Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Extending Logic Programming with Labelled Variables: Model and Semantics

(Article begins on next page)

28 April 2024

Extending Logic Programming with Labelled Variables: Model and Semantics

Roberta Calegari

ALMA MATER STUDIORUM–*Università di Bologna*
viale Risorgimento 2, 40136, Bologna, Italy
roberta.calegari@unibo.it

Enrico Denti

ALMA MATER STUDIORUM–*Università di Bologna*
viale Risorgimento 2, 40136, Bologna, Italy
enrico.denti@unibo.it

Agostino Dovier[*]

Università degli Studi di Udine
via delle Scienze 206, 33100, Udine, Italy
agostino.dovier@uniud.it

Andrea Omicini

ALMA MATER STUDIORUM–*Università di Bologna*
via Sacchi 3, 47521, Cesena, Italy
andrea.omicini@unibo.it

Abstract. In order to enable logic programming to deal with the diversity of pervasive systems, where many heterogeneous, domain-specific computational models could benefit from the power of symbolic computation, we explore the expressive power of labelled systems. To this end, we define a new notion of truth for logic programs extended with *labelled variables* interpreted in non-Herbrand domains—where, however, terms maintain their usual Herbrand interpretations.

First, a model for labelled variables in logic programming is defined. Then, the fixpoint and the operational semantics are presented and their equivalence is formally proved. A meta-interpreter implementing the operational semantics is also introduced, followed by some case studies aimed at showing the effectiveness of our approach in selected scenarios.

Keywords: logic programming, labelled systems, labelled variables, formal semantics, meta-interpretation, situated intelligence

1. Introduction

In order to face the challenges of today's pervasive systems – which are inherently complex, distributed, situated, and intelligent [1] – suitable models and technologies are required for the support of *distributed situated intelligence*.

In principle, logic programming (LP henceforth) has the potential to power the core of such models and technologies, thanks to its declarative interpretation and inferential capabilities: however, doing so effectively requires that LP is suitably extended to capture the many different domains calling for situated intelligence [2]. In fact, while logic-based approaches are natural candidates for intelligent systems, a pure LP approach does not straightforwardly fit the needs of situated systems: hence the value added of a *hybrid* approach, which makes it possible to exploit LP for what it is most suited for – such as symbolic computation – while delegating other aspects – such as situated computations – to other languages, or, to other levels of computation.

Along this line, in this paper we present a LP extension based on *labelled variables* [3, 4], called *Labelled Variables in Logic Programming (LVLP)*, whose purpose is to enable diverse computational models, each one tailored to the specific needs of situated components, to coherently and fruitfully coexist and interact within a logic-based framework. In particular, we extend our previous work presented in [5] by defining the formal model of LVLP in a more concise way, providing complete proofs of its formal properties, extending the range of admissible labels, exposing the meta-interpreter, and adding some examples along with their implementation upon the current LVLP prototype.

Unlike most other works in this area – such as [6, 7, 8, 9] –, which primarily focus on specific scenarios and systems (modal logic, deductive systems, fuzzy systems, etc.), LVLP aims at providing a general-purpose logic framework along with the mechanisms needed to fit, potentially, whatever specific application scenarios. From another perspective, Hofstedt [10] focuses on combining specialised constraint solvers into a single solver, to handle mixed-domain constraints: while its goal is somehow similar, LVLP is geared instead towards a multiplicity of distributed logic engines, situated and specialised, with limited computational requirements, addressing the needs of distributed intelligence for pervasive systems.

Some approaches dealing with inconsistencies emerging in large databases exploit the notion of *annotation*—e.g., [11]: there, atoms can be annotated by labels chosen in upper semilattices. However, upper semilattices represent a too-rigid constraint for LVLP, where labels aim instead at modelling even weakly-structured domains—such as, say, food, or colours; even more, allowing terms to be directly labelled in LVLP would result in a too-strict coupling between LP and the label domain, which instead LVLP aims at keeping well-separate, by limiting labelling to variables.

So, in the following we introduce the LVLP framework, moving from the definition of the theoretical model (Section 2) to the fixpoint and operational semantics (Section 3): we discuss correctness, completeness, and their equivalence is formally proved. Then, we present a meta-interpreter implementing the operational semantics (Section 4), which is exploited to illustrate LVLP through some case studies in different domains (Section 5). Finally, related works are discussed (Section 6).

