences` is another measurement from the statistics predicate, which indicates the number of passes via the call and redo ports in order to execute a goal. All benchmarks and systems can be downloaded from `http://www.fdi.ucm.es/profesor/fernan/iclp2021/Experiments.zip`.

Several classical benchmark programs have been selected (they can be found at the SWI-Prolog site), where some have been adapted to remove cuts. These programs are intended, firstly, to make clear the price to be paid for including hypothetical reasoning in the system; and, secondly, to compare meta-interpreted alternatives with respect to the compiled alternative. In any case, it should be recalled that a compiled approach is preferred for the implementation of a system such as BPL, because assuming a rule implies a recompilation. Furthermore, most of those programs have been adapted to build hypothetical versions by assuming either facts or a rule for the data generator. These versions add a preceding *h* before the classical test name. With respect to the factorial test program, *facttr* is the tail recursive version, and all benchmark sizes can be consulted at the URL above. As a stress test, several parametric hypothetical programs are included, to test embedded implications: The first (labelled as *hypo1* in the following Tables): $\{p \leftarrow a_1 \Rightarrow a_2 \Rightarrow \ldots \Rightarrow a_n \Rightarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n\}$ with a goal $\leftarrow p$ for $n = 2000$; the second (*hypo2*): $\{p(0) \leftarrow a, p(N) \leftarrow N > 0 \wedge N_1 \ is \ N-1 \wedge (a \Rightarrow p(N_1))\}$ with a goal $\leftarrow p(3000)$ and requesting all solutions; and the third (*hypo3*): $\{p \leftarrow a \Rightarrow \ldots \Rightarrow a \Rightarrow a\}$ with a goal $\leftarrow p$ for 3000 assumptions, also requesting all solutions. The first program is intended to test the system by assuming a large batch of different predicates iteratively. The second recursively assumes facts of the same predicate and is intended to analyse the performance in the presence of backtracking by requesting all solutions. The third is similar to the second but iteratively adding those facts with nested assumptions.

Table A 1 collects the running measurements for classical tests and Table A 2 for hypothetical tests. Rows in the Table include the following labels: Meta1 for the first meta-interpreter as shown in this appendix, Meta2 for the second, improved, instance, Comp for the compiled approach as described in Definition 3.5, and Prolog for the native execution of (classical, non-hypothetical) tests in the Prolog system.

For classical tests (Table A 1) Meta2 performs better than Meta1 with significant speed-ups, and there is even a notable speed-up in the case of *path*, where recursively traversing the list of the program, including many arcs in the graph, takes a lot of effort that is avoided with Meta2, because the program is consulted. Looking now at the compiled alternatives Comp and Prolog, they perform better than the meta-interpreted versions. Comparing Comp to Prolog, they behave similarly in most cases other than in *nrev* and *queens*, where Prolog is faster ($4.62\times$ and $1.44\times$, respectively).

For hypothetical versions of classical tests (Table A 2, roughly similar conclusions can be drawn. Performance of Comp is almost always better than Meta2, and the latter is better than Meta1. Only in *hpath* is Meta2 faster by a small amount. Results can be different for other nonclassical stress tests such as *hypo1*, for which Meta2 performs better. In this case, note that AVL trees play an important role in fast accessing of each rule (in *hypo1* there is one assumed rule for each predicate, with a total of 2000). In turn, Comp takes more time, both in total time and memory management, even when the number of inferences is roughly a fifth compared to Meta1, but the time taken by memory management is noticeable compared to the other two alternatives. This system Comp performs better for *hypo2* with a similar inference ratio with respect to Meta2. With respect to the last test *hypo3*, Meta2 is the slowest, while there is a small difference between Comp and the fastest, Meta1.

Program	System	CPUtime	Diftime	Inferences
deriv	Meta1	1.989	0.245	10,230,177
	Meta2	1.205	0.128	4,070,102
	Comp	0.491	0.030	660,078
	Prolog	0.491	0.023	660,077
fact	Meta1	2.470	0.497	11,005,602
	Meta2	1.194	0.123	4,070,102
	Comp	0.848	0.253	1,500,077
	Prolog	0.888	0.231	1,500,076
facttr	Meta1	2.636	0.648	11,016,606
	Meta2	1.234	0.123	4,070,102
	Comp	0.414	0.036	2,001,077
	Prolog	0.450	0.056	2,001,076
fib	Meta1	1.156	0.439	4,827,118
	Meta2	0.859	0.345	4,006,022
	Comp	0.109	0.006	485,645
	Prolog	0.111	0.002	485,644
nrev	Meta1	2.258	0.575	5,699,379
	Meta2	1.194	0.127	4,070,102
	Comp	0.527	0.092	1,130,321
	Prolog	0.114	0.017	1,130,320
path	Meta1	572.533	0.428	5,414,424,890
	Meta2	1.197	0.130	4,070,102
	Comp	0.152	0.014	1,799,669
	Prolog	0.148	0.007	1,799,668
primes	Meta1	4.503	0.859	29,145,106
	Meta2	1.220	0.116	4,070,102
	Comp	0.409	0.023	1,873,577
	Prolog	0.408	0.016	1,873,576
qsort	Meta1	2.083	0.439	10,075,127
	Meta2	1.167	0.119	4,070,506
	Comp	0.287	0.016	1,005,071
	Prolog	0.256	0.008	1,005,070
queens	Meta1	6.231	0.000	47,984,882
	Meta2	1.173	0.116	4,070,102
	Comp	0.417	0.000	1,971,077
	Prolog	0.289	0.000	1,971,076

