Lollipops taste of Vanilla too

(*Extended Abstract*)

Iliano Cervesato

Dipartimento di Informatica
Università di Torino
Corso Svizzera, 185
10149 Torino - ITALY
iliano@di.unito.it

Post-ICLP'94 Workshop on
Proof-Theoretical Extensions of Logic Programming
S. Margherita Ligure, Italy
June 18th 1994

Lollipops taste of Vanilla too

(*Extended Abstract*)

Iliano Cervesato

Dipartimento di Informatica
Università di Torino
Corso Svizzera, 185
10149 Torino - ITALY
iliano@di.unito.it

1 Introduction

The logic programming language Prolog is notable, among other aspects, for the ease with which medium-granularity circular meta-interpreters can be written. It is indeed well-know that the functionalities of the Horn-clauses kernel of this language can be emulated by a three clauses program, known as the Vanilla meta-interpreter. A few additional clauses allow to cope with most extra-logical predicates. The Vanilla meta-interpreter stands as the starting point for a series of essential tools in a realistic Prolog programming environment, such as debuggers, tracers, pretty-printers, cost analyzers and program transformers, as well as for the definition of flexible expert systems tools [14, 15].

Most proposals for improving the declarative flavor or the expressiveness of Prolog give scarce consideration to this issue. In this paper, we show how to define a Vanilla-like meta-interpreter for the linear logic programming language Lolli [8, 9]. Lolli clauses depart in a substantial way from traditional Prolog clauses for the variety of linear connectives they can be built upon and for the peculiar use of the clauses kept in the so-called bound context. In particular, as for the language λProlog [11], goals can contain embedded implications as well as universal quantifications.

The first partial meta-interpreter for Lolli, actually a debugger, was written by the author when working at a Lolli implementation of a form of temporal reasoning, an extended event calculus [2]. The unexpected behavior of his first uncertain attempts and the absence of any programming facility in the first prototypal realization of Lolli suggested him to build his own programming environment. In a short time, Lolli has been endowed with a rich set of programming utilities. Moreover, these meta-interpreters, initially operative on the most commonly used connectives of Lolli, have been extended to cope with the full language.

2 Meta-Programming

A meta-program is a program that manipulates other programs as data. In this paper, we are primarily interested in circular meta-interpreters, i.e. interpreters for the language they are written in, or at least for a substantial subset of it. Logic programming languages are particularly suited to this task since logic programs have the structure of terms.

Meta-programs, in a logic programming framework, can be classified according to their granularity, i.e. the ratio between the amount of interpretation that is actually simulated and the quantity of work that is passed over to the underlying interpreter for the language. Most meta-programs written in Prolog have a medium granularity: they emulate clauses and facts selection, but

defer unification to the interpreter. The major representant of this family is the Vanilla meta-interpreter, on which most programming tools in a Prolog environment can be based.

The Vanilla meta-interpreter for the base Horn-clauses language consists of the following three clauses.

```
solve(true).                          (V1)
solve((A, B)) :-                      (V2)
  solve(A),
  solve(B).

solve(A) :-                           (V3)
  clause(A, B),
  solve(B).
```

The Vanilla meta-interpreter can be given a simple declarative reading: the formula `true` holds (clause V1); the formula (A,B) holds if both A and B hold (clause V2); finally, an atomic formula A holds if there is a program clause instance A`:-`B which body holds (clause V3). These three clauses can also be given an operational interpretation: clause (V1) states that the empty goal, represented by the constant `true`, is solved; to solve a conjunctive goal (A,B), first solve A, and then solve B; to solve an atomic goal A, find a clause in the program whose head unify with A and solve its body.

Vanilla assumes each clause H`:-`B in the object program to be represented as the fact `clause(`H`, `B`)`; a fact H is represented as `clause(`H`, true)`. Commercial Prologs contain `clause` as a built-in predicate, dispensing in this way the programmer from a boring hand translation.