2. Labelled Variables in Logic Programming (LVLP)

2.1. The model

Let $\mathscr{C}$ be the set of *constants*, with $c_1, c_2 \in \mathscr{C}$ being two generic constants. Let $\mathscr{V}$ be the set of *variables*, with $v_1, v_2 \in \mathscr{V}$ being two generic variables. Let $\mathscr{F}$ be the set of *function symbols*, with $f_1, f_2 \in \mathscr{F}$ being two generic function symbols; each $f \in \mathscr{F}$ is associated to an arity $ar(f) > 0$, stating the number of function arguments. Let $\mathscr{T}$ be the set of *terms*, with $t_1, t_2 \in \mathscr{T}$ being two generic terms. Terms can be either simple – a constant (e.g., c_1) and a variable (e.g., v_2) are both *simple terms* – or compound—a functor of arity n applied to n terms (e.g., $f_1(c_2, v_1)$) is a *compound*

term. A term is said *ground* if it does not contain variables. Let $\mathscr{H}$ denote the set of ground terms, also known as the *Herbrand universe*.

A *model* for *Labelled Variables in Logic Programming* (LVLP) is defined as a triple $\langle \mathscr{B}, f_L, f_C \rangle$, where

- $\mathscr{B} = \{\beta_1, \ldots, \beta_n\}$ is the set of *basic labels*, the basic entities of the *domain of labels*

- $\mathscr{L} \subseteq \wp(\mathscr{B})$ is the set of *labels*, where each label $\ell \in \mathscr{L}$ is a subset of $\mathscr{B}$

- $f_L : \mathscr{L} \times \mathscr{L} \longrightarrow \mathscr{L}$ is the *(label-)combining function* computing a new label from two given ones

- $f_C : \mathscr{H} \times \mathscr{L} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ is the *compatibility function*, assessing the compatibility between a ground term and a label when interpreted in the domain of labels

- a *labelled variable* is a pair $\langle v, \ell \rangle$ associating label $\ell \in \mathscr{L}$ to variable $v \in \mathscr{V}$

- a *labelling* is a set of *labelled variables*

$\mathscr{B}$ can be either finite or infinite—in the latter case, with the two extra requirements that *(i)* each label ℓ can be represented finitely, including the new labels generated by the combining function f_L, and *(ii)* the compatibility function f_C can argue over the representation. Also, for the sake of simplicity, a "singleton" label $\{\beta\}$ where $\beta \in \mathscr{B}$ will be written just as β henceforth, and a "singleton" labelling $\{\langle v, \ell \rangle\}$ will be written as $\langle v, \ell \rangle$, and as $v^\wedge \ell$ in the examples.

Finally, we require that f_L is associative, commutative, and idempotent, with the empty set as its neutral element—namely:

$$f_L(\ell_1, f_L(\ell_2, \ell_3)) = f_L(f_L(\ell_1, \ell_2), \ell_3), \quad f_L(\ell_1, \ell_2) = f_L(\ell_2, \ell_1), \quad f_L(\ell, \ell) = \ell$$

Accordingly, in order to simplify notation, in the following we will simply write $f_L(\ell_1, \ldots, \ell_{n-1}, \ell_n)$ instead of $f_L(\ell_1, f_L(\ldots, f_L(\ell_{n-1}, \ell_n) \ldots))$.

Details on f_C and f_L are provided in the remainder of the paper, in particular in Subsection 2.3.

2.2. Programs, clauses, unification

An LVLP program is a collection of LVLP *rules* of the form

$$Head \leftarrow Labelling, Body.$$

to be read as "*Head* if *Body* given *Labelling*". There, *Head* is an atomic formula, *Labelling* is the list of labelled variables in the clause, and *Body* is a list of atomic formulas.

As in standard LP [12, 13], an atomic formula (or atom) has the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol and t_i are terms. Atom $p(t_1, \ldots, t_m)$ is said *ground* if $t_1, \ldots, t_n$ are ground. Predicate symbols represent relations defined by a logic program, whereas terms represent the elements of the domain. $\mathscr{H}_B$ is the *Herbrand base*, namely the set of all ground atoms of whose argument terms are in $\mathscr{H}$. Every variable occurring in a clause is universally quantified, and its scope is the clause in which the variable occurs.

An essential LP mechanism is represented by *unification*, involving two different terms that are supposed to refer to the same domain element. While discussing LP unification is out of the scope of this paper (we refer the reader to [12, 13] for the basics of LP), any extension to LP needs to define its own unification rules.