Table A 1. *Comparing Meta1, Meta2, Comp and Prolog for classical programs*

Program	System	CPUtime	Diftime	Inferences
	Meta1	2.084	0.175	11,430,623
hderiv	Meta2	1.181	0.117	4,070,102
	Comp	0.303	0.020	660,124
	Meta1	2.706	0.512	12,034,102
hfact	Meta2	1.211	0.111	4,070,102
	Comp	0.703	0.077	3,012,069
	Meta1	3.469	0.328	12,058,106
hfacttr	Meta2	1.216	0.130	4,070,102
	Comp	0.670	0.044	3,514,569
	Meta1	2.742	0.386	5,699,425
hnrev	Meta2	1.164	0.119	4,070,102
	Comp	0.605	0.098	1,130,333
	Meta1	9.698	0.089	76,637,002
hpath	Meta2	1.189	0.120	4,070,102
	Comp	1.330	0.102	4,537,562
	Meta1	4.694	0.277	30,589,606
hprimes	Meta2	1.241	0.119	4,070,102
	Comp	0.892	0.030	5,264,319
	Meta1	2.020	0.294	10,075,159
hqsort	Meta2	1.172	0.116	4,070,102
	Comp	0.272	0.014	1,005,083
	Meta1	6.039	0.000	51,927,280
hqueens	Meta2	1.139	0.116	4,070,102
	Comp	0.289	0.000	1,971,081
	Meta1	0.700	0.017	9,090,603
hypo1	Meta2	0.042	0.004	166,743
	Comp	0.939	0.162	2,023,076
	Meta1	2.045	0.081	31,618,620
hypo2	Meta2	1.498	0.128	22,657,550
	Comp	1.014	0.002	4,534,580
	Meta1	1.477	0.058	22,543,610
hypo3	Meta2	1.828	0.177	22,573,604
	Comp	1.611	0.077	4,531,578

Table A 2. *Comparing Meta1, Meta2 and Comp for hypothetical programs*

Appendix B Proof for Proposition 3.6

We divide the proof of Proposition 3.6 into two parts, starting with the proof of the statement "if there exists a derivation $\mathcal{D} \equiv (\langle (\leftarrow \mathcal{Q}), id, \Pi\rangle \Rightarrow^*_{\text{HSLD}} \langle \Box, \sigma, \Pi\rangle)$ then there exists a derivation $\mathcal{D}' \equiv (\langle \leftarrow \mathcal{Q}', id, \Pi'\rangle \Rightarrow^*_{\text{SLD}} \langle \Box, \sigma', \Pi''\rangle)$". This direction constitutes a kind of completeness result where we prove that derivations in the original program using the HSLD resolution rule can be reproduced by the SLD operational mechanism in the transformed program. In the second part, we prove the converse of the first statement, that is "there exists a derivation $\mathcal{D} : \langle (\leftarrow \mathcal{Q}), \theta, \Pi\rangle \Rightarrow^*_{\text{HSLD}} \langle \Box, \sigma, \Pi\rangle$ if there exists a derivation $\mathcal{D}' : \langle \leftarrow \mathcal{Q}', \theta, \Pi'\rangle \Rightarrow^*_{\text{SLD}} \langle \Box, \sigma', \Pi''\rangle$", which guarantees that the present implementation does not compute answers which are not computed by the HSLD semantics, leading to a sort of soundness result.

B.1 Proof of Part I

Lemma Appendix B.1
Let Π be a program and $\mathcal{G} \equiv (\leftarrow A \wedge \mathcal{Q}_1)$ a goal, if there exists the step

$$\mathcal{S} \equiv \langle (\leftarrow A \wedge \mathcal{Q}_1), \theta, \Pi\rangle \Rightarrow_{\text{HSLD}} \langle (\leftarrow \mathcal{Q}_2\sigma), \theta\sigma, \Pi\rangle$$

then there exists the following derivation in the translated program Π':

$$\langle (\leftarrow A' \wedge \mathcal{Q}'_1), \theta, \Pi'\rangle \Rightarrow^+_{\text{SLD}} \langle (\leftarrow \mathcal{Q}'_2(\sigma \cup \delta)), \theta\sigma \cup \delta, \Pi' \cup \Pi_{reg}\rangle,$$

where A', $\mathcal{Q}'_i$ are the goal translations of A, $\mathcal{Q}_i$ respectively, and Π_{reg} is the set of all the *reg*/3 assertions due to embedded implication solving. The domain of the substitution δ shares variables with neither θ nor σ, and $\theta\sigma = \theta(\sigma \cup \delta)[\mathcal{V}ar(\mathcal{G})]$. ■

Proof
We proceed by induction on the context identifier s associated with the program context.