The Vanilla meta-interpreter described by clauses (V1-V3) deals with the declarative subset of Prolog. Indeed, most extra-logical features can be readily embedded into its architecture. The following four clauses extend Vanilla to deal respectively with disjunction in goals, negation as failure, solutions grouping and generic system calls (i.e. arithmetic).

```
solve((A;B)) :-                       (V+4)
  solve(A); solve(B).
solve(not A) :-                       (V+5)
  not solve(A).
solve(bagof(X, G, Xs)) :-             (V+6)
  bagof(X, solve(G), Xs)).

solve(A) :-                           (V+7)
  functor(A, F, N),
  system(F, N),
  A.
```

Although a more complex task, Vanilla can even be enhanced to deal correctly control directives such as cut.

The basic Vanilla meta-interpreter has been subject to heavy criticisms in the recent years [6, 7, 10] as well as to objective defences [14, 15]. This dispute resulted in the design of a good number of extensions to Prolog or to its declarative core aimed at enhancing its meta-programming capabilities [1]. The language Gödel [6, 7, 10] is noteworthy in this respect for committing to typed versions of the Vanilla meta-interpreter. It is interesting to observe how most of these extensions loose the ease of programming medium-granularity meta-interpreter, although they improve the logical reading of finer-granularity meta-interpreters [1, 3, 4].

3 The Linear Logic Programming Lolli

Miller et al. [13] have defined a general criterion (the existence of *uniform proofs*) for extracting fragments of logical systems suited to be implemented as logic programming languages. The language of Horn-clauses, on which Prolog is based, has been proven to have the required properties, and has indeed been shown to exploit only to a minimal extent the connective patterns of predicate logic that ensure the existence of such proofs. In particular, they show that it is possible to go beyond Horn-clauses by allowing implication and universal quantification in goals. While not prejudicing the possibility of an efficient implementation, this enhancement increases enormously the expressiveness of the language [12]. The language λProlog is a realization of this idea in a simply-typed higher-order setting [11, 13].

λProlog stems from intuitionistic predicate logic. However, uniform proofs are a general concept in proof theory. The logic programming language Lolli [8, 9] is the result of applying this criterion to linear logic [5]. The short space available forbids us to give a detailed account of linear logic and to present a complete description of Lolli. The interested reader is invited to read the above mentioned references. We intend to illustrate only those aspects that are needed in the subsequent.

Linear logic is a refinement of traditional logic that constrains the number of times an assumption is used in a proof. A formula *G* is derivable from a set of formulas *D* if and only if there is a proof of *G* that uses exactly once every formula in *D*, with the exception of formulas that are preceded by the modal operator ! (read *of-course* or *bang*), which indicates that its argument can be used as many times as needed in the derivation, eventually zero. It is convenient to write linear sequents as $D; R \vdash G$, where *G*, *D* and *R* are respectively the formula to prove (the goal), the set of banged assumptions and the set of non-banged premises. *R* can be seen as perishable resources available for the proof of *G*, while the elements of *D* are reusable. *D* and *R* are called the *bound* and *unbound context* respectively.

The bipartition of the left-hand side of linear sequents, a direct consequence of the constraints on the use of formulas in proofs, entails the definition of a new set of connectives for the resulting logic. With a drastic simplification, we can say that every binary connective in traditional logic is transformed in a pair of linear connectives, one requiring the bound context to be duplicated when proving its subformulas, and the other having it split among them. The linear connectives on which Lolli is based are ⊗ (multiplicative conjunction, or *times*), & (additive conjunction, or *with*), ⊕ (multiplicative disjunction, or *oplus*), -o (linear implication, or *lollipop*), => (traditional implication), and in addition to these, ! (bang) and the two constants *true* and *erase*. Full linear logic, as originally defined in [5], extends this set with @ (additive disjunction or *par* - usually written as an upside-down ampersand &), $.^{\perp}$ (linear negation or *nil*) and the two constants *zero* and *false*.