	constant c_2	variable v_2	labelled variable $\langle v_2, \ell_2 \rangle$	compound term s_2
constant c_1	if $f_C(c_1, c_2) =$ true then true else false	true, $\{v_2/c_1\}$	if $f_C(c_1, \ell_2) =$ true then true, $\{v_2/c_1\}, \ell_2$ else false	false
variable v_1	true, $\{v_1/c_2\}$	true, $\{v_1/v_2\}$	true, $\{v_1/v_2\}, \ell_2$	if v_1 does not occur in s_2 then true, $\{v_1/s_2\}$ else false
labelled variable $\langle v_1, \ell_1 \rangle$	if $f_C(c_2, \ell_1) =$ true then true, $\{v_1/c_2\}, \ell_1$ else false	true, $\{v_1/v_2\}, \ell_1$	if $f_L(\ell_1, \ell_2) \neq \emptyset$ then true, $\{v_1/v_2\}, f_L(\ell_1, \ell_2)$ else false	if v_1 does not occur in s_2, and $f_L(\ell_1, \ell_1', ..., \ell_n') \neq \emptyset$ where $\ell_1', ... \ell_n'$ are the labels in s_2 then true, $\{v_1/s_2\}, f_L(\ell_1, \ell_1', ..., \ell_n')$ else false
compound term s_1	false	if v_2 does not occur in s_1 then true, $\{v_2/s_1\}$ else false	if v_2 does not occur in s_1, and $f_L(\ell_2, \ell_1', ..., \ell_n') \neq \emptyset$ where $\ell_1', ... \ell_n'$ are the labels in s_1 then true, $\{v_2/s_1\}, f_L(\ell_2, \ell_1', ..., \ell_n')$ else false	if s_1, s_2 have same functor / arity, and their arguments (recursively) unify then true else false

Table 1. Unification rules in LVLP, adopting standard LP unification rules and representation

Thus, Table 1 reports the unification rules for LVLP. Since, by design, only variables can be labelled, the only case to be added to the standard unification table is represented by labelled variables. There, given two generic LVLP terms, the unification result is represented by the extended tuple

$$(\mathsf{true/false}, \theta, \ell)$$

where true/false represents the existence of an answer, θ is the *most general unifier* (mgu), and ℓ is the new label associated to the unified variables defined by the (label-)combining function f_L. In order to lighten the notation, undefined elements in the tuple (i.e., labels or substitutions that make no sense in a given case) are omitted in Table 1.

Taking into account all the variables of a goal, a solution for a LVLP computation is represented by the extended tuple

$$(\mathsf{true/false}, \Theta, A)$$

where Θ represents the mgu for all the variables, and A represents the corresponding labelling.

2.3. Compatibility

Expressing the solution of the labelled variables program as a tuple (true/false, Θ, A) implicitly assumes that the LP computation, whose answer is given by Θ, and the label computation, whose answer is given by A, can be read somehow independently from each other. So, whereas any computed label-variable association could be acceptable as far as LP is concerned (where symbols are uninterpreted), some label-variable association could be actually unacceptable when interpreted in the domain of labels.

To formalise such a notion of acceptability, the *compatibility function* f_C is defined as follows:

$$f_C : \mathscr{H} \times \mathscr{L} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$$

In particular, given a a ground term $t \in \mathscr{H}$ and label $\ell \in \mathscr{L}$:

$$f_C(t, \ell) = \begin{cases} \mathsf{true} & \text{if there exists at least one element of the domain of} \\ & \text{labels which the interpretations of } t \text{ and } \ell \text{ both refer to} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Example 2.1 illustrates an application scenario where variables are labelled with their admissible numeric interval, formalising the f_L and f_C functions accordingly.

Example 2.1. (LVLP with numeric intervals)

As a simple LVLP example, let us suppose that logic variables span over integer domains, and are labelled with their admissible numeric intervals. The combining function f_L, which embeds the scenario-specific label semantics, is supposed to intersects intervals—that is, given two labels $\ell_1 = \{\beta_{11}, \ldots, \beta_{1n}\}$ and $\ell_2 = \{\beta_{21}, \ldots, \beta_{2m}\}$, the resulting label ℓ_3 is the intersection of ℓ_1 and ℓ_2:

$$\ell_3 = f_L(\ell_1, \ell_2) = f_L(\{\beta_{11}, \ldots \beta_{1n}\}, \{\beta_{21}, \ldots \beta_{2m}\}) =$$
$$\{(\beta_{11} \cap \beta_{21}), \cdots, (\beta_{11} \cap \beta_{2m}), \ldots, (\beta_{1n} \cap \beta_{21}), \cdots, (\beta_{1n} \cap \beta_{2m})\}$$

Here the LP computation aims at computing numeric values, while the label computation aims at computing admissible numeric intervals for logic variables.

In principle, since the LP computation and the label computation proceed independently, the solution tuple (true/false, Θ, A) could also describe situations such as (true, $X/3$, $\langle X, [4, 5] \rangle$), where logic variable X would be associated to both value 3 and label $[4, 5]$. However, if the numeric intervals are to be interpreted as the boundaries for acceptable values of LP variables, such labelling would be inconsistent, and the system should reject such a solution as *incompatible*.

This is what the compatibility function f_C is for: $f_C(t, \ell)$ connects the LP and the label universes by checking whether ground term $t \in \mathscr{H}$ is *compatible* with label $\ell \in \mathscr{L}$. In particular, in our example $f_C(t, \ell)$ is supposed to be true only if t belongs to the interval represented by ℓ: in this case, $f_C(3, [4, 5])$ should reasonably return false, rejecting labelling $\langle X, [4, 5] \rangle$, with $X = 3$, as incompatible.