1. Base case($s = \varepsilon$): In this case query A must be an atom, that is, $A \equiv p(\overline{s_n})$ and there must be a rule $(p(\overline{t_n}) \leftarrow \mathcal{B}) \in \Pi$ for which $\sigma = mgu(\{p(\overline{s_n}) = p(\overline{t_n})\}) = mgu(\{\overline{s_n = t_n}\})$ (where $s_i = t_i$, with $1 \leq i \leq n$, are unification problems) and step $\mathcal{S}$ is

$$(\langle \leftarrow p(\overline{s_n}) \wedge \mathcal{Q}_1, \theta, \Pi\rangle \Rightarrow_{\text{HSLD}} \langle \leftarrow (\mathcal{B} \wedge \mathcal{Q}_1)\sigma, \theta\sigma, \Pi\rangle)$$

   According to Definition 3.5, the rule translation of $(p(\overline{t_n}) \leftarrow \mathcal{B}) \in \Pi$ is $p(\overline{t_n}, [], 0, \varepsilon, S^C) \leftarrow \mathcal{B}'$ and the goal translation of $\leftarrow p(\overline{s_n})$ is $\leftarrow p(\overline{s_n}, L, I, C, S)$ where S^C, L, I, C and S are fresh variables. It is then easy to verify that:

$$\begin{aligned} & mgu(\{p(\overline{s_n}, L, I, C, S) = p(\overline{t_n}, [], 0, \varepsilon, S^C)\}) \\ & \quad = mgu(\{\overline{s_n = t_n}, L = [], I = 0, C = \varepsilon, S = S^C\}) \\ & \quad = \sigma\{L/[], I/0, C/\varepsilon, S/S^C\} = \sigma \cup \delta \end{aligned}$$

   where $\delta = \{L/[], I/0, C/\varepsilon, S/S^C\}$, and its domain does not share variables with σ. Hence, the following step is possible in the translated program Π':

$$\langle \leftarrow p(\overline{s_n}, L, I, C, \varepsilon) \wedge \mathcal{Q}'_1, \theta, \Pi'\rangle \Rightarrow_{\text{SLD}} \langle \leftarrow (\mathcal{B}' \wedge \mathcal{Q}'_1)(\sigma \cup \delta), \theta(\sigma \cup \delta), \Pi'\rangle$$

   and $\theta\sigma = \theta(\sigma \cup \delta)[\mathcal{V}ar(\mathcal{G})]$.

2. Inductive case ($s > \varepsilon$): The query $A \equiv (H \Rightarrow G)$, that is, an embedded implication (chain), and the step

$$\mathscr{S} \equiv \langle (\leftarrow (H \Rightarrow G) \wedge \mathcal{Q}_1), \theta, \Pi \rangle \Rightarrow_{\text{HSLD}} \langle (\leftarrow \mathcal{Q}_1\sigma), \theta\sigma, \Pi \rangle$$

Therefore, Rule 2 of Definition 3.1 was applied and the derivation

$$\mathcal{D}_1 \equiv (\langle (\leftarrow G), id, \Pi_1 \rangle \Rightarrow^*_{\text{HSLD}} \langle \square, \sigma, \Pi_1 \rangle)$$

must exist, where $\Pi_1 = \Pi \cup \{H\}$ is a new program with context identifier $1 > \varepsilon$. Then, by the Inductive hypothesis, the following derivation in the translated program Π':

$$\mathcal{D}'_1 \equiv (\langle (\leftarrow G'), id, \Pi' \rangle \Rightarrow^*_{\text{SLD}} \langle \square, \sigma \cup \delta, \Pi' \cup \Pi_{reg} \rangle)$$

must exist, where G' is the goal translation of G, Π_{reg} is the set of rule registrations and δ is a substitution for which its domain shares variables with neither the original (not translated) goal G nor the substitution σ.
On the other hand, note that the translation of the goal $(H \Rightarrow G)$ is $\Rightarrow (0, [\overline{X}], G', I^C, S^C)$ plus the rule translation H' of H.
Now, using Definition 3.4 for solving embedded implication clauses and Definition 3.3 of rule registration, and the derivation $\mathcal{D}'_1$, it is possible to build the following derivation $\mathcal{D}'$:

$$\begin{array}{l}
\langle \leftarrow (\Rightarrow (0, [\overline{X}], G', I^C, S^C) \wedge \mathcal{Q}'_1), \theta, \Pi' \rangle \\
\Rightarrow_{\text{SLD}} \langle \leftarrow (get\_ci(I^C) \wedge reg\_rule(0, [\overline{X}], I^C, S^C) \wedge call(G') \wedge \mathcal{Q}'_1), \theta, \Pi' \rangle \\
\Rightarrow_{\text{SLD}} \langle \leftarrow (reg\_rule(0, [\overline{X}], 1, S^C) \wedge call(G') \wedge \mathcal{Q}'_1)\{I^C/1\}, \theta \cup \{I^C/1\}, \Pi' \rangle \\
\Rightarrow_{\text{SLD}} \langle \leftarrow (assertz(reg(0, [\overline{X}], S^C)) \wedge call(G') \wedge \mathcal{Q}'_1)\{I^C/1\}, \theta \cup \{I^C/1\}, \Pi' \rangle \\
\Rightarrow_{\text{SLD}} \langle \leftarrow (call(G') \wedge \mathcal{Q}'_1)\{I^C/1\}, \theta \cup \{I^C/1\}, \Pi' \cup \{reg(0, [\overline{X}], S^C)\} \rangle \\
\Rightarrow_{\text{SLD}} \langle \leftarrow (G' \wedge \mathcal{Q}'_1)\{I^C/1\}, \theta \cup \{I^C/1\}, \Pi' \cup \{reg(0, [\overline{X}], S^C)\} \rangle \\
\Rightarrow^*_{\text{SLD}} \langle \leftarrow Q'_1(\sigma \cup \{I^C/1\} \cup \delta), (\theta\sigma) \cup \{I^C/1\} \cup \delta, \Pi' \cup \{reg(0, [\overline{X}], S^C)\} \cup \Pi_{reg} \rangle
\end{array}$$

where the domain of $\{I^C/1\} \cup \delta$ shares variables with neither θ nor σ, and $\theta\sigma = (\theta\sigma \cup \{I^C/1\} \cup \delta)[\mathcal{V}ar(\mathscr{G})]$.

$\square$

Proposition Appendix B.2
For a program Π and a goal $\leftarrow \mathcal{Q}$, if there exists a derivation $\mathcal{D} \equiv \langle (\leftarrow \mathcal{Q}), id, \Pi \rangle \Rightarrow^*_{\text{HSLD}} \langle \square, \sigma, \Pi \rangle$ then there exists a derivation $\mathcal{D}' \equiv \langle \leftarrow \mathcal{Q}', id, \Pi' \rangle \Rightarrow^*_{\text{SLD}} \langle \square, \sigma', \Pi' \cup \Pi_{reg} \rangle$, where Π' is the translation of each rule in Π, $\mathcal{Q}'$ is the goal translation of $\mathcal{Q}$, Π_{reg} is the set of rule registrations, that is, all the *reg*/3 assertions due to embedded implication solving, and $\sigma = \sigma'[\mathcal{V}ar(\mathscr{G})]$. ■

Proof
By induction on the length of the derivation $\mathcal{D}$ and Lemma Appendix B.1. $\square$

As mentioned above, Proposition Appendix B.2 constitutes a kind of completeness result. In the following we concentrate on the other direction, which leads to a sort of soundness result.

B.2 Proof of Part II

Proposition Appendix B.3

Let Π' be the translation of a program Π and $\mathscr{G}' \equiv (\leftarrow \mathcal{Q}'_1)$ the goal translation of a goal $\mathscr{G} \equiv (\leftarrow \mathcal{Q}_1)$, if there exists a derivation $\mathcal{D}' \equiv \langle \leftarrow \mathcal{Q}', \theta \cup \delta, \Pi' \rangle \Rightarrow^*_{\text{SLD}} \langle \Box, (\theta\sigma) \cup \delta', \Pi' \cup \Pi_{reg} \rangle$, where Π_{reg} is the set of rule registrations, that is, all the *reg*/3 assertions due to embedded implication solving, then there exists a derivation $\mathcal{D} \equiv \langle (\leftarrow \mathcal{Q}), \theta, \Pi \rangle \Rightarrow^*_{\text{HSLD}} \langle \Box, \theta\sigma, \Pi \rangle$. The domains of the substitutions δ and δ' share variables with neither θ nor σ, and $\theta\sigma = (\theta\sigma) \cup \delta'[\mathcal{V}ar(\mathscr{G})]$. ■

Proof

The proof proceeds by induction on the length of the derivation $\mathcal{D}'$. Without loss of generality, thanks to the independence of the computation rule in the SLD operational mechanism, several steps in the derivation $\mathcal{D}'$ can be conveniently ordered. So, it is possible to group fragments of the derivation $\mathcal{D}'$ in the translated program Π', which correspond with the steps in the derivation $\mathcal{D}$, in the program Π.