The syntax of Lolli is reminiscent of that of Prolog, with some notational differences. Variables begin with an uppercase letter, unless explicitly quantified, while constants always start with a lowercase letter. Terms and atoms are written in curried form. For instance, the Prolog term `f(a, g(b, c), d)` is written in Lolli as `(f a (g b c) d)` (the outermost parentheses are not needed in practice).

Lolli programs and goals are constructed according to the grammar show below, where *A*, *G*, *R* and *D* are syntactic variables for atoms, goal formulas, linear clauses and banged clauses, respectively; the use of the *G*, *R* and *D* is coherent with their use in linear sequents above.

R ::= `true` | *A* | R_1 `&` R_2 | *R* `:-` *G* | *R* `<=` *G* | `forall` *x* `\` *R*

D ::= *R* | `{`*R*`}` | D_1 `,` D_2

G ::= `true` | `erase` | *A* | `{`*G*`}` | G_1 `&` G_2 | G_1 `,` G_2 | *D* `-o` *G* | *R* `=>` *G* | `forall` *x* `\` *G*
| `exists` *x* `\` *G* | G_1 `;` G_2 | G_1 `->` G_2 `|` G_3

We give now a connective-directed operational semantics for goals. We write $S = S_1 + S_2$ to indicate that S_1 and S_2 form a partition of the set *S*. In order to keep the notation simple, we write $S+s$ and $S \cup s$ instead of $S+\{s\}$ and $S \cup \{s\}$ when no ambiguity arises.

`true` always succeeds without consuming any resource. Therefore, any formula in the bound context must be used in other parts of the proof of the current goal.

$D; \varnothing \vdash_{\text{LOLLI}}$ `true`

`erase` always succeeds too, but consumes every resource in the bound context. It is usually exploited to ignore information eventually present into the bound context and not needed in the proof of the current goal.

$D; R \vdash_{\text{LOLLI}}$ `erase`

`{`G`}` (*bang* G) attempts to solve G without using any formula in the bound context.

$D; R \vdash_{\text{LOLLI}} \{G\}$ iff $D; \varnothing \vdash_{\text{LOLLI}} G$

G_1 `,` G_2 (G_1 *times* G_2) first attempts to find a proof for G_1 and, in case of success, looks for a proof for G_2 that uses whatever elements of the bound context were not used in the proof of G_1 . Thus, it divides the resources in the bound context between G_1 and G_2.

$D; R_1 + R_2 \vdash_{\text{LOLLI}} G_1 , G_2$ iff $D; R_1 \vdash_{\text{LOLLI}} G_1$ and $D; R_2 \vdash_{\text{LOLLI}} G_2$

G_1 `&` G_2 (G_1 *with* G_2) operates as in the previous case with the difference that it tries to prove both G_1 and G_2 using the same resources. Therefore, it duplicates the resources in the bound context for each conjunct.

$D; R \vdash_{\text{LOLLI}} G_1 \,\&\, G_2$ iff $D; R \vdash_{\text{LOLLI}} G_1$ and $D; R \vdash_{\text{LOLLI}} G_2$

G_1 `;` G_2 (G_1 *oplus* G_2) first attempts to find a proof for G_1 and, in case of success, have the overall goal succeed. In case of failure, a proof for G_2 is attempted using the same set of resources as for G_1.

$D; R \vdash_{\text{LOLLI}} G_1 \,;\, G_2$ iff $D; R \vdash_{\text{LOLLI}} G_1$ or $D; R \vdash_{\text{LOLLI}} G_2$

r `-o` G (r *lollipop* G) causes the clause r to be added to the bound context and a proof for G to be attempted from this augmented program.

$D; R \vdash_{\text{LOLLI}} r \text{ -o } G$ iff $D; R + r \vdash_{\text{LOLLI}} G$

d `=>` G (d *implies* G) operates like in the previous case, but adds d to the unbound context. It is logically equivalent to `{`d`}-o` G.