Summing up, the result of a LVLP program can be written as

$$((\text{true/false}) \wedge f_C(\Theta, A), \Theta, A)$$

meaning that the truth value potentially computed by the LP computation can be restricted – i.e., forced to false – by $f_C(\Theta, A)$; in turn, this is just a convenient shortcut for the conjunction of all $f_C(t, \ell)$ $\forall(t, \ell)$ pairs, where ℓ and t are such that $\langle v, \ell \rangle \in A$ and $v/t \in \Theta$. Of course, in case of independent domains, $f_C(t, \ell)$ is merely true $\forall t$ and $\forall \ell$.

2.4. The LVLP vision: Enabling distributed situated intelligence

The requirement for intelligence in pervasive systems is ubiquitous: computation surrounds us, devices and software components are required to behave intelligently, by understanding their own goals as well as the context where they work; integration of software components is supposed to add further (social) intelligence, possibly through coordination [14]. This is the case, for instance, of the Internet of Things (IoT) [15, 16, 17], where physical objects are networked, and are required to understand each other, learn, understand situations, and understand us [18]—in short, our everyday objects are expected to be(come) intelligent in the Internet of Intelligent Things (IoIT) [19].

In the overall, IoIT scenarios mandate for distributed and situated *micro-intelligence*, where huge numbers of small units of computation, situated within a spatially-distributed environment, are required to behave in a smart way, and need to cooperate in order to achieve a coherent and intelligent social behaviour. However, engineering effective *distributed situated intelligence* is far from trivial, mostly due to *(i)* the huge amount of data, information, and knowledge to handle, *(ii)* the need for adaptation and self-management, *(iii)* the requirements of resource constrained devices, *(iv)* the heterogeneity of models and technologies against interoperability, and *(v)* the many diverse specific domains to be integrated.

Along that line, the goal of the LVLP model is to exploit the potential of LP and its extensions as sources of micro-intelligence for IoIT scenarios, in particular to deal with the domain-specific aspects. The LVLP domain-specific perspective further emphasises the role of situatedness, already brought along by spatial distribution of components in pervasive systems.

Accordingly, lightweight and interoperable LVLP Prolog engines could be distributed even on resource-constrained devices [20]: multiple logic theories would then be scattered around, encapsulated in each engine, and associated to individual computational devices and things in the IoT. As a result, each logic theory is conceived as *situated*, and represents what is *locally* true, according to a simple paraconsistent overall interpretation. The LP resolution process is then local to each theory / engine, so it is both standard and consistent [21]. Thus, LVLP allows in principle logic-based micro-intelligence to be encapsulated within devices of any sort, and make them work together in groups,

aggregates, and societies, by promoting features such as observability, malleability, understandability, and formalisability via LP.

3. Semantics

In order to maintain the basic theoretical results of LP, such as the equivalence of denotational and operational semantics, labels domains must support tests and operations on labels.

To this end, Subsection 3.1 defines the denotational (fixpoint) semantics in the context of labels domain (under reasonable requirements for f_L), while Subsection 3.2 discusses the operational semantics of the model through an abstract state machine.

3.1. Fixpoint semantics

Let us call $\mathcal{X} = (\mathcal{H}, \mathcal{L})$ a *LVLP domain*, and define the notion of $\mathcal{X}$-interpretation I as a set of pairs of the form

$$\langle p(t_1, \ldots, t_n), [\ell_1, \ldots, \ell_n] \rangle$$

where $p(t_1, \ldots, t_n)$ is a ground atom, and $\ell_1, \ldots, \ell_n$ are labels s.t. for $i = 1, \ldots, n$ the term t_i is *compatible* with the label ℓ_i, i.e., $f_C(t_i, \ell_i) = \text{true}$. Truthness of f_C is based on the LVLP domain $\mathcal{X}$. With a slight abuse of notation, we write $\mathcal{X} \models [\langle t_1, \ell_1 \rangle, \ldots, \langle t_n, \ell_n \rangle]$ iff $\bigwedge_{i=1}^{n} f_C(t_i, \ell_i) = \text{true}$. We also write $\mathcal{X} \models \langle p(t_1, \ldots, t_n), [\ell_1, \ldots, \ell_n] \rangle$ if p is a predicate symbol and $\bigwedge_{i=1}^{n} f_C(t_i, \ell_i) = \text{true}$.