1. Base case($n = 1$): In this case the query $\mathcal{Q}' \equiv p(\overline{s_n}, [], I, C, \varepsilon)$ (translation of $\mathcal{Q} \equiv p(\overline{s_n})$, since an initially launched goal is solved in the initial context ε and its list of shared variables is empty) and there must be a rule $p(\overline{t_n}, [], i, \varepsilon, S^C)$, with rule index i (translation of the fact $p(\overline{t_n}) \in \Pi$), for which

$$\begin{aligned} & mgu(\{p(\overline{s_n}, [], I, C, \varepsilon) = p(\overline{t_n}, [], i, \varepsilon, S^C)\}) \\ & \quad = mgu(\{\overline{s_n = t_n}, [] = [], I = i, C = \varepsilon, \varepsilon = S^C\}) \\ & \quad = \sigma\{I/i, C/\varepsilon, S^C/\varepsilon\} = \sigma \cup \delta_1 \end{aligned}$$

   where $mgu(\{\overline{s_n = t_n}\}) = \sigma$ must be solvable, $\delta_1 = \{I/i, C/\varepsilon, S^C/\varepsilon\}$, and its domain shares variables with neither θ nor σ. Therefore, the following one-step derivation $\mathcal{D}$ is possible in the program Π: $\langle \leftarrow p(\overline{s_n}), \theta, \Pi \rangle \Rightarrow_{\text{HSLD}} \langle \Box, \theta\sigma, \Pi \rangle$.
2. Inductive case ($n > 1$): In the analysis of the inductive case we consider two possibilities, depending on whether the first step is performed with a sub-goal, which is an instance of the atom $\Rightarrow (I^H, [X'], G', I^C, S^C)$, or not. This kind of sub-goal comes from embedded implications that appear in the body of some rule, or are submitted directly in the initial query proposed to the system.
   (a) First, consider that $\mathcal{Q}' \equiv \leftarrow p(\overline{s_n}, [], I, C, epsilon) \wedge \mathcal{Q}'_1$ (translation of $\mathcal{Q} \equiv \leftarrow p(\overline{s_n}) \wedge \mathcal{Q}_1$) and the first step is performed with a rule $(p(\overline{t_n}, [], i, \varepsilon, S^C) \leftarrow \mathcal{B}') \in \Pi'$, with index rule i (translation of $(p(\overline{t_n}) \leftarrow \mathcal{B}) \in \Pi$),[1] whose head unifies with the selected sub-goal:

$$\begin{aligned} & mgu(\{p(\overline{s_n}, [], I, C, epsilon) = p(\overline{t_n}, [], i, \varepsilon, S^C)\}) \\ & \quad = mgu(\{\overline{s_n = t_n}, [] = [], I = i, C = \varepsilon, \varepsilon = S^C\}) \\ & \quad = \sigma\{I/i, C/\varepsilon, S^C/\varepsilon\} = \sigma_1 \cup \delta_1 \end{aligned}$$

   where $mgu(\{\overline{s_n = t_n}\}) = \sigma_1$ must be solvable, $\delta_1 = \{I/i, C/\varepsilon, S^C/\varepsilon\}$ and its domain

[1] Note that in the translated program Π', we can find rules of the form $p(\overline{t_n}, [\overline{X'}], I^H, S^H, S^C) \leftarrow reg(I^H, [\overline{X'}], C^R) \wedge chk(C^R, S^C) \wedge \mathcal{B}'$ coming from the translation of the embedded implications in the body of the rules in Π (see Definition 3.5). However, this type of rule does not contribute to the first step of a derivation.

shares variables neither with θ nor with σ. Hence, derivation $\mathcal{D}'$ proceeds thus:

$$\begin{aligned}&\langle\leftarrow p(\overline{s_n}) \wedge \mathcal{Q}'_1, \theta \cup \delta, \Pi'\rangle\\ &\quad\Rightarrow_{\text{SLD}} \langle\leftarrow (\mathcal{B}' \wedge \mathcal{Q}'_1)(\sigma_1 \cup \delta_1), (\theta \cup \delta)(\sigma_1 \cup \delta_1), \Pi'\rangle\\ &\quad\overset{*}{\Rightarrow}_{\text{SLD}} \langle\Box, (\theta \cup \delta)(\sigma_1 \cup \delta_1)(\sigma_2 \cup \delta_2), \Pi' \cup \Pi_{reg}\rangle\end{aligned}$$

Now, because $mgu(\{p(\overline{s_n}) = p(\overline{t_n})\}) = mgu(\{\overline{s_n = t_n}\}) = \sigma_1$ is solvable, there exists the step in Π:

$$\langle\leftarrow p(\overline{s_n}) \wedge \mathcal{Q}_1, \theta, \Pi\rangle \Rightarrow_{\text{HSLD}} \langle\leftarrow (\mathcal{B}' \wedge \mathcal{Q}'_1)\sigma_1, \theta\sigma_1, \Pi\rangle$$

on the other hand, by the inductive hypothesis, there exists a derivation in Π such that:

$$\langle\leftarrow (\mathcal{B} \wedge \mathcal{Q}_1)\sigma_1, \theta\sigma_1, \Pi\rangle \overset{*}{\Rightarrow}_{\text{HSLD}} \langle\Box, \theta\sigma_1\sigma_2, \Pi\rangle$$

and the derivation $\mathcal{D}$ in Π can be constructed.