$D; R \vdash_{\text{LOLLI}} d \Rightarrow G$ iff $D \cup d; R \vdash_{\text{LOLLI}} G$

`forall` x `\` G (universal quantification) is proven by finding a proof for a generic instantiation of G: the variable x is substituted with a new constant c, not present in neither in G nor into the current context.

$D; R \vdash_{\text{LOLLI}} \texttt{forall}\ x \setminus G$ iff $D; R \vdash_{\text{LOLLI}} [c/x]G$ where c is a new constant

`exists` x `\` G (existential quantification) holds if a term t can be found such that G has a proof when substituting x with t. In practice, it causes a new logical variable to be substituted for x and then forgotten when returning from the call to G.

$D; R \vdash_{\text{LOLLI}} \texttt{exists}\ x \setminus G$ iff $D; R \vdash_{\text{LOLLI}} [t/x]G$ for some term t

G`->`G_s`|`G_f (guarded expression) is the one extra-logical operator included into the current implementation of Lolli (apart from a few I/O predicates). A proof of G is attempted. In case of success, the overall goal succeeds if G_s succeeds. If instead the interpreter fails to find a proof for G, it tries to find a proof for G_f and its success determines the success of the overall goal.

$D; R \vdash_{\text{LOLLI}} G \text{ -> } G_s \text{ | } G_f$ iff if $D; R \vdash_{\text{LOLLI}} G$ then $D; R \vdash_{\text{LOLLI}} G_s$ else $D; R \vdash_{\text{LOLLI}} G_f$

Such guarded expressions are particularly useful to model negation as failure, defined by the following clause.

```
not G :- G -> fail | true.
```

In order to describe the behavior of Lolli when the current goal is an atom, we need to spend some words illustrating the different kinds of clauses definable in this language, as specified by the grammar rules for D (and R) given above. First, notice that the head of a clause is not constrained to be an atom, like in more traditional logic programming languages, but is allowed to be an additive conjunction of atoms of any length (logical equivalences permit us to limit ourselves to this case). An atomic goal matches such a clause if it matches any of the conjuncts. In this case, the execution backchains over its body, as if the clause had a single head. A second peculiarity of Lolli clauses, deriving from its origin from linear logic, is their subdivision in linear and non-linear (banged). As a matter of convenience, since most clauses in a program are indeed reusable, linear clauses are

marked by means of the prefix `LINEAR` while non-linear clauses remain unqualified. Finally, notice that the clauses in a program can be separated either by multiplicative or additive conjunction, with the restriction that non-linear clauses must be separated by the former. Additive conjunction on linear clauses achieves exclusivity.

A looks for a linear or banged clause h :- b (eventually a fact h) which head unifies with the atomic goal A and tries to solve its body in the current context. If the selected clause is linear, it is removed from the bound context.

$D \cup h$:- b; $R \vdash_{\text{LOLLI}} A$ if $\sigma = match(A, h)$ and $D \cup h$:- b; $R \vdash_{\text{LOLLI}} b^\sigma$

D ; $R + h$:- $b \vdash_{\text{LOLLI}} A$ if $\sigma = match(A, h)$ and D; $R \vdash_{\text{LOLLI}} b^\sigma$

D ; $R + c_1 \& ... \& c_n \vdash_{\text{LOLLI}} A$ if D; $R + c_i \vdash_{\text{LOLLI}} A$ for some i=1...n

where $\sigma = match(A, A_1 \& ... \& A_n)$ iff $A^\sigma = A_i^\sigma$ for some i=1...n.

Implication in goals combined with universal quantification yields a powerful scoping mechanism [12]. Lolli provides a module system realized through the combination of the two. Modules are parametric and permit abstraction and information hiding by means of the `LOCAL` declaration. We do not investigate the meta-level treatment of the module system of Lolli in this paper.

4 A Vanilla Meta-Interpreter for Lolli

Program 1 implements a Vanilla-like meta-interpreter for Lolli, written in Lolli. Due to the richness of linear connectives available in Lolli to construct both goals and programs, the resulting code cannot be as short as in the case of pure Prolog. Nevertheless, the resulting program stays comfortably into one page.