We denote as Λ the part of clause body that stores labelling. Without loss of generality we assume that there is exactly one labelling for each variable in the head. We define the function $\widetilde{f_L}$ that extends f_L and takes as arguments, orderly, a rule

$$r = h \leftarrow \Lambda, b_1, \ldots, b_n$$

a labelling, and n lists of labels. The rule r is used here to identify multiple occurrences of the variables. Let us assume the variables in h are $x_1, \ldots, x_m$, and consider one of them, say x_i. If x_i occurs in h (and hence in Λ) and in (some of) $b_1, \ldots, b_n$ then the corresponding labels $\ell, \ell_{1,i}, \ldots, \ell_{n,i}$ for x_i are retrieved from Λ (if x_i does not occur in b_j we simply do not consider such contribution). Then $\ell'_i = f_L(\ell, f_L(\ell_{1,i}, \ldots, f_L(\ell_{n-1,i}, \ell_{n,i})))$ is computed, and the pair $\langle x_i, \ell'_i \rangle$ is returned. This is done for all variables $x_1, \ldots, x_m$ occurring in the head h, and the list $[\ell'_1, \ldots, \ell'_m]$ is returned.

The denotational semantics is based on the one-step consequence functions T_P defined as:

$$
T_P(I) = \left\{
\begin{array}{ll}
\langle p(\widetilde{t}), \widetilde{\ell} \rangle : \\[4pt]
r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}, \, b_1, \ldots, b_n & (1) \\[2pt]
\text{where r is a fresh renaming of a rule of } P, \\[2pt]
v \text{ is a valuation on } \mathcal{H} \text{ such that } v(\widetilde{x}) = \widetilde{t}, & (2) \\[2pt]
\bigwedge_{i=1}^{n} \exists \, \widetilde{\ell}_i \text{ s.t. } \langle v(b_i), \widetilde{\ell}_i \rangle \in I, & (3) \\[2pt]
\mathcal{X} \models \Lambda_{\widetilde{t}} \wedge \bigwedge_{i=1}^{n} f_C\big(v(b_i), \widetilde{\ell}_i\big), & (4) \\[2pt]
\widetilde{\ell} = \widetilde{f_L}\big(r, \Lambda_{\widetilde{t}}, \widetilde{\ell}_1, \cdots, \widetilde{\ell}_n\big) & (5)
\end{array}
\right\}
$$

where $\Lambda_{\widetilde{t}} = v(\Lambda_{\widetilde{x}}) = [\langle t_1, \ell_1 \rangle, \ldots, \langle t_n, \ell_m \rangle]$ if $\Lambda_{\widetilde{x}} = [\langle x_1, \ell_1 \rangle, \ldots, \langle x_m, \ell_m \rangle]$. Notice that the condition $\mathcal{X} \models \bigwedge_{i=1}^n f_C\big(v(b_i), \widetilde{\ell_i}\big)$ in line 4 is always satisfied when T_P is used bottom-up, starting from $I = \emptyset$.

For the sake of convenience, unspecified labels are assumed to be read as the *any* label, defined as the neutral element of f_L: in this way, any standard (i.e. non labelled) LP variable can be read as implicitly labelled with such *any* label—represented as $\diamond$ henceforth.

Example 3.1 shows the computation of the least fixpoint of T_P in a simple case. There, and in the following examples, =/2 represents the equality operator of LVLP, whose behaviour is described in Table 1 (unification rules), and can be summarised as:

$$X = Y : -[< X, \diamond >, < Y, \diamond >], X =_{LP} Y$$

where $=_{LP}$ is the =/2 standard LP unification operator.

Example 3.1. (Computing the least fixpoint of T_P in a simple case)

Let us consider the LVLP program P:

```
r(0). r(1). r(2). r(3). r(4). r(5). r(6). r(7). r(8). r(9).
q(Y,Z) :- Y^[[2,4]], Z^[[3,8]], Y=Z, r(Y), r(Z).
p(X,Y,Z) :- X^[[0,3]], Y^[◊], Z^[◊], X=Y, q(Y,Z).
```

where $\diamond$ is used as a shortcut for any basic label (any interval).
Let us consider the interpretation $I_0 = \emptyset$. Then, the next interpretation I_1 can be obtained as:

$$I_1 = T_P(I_0) = \big\{ \langle r(0), [\diamond] \rangle, \ldots, \langle r(9)[\diamond] \rangle \big\}$$

Applying T_P to I_1 leads to I_2:

$$I_2 = T_P(I_1) = I_1 \cup \big\{ \langle q(3,3), [[3,4],[3,4]] \rangle, \langle q(4,4), [[3,4],[3,4]] \rangle \big\}$$

One further step leads to I_3, which is also the least fixpoint of T_P:

$$I_3 = T_P(I_2) = I_2 \cup \big\{ \langle p(3,3,3), [[3],[3],[3]] \rangle \big\}$$

The example above shows how LVLP on a domain $\mathcal{X} - LVLP(\mathcal{X})$ – looks like $CLP(\mathcal{X})$: in fact, $[\langle Y, [2,4] \rangle, \langle Z, [3,8] \rangle]$ can be interpreted as the constraints $Y \in [2,4], Z \in [3,8]$. However, this does not hold for all the label domains, since LVLP aims at covering domains beyond the reach of constraint logic programming.