(b) In this case, the first step is performed on a solving implication clause, that is $\mathcal{Q}' \equiv\leftarrow (\Rightarrow (i, [\overline{X}], G', I^C, S^C) \wedge \mathcal{Q}'_1)$ which is the translation of $\mathcal{Q} \equiv\leftarrow (H \Rightarrow G) \wedge \mathcal{Q}_1$. For simplicity, we will assume that $G' \equiv q(\overline{s_n})$ and the goal translation $G \equiv q(\overline{s_n}, L, I, C, S^C.I^C)$. Then the shape of the derivation $\mathcal{D}'$ is:

$$\begin{aligned}&\langle\leftarrow (\Rightarrow (i, [\overline{X}], G', I^C, S^C) \wedge \mathcal{Q}'_1), \theta, \Pi'\rangle\\ &\Rightarrow_{\text{SLD}} \langle\leftarrow (get\_ci(I^C) \wedge reg\_rule(i, [\overline{X}], I^C, S^C) \wedge call(G') \wedge \mathcal{Q}'_1), \theta, \Pi'\rangle\\ &\Rightarrow_{\text{SLD}} \langle\leftarrow (reg\_rule(i, [\overline{X}], j, S^C) \wedge call(G') \wedge \mathcal{Q}'_1)\{I^C/j\}, \theta \cup \{I^C/j\}, \Pi'\rangle\\ &\Rightarrow_{\text{SLD}} \langle\leftarrow (assertz(reg(i, [\overline{X}], S^C)) \wedge call(G') \wedge \mathcal{Q}'_1)\{I^C/j\}, \theta \cup \{I^C/j\}, \Pi'\rangle\\ &\Rightarrow_{\text{SLD}} \langle\leftarrow (call(G') \wedge \mathcal{Q}'_1)\{I^C/j\}, \theta \cup \{I^C/j\}, \Pi' \cup \{reg(i, [\overline{X}], S^C)\}\rangle\\ &\Rightarrow_{\text{SLD}} \langle\leftarrow (G' \wedge \mathcal{Q}'_1)\{I^C/1\}, \theta \cup \{I^C/1\}, \Pi' \cup \{reg(0, [\overline{X}], S^C)\}\rangle\\ &\overset{+}{\Rightarrow}_{\text{SLD}} \langle\leftarrow Q'_1(\sigma_1 \cup \{I^C/1\} \cup \delta_1), (\theta\sigma_1) \cup \{I^C/j\} \cup \delta_1, \Pi' \cup \{reg(i, [\overline{X}], S^C)\} \cup \Pi_{reg1}\rangle\\ &\overset{*}{\Rightarrow}_{\text{SLD}} \langle\Box, (\theta\sigma_1\sigma_2) \cup \{I^C/j\} \cup \delta_1 \cup \delta_2, \Pi' \cup \{reg(i, [\overline{X}], S^C)\} \cup \Pi_{reg1} \cup \Pi_{reg2}\rangle\end{aligned}$$

with two clear parts. The first part corresponds to the HSLD step performed on $(H \Rightarrow G)$ using Rule 2 of Definition 3.1. It groups the associated derivations submitted by the occurrence of embedded implications and their successive program contexts. Then there must exist the HSLD step $\langle\leftarrow (H \Rightarrow G) \wedge \mathcal{Q}_1), \theta, \Pi\rangle \Rightarrow_{\text{HSLD}} \langle Q_1\sigma_1, \theta\sigma_1, \Pi\rangle$ in the program Π. As for the second part, by the inductive hypothesis, there must exist the HSLD derivation $\langle Q_1\sigma_1, \theta\sigma_1, \Pi\rangle \overset{*}{\Rightarrow}_{\text{SLD}} \langle\Box, \theta\sigma_1\sigma_2, \Pi\rangle$. Combining both, the former step and the last derivation, we obtain the derivation $\mathcal{D}$.