Our meta-interpreter assumes the following meta-representation for Lolli programs:

- a non-linear clause (or fact) *D* is reified as `clause (`*D*`)`;
- a linear clause (or fact) `LINEAR` *D* is represented, at the meta-level as `LINEAR clause (`*D*`)`.

It is a simple exercise to write a LEX and YACC program that transform a correct Lolli program according to these rules. An implementation of such a program transformer directly in Lolli has been attempted without success due to the difficulty of reading unformated data from a file.

The grammar of Lolli presented above gives the user the freedom of writing his/her programs in several logically equivalent ways. Nevertheless, we require him/her to use a reasonably simple syntax. For instance, the above meta-interpreter will not handle correctly the clause `LINEAR clause ({`*D*`})`, originally written as `LINEAR {`*D*`}`. The latter is indeed logically equivalent to *D*.

This meta-interpreter is clearly constructed according to the philosophy underlying the basic Vanilla: the object level connectives and operators are directly mapped to the meta-level corresponding entities (clauses G1-G14); notice that this same technique applies to quantifiers too (clauses G11-G12). Intuitionistic and linear implication escape partially to this rule since they require their left-hand side to be added to the unbound and bound context respectively (Clauses G9-G10). This is easily obtained by embedding the augmenting clause in a `clause` atom and loading the resulting clause in the appropriate context. Due to the variety of forms a linear clause can assume, linear implication can be handled in different ways depending on the pattern of its left-hand side (clauses L1-L5); most of these clauses are rewriting rules relating equivalent linear logic formulas.

A substantial difference from the Horn-clauses Vanilla meta-interpreter concerns the way atomic goals are handled. This is once more due to the multiplicity of clauses types that can appear in a Lolli program. Clauses (D1-D7) deal with this multiplicity of cases. These rules rely on the auxiliary predicates `inHead` (clauses A1-A2), that checks if an atomic goal matches an element in the head of a clause, represented as an additive conjunction of goals, `inTimes` (clauses A3-A4) and `inWith` (clauses A5-A6) that deal with respectively multiplicative and additive conjunctions of clauses.

```
MODULE meta.

LOCAL    not solveLollipop solveAtomic inTimes inWith.

solve true :- true.                          (G1)
solve erase :- erase.                        (G2)
solve fail :- fail.                          (G3)
solve top :- top.                            (G4)
solve ({G}) :-                               (G5)
  {solve G}.
solve (G1 , G2) :-                           (G6)
  solve G1,
  solve G2.
solve (G1 & G2) :-                           (G7)
  solve G1 &
  solve G2.
solve (G1 ; G2) :-                           (G8)
  solve G1;
  solve G2.
solve (D -o G) :-                            (G9)
  solveLollipop D G.
solve (R => G) :-                            (G10)
  clause R => solve G.
solve (forall X \ G) :-                      (G11)
  forall X \ solve G.
solve (exists X \ G) :-                      (G12)
  exists X \ solve G.
solve (G -> Gs | Gf) :-                      (G13)
  solve G -> solve Gs | solve Gf.
solve A :-                                   (G14)
  clause D,
  solveAtomic A D.

solveLollipop (D1 -o D2) G :-                (L1)
  solveLollipop D1 (D2 -o G).
solveLollipop (D1 => D2) G :-                (L2)
  clause D1 => solve (D2 -o G).
solveLollipop (D1, D2) G :-                  (L3)
  solveLollipop D1 (D2 -o G).
solveLollipop ({D}) G :-                     (L4)
  clause D => solve G.
solveLollipop D G :-                         (L5)
  clause D -o solve G.

solveAtomic A A.                             (D1)
solveAtomic A ({D}) :-                       (D2)
  solveAtomic A D.
solveAtomic A (D1, D2) :-                    (D3)
  inTimes A (D1, D2).
solveAtomic A (R1 & R2) :-                   (D4)
  inWith A (R1 & R2).
solveAtomic A (R :- G) :-                    (D5)
  inHead A R,
  solve G.
solveAtomic A (R <= G) :-                    (D6)
  inHead R,
  solve ({G}).
solveAtomic A (forall X \ R) :-              (D7)
  solveAtomic A R.

inHead A A.                                  (A1)
inHead A (R1 & R2) :-                        (A2)
  inHead A R1;
  inHead A R2.

inTimes A (D1 , D2) :-                       (A3)
  solveAtomic A D1;
  solveAtomic A D2.
inTimes A D :-                               (A4)
  not (D = (D1 , D2)),
  solveAtomic A D.

inWith A (R1 & R2) :-                        (A5)
  solveAtomic A R1;
  solveAtomic A R2.
inWith A R :-                                (A6)
  not (R = (R1 & R2)),
  solveAtomic A R.

not A :-                                     (A7)
  A -> fail | true.
```