This is the case, for instance, of Example 3.2, where labels are words in the WordNet lexical database [22]. There, the combining function f_L is supposed to find and return a WordNet synset compatible with both the given labels, or fail otherwise: for instance, if $\ell_1 = $ 'pet' and $\ell_2 = $ 'domestic cat', the new label generated by f_L could be ['cat', 'domestic cat', 'pet', 'mammal']. The compatibility function f_C is always true, since any animal name is considered compatible with any animal group.

Example 3.2. (T_P in the WordNet case)

In this example, labels are words describing the object represented by the variable. The combination of two different labels (performed by the combining function f_L) returns a new label only if the two labels have a

lexical relation, or fails otherwise. The decision is based on the WordNet network [22], a large lexical database of English where nouns, verbs, adjectives, and adverbs are grouped into sets of cognitive synonyms (synsets). Synsets are interlinked by means of conceptual-semantic and lexical relations: the resulting network of meaningfully related words and concepts can be navigated. WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings.

In this first example, some WordNet groups are collected and stored in the knowledge base a priori, but a dynamic consultation to WordNet could be implemented. Let the program P be represented by the following facts—where `wordnet_fact` is a simulated wordnet synset, while `animal(Name)` is a predicate computing the animal's name:

```
wordnet_fact(['dog','domestic dog','canis','pet','mammal']).
wordnet_fact(['cat', 'domestic cat', 'pet', 'mammal']).
wordnet_fact(['fish','aquatic vertebrates', 'vertebrate']).
wordnet_fact(['frog','toad', 'anuran', 'batrachian']).

pet_name('minnie').
fish_name('nemo').
animal(X) :- X^['pet'], pet_name(X).
animal(X) :- X^['fish'], fish_name(X).
```

The combining function f_L is supposed to find and return a WordNet synset compatible with both labels. So, if $\ell_1 = $ 'pet' and $\ell_2 = $ 'domestic cat', the new generated label ℓ_3 is ['cat', 'domestic cat', 'pet', 'mammal']. The compatibility function f_C in this scenario is always true, since any animal name is considered compatible with any animal group.

In order to show the construction of T_P, let us consider the interpretation $I_0 = \emptyset$. Then, the subsequent step I_1 can be computed as:

$$I_1 = T_P(I_0) = \{ \quad \langle pet\_name(\text{'}minnie\text{'}), [\diamond]\rangle, \langle fish\_name(\text{'}nemo\text{'})[\diamond]\rangle \}$$

Now, let us apply T_P to I_1 to compute I_2:

$$I_2 = T_P(I_1) = I_1 \cup \{$$
$$\langle animal(\text{'}minnie\text{'}), [[\text{'}dog\text{'}, \text{'}domestic\ dog\text{'}, \text{'}canis\text{'}, \text{'}pet\text{'}, \text{'}mammal\text{'}]]\rangle,$$
$$\langle animal(\text{'}minnie\text{'}), [[\text{'}cat\text{'}, \text{'}domestic\ cat\text{'}, \text{'}pet\text{'}, \text{'}mammal\text{'}]]\rangle,$$
$$\langle animal(\text{'}nemo\text{'}), [[\text{'}fish\text{'}, \text{'}aquatic\ vertebrates\text{'}, \text{'}vertebrate\text{'}]]\rangle \}$$

which is also the least fixpoint of T_P.

We prove that T_P, if applied bottom-up starting from the empty interpretation, always leads to a minimum fixpoint (Corollary 3.5). Such an interpretation is the denotational semantics of the program P. In order to achieve that result, we need to prove that T_P is monotonic and continuous, and use the Knaster-Tarksi and Kleene's fixpoint theorems (Proposition 3.4).

In order to be a model, an interpretation should satisfy the meaning of every rule—namely, if for a given valuation of the variables the body is considered true by the interpretation, then the head must also be true. We state this property formally:

Definition 3.3. An interpretation I is a model of a program P if, for each rule $r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}, b_1, \ldots, b_n$ of P and each valuation v of the variables in r on $\mathscr{H}$ (let us denote with $\widetilde{t} = v(\widetilde{x})$), it holds that if

- for $i = 1, \ldots, n$ there are $\langle v(b_i), \widetilde{\ell_i}\rangle \in I$
- such that $\mathscr{X} \models f_C(v(b_i), \widetilde{\ell_i})$ and
- $\mathscr{X} \models \Lambda_{\widetilde{t}}$

then it holds that $\langle p(\widetilde{t}), \widetilde{f}_L(r, \Lambda_{\widetilde{t}}, \widetilde{\ell}_1, \cdots, \widetilde{\ell}_n)\rangle \in I$.