$\Box$

Corollary Appendix B.4

Let Π' be the translation of a program Π and $\mathscr{G}' \equiv (\leftarrow \mathcal{Q}'_1)$ the goal translation of a goal $\mathscr{G} \equiv (\leftarrow \mathcal{Q}_1)$, if there exists a derivation $\mathcal{D}' \equiv \langle\leftarrow \mathcal{Q}', id, \Pi'\rangle \overset{*}{\Rightarrow}_{\text{SLD}} \langle\Box, \sigma', \Pi' \cup \Pi_{reg}\rangle$, where Π_{reg} is the set of rule registrations, that is, all the *reg*/3 assertions due to embedded implication solving, then there exists a derivation $\mathcal{D} \equiv \langle(\leftarrow \mathcal{Q}), id, \Pi\rangle \overset{*}{\Rightarrow}_{\text{HSLD}} \langle\Box, \sigma, \Pi\rangle$, and $\sigma = \sigma'[\mathcal{V}ar(\mathscr{G})]$. ■

Appendix C Related Work

The term "hypothetical reasoning" appears in many important contexts in the philosophical and scientific literature. Auguste Comte, founder of Positivism, is one of the first thinkers to highlight the importance of hypotheses in science (Bourdeau 2020). Although Comte does not establish laws for hypothetical reasoning, he begins the path, influencing (according to Bourdeau (2014)) other thinkers such as Peirce and its abductive reasoning. According to that American philosopher, human thought has three modes of reasoning: deductive, inductive and abductive. "Abduction is the first step of scientific reasoning," because, as he says, "abduction is the process of forming explanatory hypotheses. It is the only logical operation that introduces a new idea" (Douven 2021).

However, our work is centered in a more specific area with a long tradition in the field of Logic Programming, in which the purpose is prospective: to propose hypotheses in order to evaluate its consequences. Mainly, our work is influenced by those of Gabbay and Reyle (1984), Gabbay (1985) and Bonner (1988; 1990; 1994; 1997) and, to a lesser extent, by those of L.T. McCarty (1988a; 1988b).

Gabbay first deals with hypothetical implications in logic programming. In Gabbay and Reyle (1984), they focused only on addition operations because deletion is problematic; thus they let it for another paper. Addition is essentially monotonic and deletion is not. We use a similar technique to the one followed by Gabbay for implementing hypothetical implications, by asserting the antecedent to the program database, trying to derive the consequent and finally retracting the antecedent. Gabbay (1985) investigates the logical properties of N-PROLOG and the way it relates to classical logic and the classical quantifiers. He also introduced negation as failure into N-PROLOG. He saw that success in the N-PROLOG computation of a goal G from the database P means logically that $P \vdash G$ in intuitionistic logic. It is credited that was Gabbay the first one to realize the important connection between hypothetical reasoning and intuitionistic logic (Bonner 1994).

McCarty (1988a) presents a clausal language that extends Horn-clause logic by adding negations and embedded implications (i.e., hypothetical implications –he was the one who first used this designation–) to the right-hand side of a rule, and interpreting these new rules intuitionistically in a set of partial models. Lately, McCarty (1988b) shown that clausal intuitionistic logic has a tableau proof procedure that generalizes Horn-clause refutation proofs and it is proved sound and complete.

As it has been said, Bonner has extensive experience in this field, starting from a language with embedded implications (close to ours) and exploring its applications and formal properties, including results on complexity (Bonner 1988; Bonner 1990; Bonner 1994). In his latest work on this topic (Bonner 1997), he broke with his initial works and he developed a logic programming language with a dedicated syntax in which users can create hypotheses and draw inferences from them. He provides two specific operations with a modal-like notation: hypothetical insertion of facts into a database ($Q[add : A]$ meaning that "Q would be true if A were added to the database"), that has a well-established logic (intuitionistic logic) and hypothetical deletion ($Q[del : A]$ meaning that "Q would be true if A were deleted from the database"). In this paper, he develop a logical semantics for hypothetical insertions and deletions (including a proof theory, model theory, and fixpoint theory). He analyses the expressibility of the language and he shows that classical logic cannot express some simple hypothetical queries. However, we believe that the language introduced, with specific insertion and deletion operations, may have limitations compared to the one we have proposed in this work (e.g., only atoms can be inserted or deleted). Finally, he augmented the logic with negation-as-failure so that nonmonotonic queries can be expressed, a subject that we let as future work.

Another piece of related work is λ-Prolog, which uses a syntax based on the so-called "hereditary Harrop formulas." Thanks to this type of formulas, λ-Prolog subsumes a set of increasingly complex sublanguages, ranging from Horn clauses and higher-order Horn clauses to hereditary Harrop formulas. This type of formula, for example, allows rules with bodies that contain (hypothetical) implications whose hypotheses are in turn rules.