Program 1

References

[1] K.A. Bowen, R. Kowalski: "Amagalmating language and meta-language in logic programming", in *Logic Programming* (K.L. Clark, S.Å. Tärnlund Eds.), pp 153-172, Academic Press, 1982.

[2] I. Cervesato, L. Chittaro, A. Montanari: "Modal event calculus in Lolli", submitted to the *International Logic Programming Symposium - ILPS'94*, Ithaca, NY, USA, November 1994.

[3] I. Cervesato, G.F. Rossi: "Logic meta-programming facilities in 'Log'", *Proceedings of the Third International Workshop on Meta-Programming in Logic Programming - META'92*, pp 148-161, Uppsala, Sweden, June 1992, Springer-Verlag LNCS 649.

[4] S. Costantini, G.A Lanzarone.: "A Metalogic Programming Language.", in "*Proceedings of the Sixth International Conference on Logic Programming*" (G. Levi, M. Martelli, Eds), pp 218-233, MIT Press, 1989.

[5] J.Y. Girard: "Linear logic", in *Theoretical Computer Science*, Vol. 50, pp 1-101, North-Holland, Amsterdam, 1987.

[6] P.M. Hill, J.W. Lloyd: "The Gödel report (preliminary version)", *Technical Report n. TR-91-02*, University of Bristol, Department of Computer Science, March 1991.

[7] P.M. Hill, J.W. Lloyd: "*Analysis of meta-programs*", Technical Report CS-88-08, Department of Computer Science, University of Bristol, 1988.

[8] J.S. Hodas: Documentation of the release 0.7 of Lolli, available by anonymous ftp to *ftp.cis.upenn.edu*, directory */pub/Lolli*.

[9] J.S. Hodas, D. Miller: "Logic Programming in a Fragment of Linear Logic", to appear in the *Journal of Information and Computation*.

[10] J.W. Lloyd: "Directions for meta-programming", in Proceedings of the *International Conference on the Fifth Generation Computer Systems*, pp 609-617, Tokyo, Japan, November 1988.

[11] D. Miller: "A logic programming language with lambda-abstraction, function variables, and simple unification", in *Proceedings of the International Workshop on Proof-Theoretical Extensions of Logic Programming* (P. Schroeder-Heister Ed.), pp 253-281, Tuebingen, Gemany, 1989, Springer-Verlag LNAI 475.

[12] D. Miller: "A logical analysis of modules in logic programming", in *Journal of Logic Programming*, Vol. 6, 1989, pp 79-108.

[13] D. Miller, G. Nadathur, F. Pfenning, A. Scedrov: "Uniform proofs as a foundation for logic programming", *Annals of Pure and Applied Logic* 51 (1991), pp 125-157, North-Holland.

[14] L. Sterling: "*The paradigm of meta-programming*", Lecture notes of the *Second Workshop on Meta-Programming in Logic Programming - META'92* (M. Bruynooghe Ed.), Leuven, Belgium, April 1990.

[15] L. Sterling, E. Shapiro: "*The art of Prolog*", MIT Press, 1986.