Proposition 3.4. Given a LVLP program P and a LVLP domain $\mathcal{X}$, T_P is (1) monotonic and (2) (upward) continuous, and (3) $T_P(I) \subseteq I$ iff I is a model of P.

Proof:

(1) Let I and J be two interpretations such that $I \subseteq J$. We need to prove that $T_P(I) \subseteq T_P(J)$. If $a = \langle p(\widetilde{t}), \widetilde{\ell}\,\rangle \in T_P(I)$, there are a clause $r \in P$, a valuation v on $\mathcal{H}$ for the variables in r and n elements $\langle v(b_i), \widetilde{\ell}_i\rangle \in I$ satisfying the remaining conditions. Since $I \subseteq J$, they belong to J as well: so, $a \in T_P(J)$.

(2) Let us consider a chain of interpretations $I_0 \subseteq I_1 \subseteq \ldots$: we need to prove that $T_P\left(\bigcup_{k=0}^{\infty} I_k\right) = \bigcup_{k=0}^{\infty} T_P(I_k)$.

($\subseteq$) Let $a = \langle p(\widetilde{t}), \widetilde{\ell}\,\rangle \in T_P\left(\bigcup_{i=0}^{\infty} I_k\right)$. Thus, there are a clause $r \in P$, a valuation v on $\mathcal{H}$ for the variables in r, and n elements $\langle v(b_i), \widetilde{\ell}_i\rangle \in \bigcup_{k=0}^{\infty} I_k$ satisfying the remaining conditions. This means that there are $j_1, \ldots, j_n$ such that for $i = 1, \ldots, n$ $\langle v(b_i), \widetilde{\ell}_i\rangle \in I_{j_i}$. Now let $j = \max\{j_1, \ldots, j_n\}$: since $I_0 \subseteq I_1 \subseteq \cdots \subseteq I_j$, all $\langle v(b_i), \widetilde{\ell}_i\rangle \in I_j$. Thus $a \in T_P(I_{j+1})$ and henceforth $a \in \bigcup_{k=0}^{\infty} T_P(I_k)$.

($\supseteq$) Let $a = \langle p(\widetilde{t}), \widetilde{\ell}\,\rangle \in \bigcup_{k=0}^{\infty} T_P(I_k)$. This means that there is j such that $a \in T_P(I_j)$. Then, due to monotonicity, $a \in T_P\left(\bigcup_{k=0}^{\infty} I_k\right)$.

(3) Let $T_P(I) \subseteq I$ and let $r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}$, $b_1, \ldots, b_n$ be a generic clause of P, and v be a generic valuation on $\mathcal{H}$ of all the variables of r. If we assume that $\langle v(b_i), \widetilde{\ell}_i\rangle \in I$ for $i = 1, \ldots, n$, and that $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell}_i)$ and $\mathcal{X} \models \Lambda_{\widetilde{t}}$, then the pair $h = v(p(\widetilde{x}), \widetilde{\ell}) \in T_P(I)$ by definition of T_P—and $\widetilde{\ell}$ is the same of the definition of model. Since $T_P(I) \subseteq I$ then $h \in I$, therefore (since r and v are chosen in general) I is a model of P.

On the other hand, if I is a model of P we prove that $T_P(I) \subseteq I$. Let $a = \langle p(\widetilde{t}), \widetilde{\ell}\,\rangle \in T_P(I)$. This means that there is a rule $r = p(\widetilde{x}) \leftarrow \Lambda_{\widetilde{x}}$, $b_1, \ldots, b_n$ of P and a valuation v of the variables in r on $\mathcal{H}$ such that for $i = 1, \ldots, n$ there are $\left(v(b_i), \widetilde{\ell}_i\right) \in I$ and $\mathcal{X} \models \Lambda_{\widetilde{t}}$ and $\widetilde{\ell} = \widetilde{f}_L(r, \Lambda_{\widetilde{t}}, \widetilde{\ell}_1, \cdots, \widetilde{\ell}_n))$, such that $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell}_i)$. Since I is a model and $\mathcal{X} \models f_C(v(b_i), \widetilde{\ell}_i)$ then $a \in I$. $\qquad\square$

Let $T_P \uparrow \omega$ be defined as usual: $T_P \uparrow \omega = \bigcup\{T_P \uparrow k : k \geq 0\}$, where $T_P \uparrow 0 = \emptyset$ and $T_P \uparrow (n+1) = T_P(T_P \uparrow n)$. Then

Corollary 3.5. T_P has a least fixpoint.