The use of higher-order Horn clauses and a non-deterministic goal-directed search-based operational semantics (which is complete with respect to an intuitionistic sequent calculus as stated by Miller et al. (1991)) allows λ-Prolog the ability to perform hypothetical reasoning. In practical and informal terms, the λ-Prolog operational mechanism performs operations similar to those performed by Comp to solve a hypothetical implication ($H \Rightarrow G$): Assert H to the rules of the program (creating a new context) and launch the goal G; if G is successful, then ($H \Rightarrow G$) is also successful. It is nothing other than the rule "AUGMENT" ($P \vdash (D \Rightarrow G)$ only if $P + D \vdash G$.), one of the operational rules that models the computation-as-goal-directed-search of λ-Prolog.

Therefore, both mechanisms are comparable, except that λ-Prolog fits into a more general and ambitious framework, while Comp simply tries to extend the language of Horn clauses and the resolution principle with additional features, among which the hypothetical reasoning is found, as a platform for a fuzzy logic programming system.

What specific expressive capabilities for hypothetical reasoning λ-Prolog incorporates depends on the implementation. For example, we know in the words of D. Miller himself[2] about Teyjus, an implementation of λ-Prolog, that "Teyjus does not permit implications to be used in top-level goals. This is a characteristic that may change in the future when the compilation model is extended also to these goals but, for now, it means that some of the examples presented, eg, in Section 3.2, cannot be run directly using this system." Instead, the future implementation of Comp is planned to be able to allow this by compiling the goal in the context of the loaded program before submitting it.

Moreover, λ-Prolog does not work with "negative assumptions", a matter that we have let marked as future work, and that we will undertake by following some ideas already proposed for the implementation of a Fuzzy Datalog system (Julián-Iranzo and Sáenz-Pérez 2018; Julián-Iranzo and Sáenz-Pérez 2020).

Despite all the above and the possible relations between the foundations of our proposal and λ-Prolog, as we have just commented in this section, our work has its roots in the previous work carried out by Gabbay and Bonner. On the other hand, the main contribution of this article is the development of efficient high-level implementation techniques for implementing a fuzzy logic programming system (HBPL) with the possibility of hypothetical reasoning based on BPL (Rubio-Manzano and Julián-Iranzo 2014; Julián-Iranzo and Rubio-Manzano 2017). In such a system, assumptions imply compilations which with our proposal are possible to perform at compile-time.

[2] `stackoverflow.com/questions/65176668/?prolog-rejecting-hypothetical-reasoning-queries`

References

BONNER, A. J. 1988. A logic for hypothetical reasoning. In *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*. AAAI Press / The MIT Press, 480–484.

BONNER, A. J. 1990. Hypothetical datalog: Complexity and expressibility. *Theor. Comput. Sci. 76,* 1, 3–51.

BONNER, A. J. 1994. Hypothetical reasoning with intuitionistic logic. *Non-Standard Queries and Answers, Studies on Logic and Computation*, 187–219.

BONNER, A. J. 1997. A logical semantics for hypothetical rulebases with deletion. *J. Log. Program. 32,* 2, 119–170.

BOURDEAU, M. 2014. La théorie positive des hypothéses (in french). In *In Proc of the Conf. on Hypothetical Reasoning*. University of Tübingen, 23–29.

BOURDEAU, M. 2020. Auguste Comte. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University.

DOUVEN, I. 2021. Abduction. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University.

GABBAY, D. M. 1985. N-PROLOG: An extension of PROLOG with hypothetical implication II - logical foundations, and negation as failure. *J. Log. Program. 2,* 4, 251–283.

GABBAY, D. M. AND REYLE, U. 1984. N-PROLOG: An extension of PROLOG with hypothetical implications I. *J. Log. Program. 1,* 4, 319–355.

JULIÁN-IRANZO, P. AND RUBIO-MANZANO, C. 2017. A sound and complete semantics for a similarity-based logic programming language. *Fuzzy Sets and Systems*, 1–26.

JULIÁN-IRANZO, P. AND SÁENZ-PÉREZ, F. 2018. A Fuzzy Datalog Deductive Database System. *IEEE Transactions on Fuzzy Systems 26,* 5, 2634–2648.

JULIÁN-IRANZO, P. AND SÁENZ-PÉREZ, F. 2020. A System implementing Fuzzy Hypothetical Datalog. In *International Conference on Fuzzy Systems (FUZZ-IEEE), UK, July 19-24*. 1–8.

MCCARTY, L. T. 1988a. Clausal intuitionistic logic I - fixed-point semantics. *J. Log. Program. 5,* 1, 1–31.

MCCARTY, L. T. 1988b. Clausal intuitionistic logic II - tableau proof procedures. *J. Log. Program. 5,* 2, 93–132.

MILLER, D., NADATHUR, G., PFENNING, F., AND SCEDROV, A. 1991. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Log. 51,* 1-2, 125–157.

RUBIO-MANZANO, C. AND JULIÁN-IRANZO, P. 2014. Fuzzy Linguistic Prolog and its Applications. *Journal of Intelligent and Fuzzy Systems 26*, 1503–1516.

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.