Proof:

Being T_P monotonic, the Knaster-Tarksi theorem ensures that it admits a least (and a greatest) fixpoint. Being (upward) continuous, Kleene's fixpoint Theorem states that $T_P \uparrow \omega$ is the least fixpoint. $\qquad\square$

3.2. Operational semantics

In this section we define the operational interpretation of labels. Our approach is inspired by the methodology introduced for constraint logic programming (CLP) [23, 24, 25]: accordingly, we define the LVLP abstract state machine based on that suggested by Colmerauer for Prolog III [26]. We define a labelled-machine state σ as the triplet:

$$\sigma = \langle t_0 \ t_1 ... t_n, W, \Lambda\rangle$$

in which $t_0 \; t_1...t_n$ is the list of terms (goals), W is the current list of variable bindings, Λ is the current labelling on W.

An inference step for the machine consists of making a transition from the state σ to a state σ' by applying a program rule. If $m \geq 0$ and Λ' is a set of labelled variables, a program rule

$$s_0 \leftarrow \Lambda', s_1, s_2, \ldots, s_m$$

is applicable if the following conditions hold:

- $\exists \, mgu \; \theta$ such that $\theta(t_0) = \theta(s_0)$, and

- $\widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda')) \neq \emptyset$

Function $\widetilde{f}_{\theta L}$ is a generalisation of f_L taking as arguments two labellings. If x_i occurs in both Λ and $\theta(\Lambda')$ with labels ℓ and ℓ', $\ell'' = f_L(\ell, \ell')$ is first calculated, then the labelled variable $\langle x_i, \ell'' \rangle$ is returned, provided that $f_C(\theta(x_i), \ell'') = \text{true}$; otherwise, false is returned. Thus the new state becomes:

$$\sigma' = \langle \theta(s_1, \ldots, s_m, t_1, \ldots, t_n), W' = W \circ \theta, \Lambda'' = \widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda')) \rangle$$

where $\circ$ applies the classical composition of substitutions.

A solution is found when a final state is reached. The final state has the form:

$$\sigma_f = (\epsilon, W_f, \Lambda_f)$$

where ϵ is the empty sequence, W_f is the final list of variables and bindings, and Λ_f is the corresponding labelling. A sequence of applications of inference steps is said to be a *derivation*. A derivation is *successful* if it ends in a final state, or *failing* if it ends in a non-final state where no further inference step is possible.

Proposition 3.6. Let $p(\widetilde{x})$ be an atom, v a valuation on $\mathcal{H}$ such that $v(\widetilde{x}) = \widetilde{t}$ where $\widetilde{t}$ are ground terms, and $\widetilde{\ell}$ a list of labels. Then there is a successful derivation for $\langle p(\widetilde{t}), v, \langle v(\widetilde{x}), \widetilde{\ell} \rangle \rangle$ iff $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow \omega$.

Proof:
In the following we omit some standard details for the sake of brevity, please refer to, e.g., [25]

($\rightarrow$). We prove the proposition by induction on the length k of the derivation. If $k = 0$ the result holds trivially.

For the inductive case, let us suppose that there is a successful derivation for $\langle p(\widetilde{t}), v, \langle v(\widetilde{x}), \widetilde{\ell} \rangle \rangle$ of $k + 1$ steps. Let us focus on the first step: there is a rule $r: \; s_0 \leftarrow \Lambda', s_1, s_2, \ldots, s_m$ such that $\theta(p(\widetilde{t})) = \theta(s_0)$ leading to the new state $\sigma = \langle \theta(s_1, s_2, \ldots, s_m), v \circ \theta, \Lambda'' = \widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda')) \rangle$, where $f_C(\Lambda'') = \text{true}$, that admits a successful derivation of k steps.

Consider now the states $\sigma_1, \ldots, \sigma_m$ defined as $\sigma_i = \langle \theta(s_i), v \circ \theta, \Lambda''_i \rangle$ where Λ''_i is the restriction of Λ'' to the variables in s_i. Since σ admits a successful derivation of $k+1$ steps, each σ_i should admit a successful derivation of at most k steps.

If for all $i \in \{1, \ldots, m\}$, $\theta(s_i)$ is ground, then, by inductive hypothesis we have that $\langle \theta(s_i), \ell_i \rangle \in T_P \uparrow \omega$ where $\ell_i = \pi_2(\Lambda''_i)$, and hence that there are h_is such that $\langle \theta(s_i), \ell_i \rangle \in T_P \uparrow h_i$. Since T_p is monotonic, all of them belong to $T_P \uparrow h$ where $h = \max_{i=1,\ldots,m} h_i$. Then, by applying T_P

considering the rule r, since we already know that $\Lambda'' = \widetilde{f}_{\theta L}(\Lambda, \theta(\Lambda'))$, and $f_C(\Lambda'') = \text{true}$, we have that $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow h + 1$, hence to $T_P \uparrow \omega$.

If for some i, $\theta(s_i)$ is not ground, the above property holds for any ground instantiation of the remaining variables and again the results follows.

($\leftarrow$). Now, let us analyse the converse direction. If $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow \omega$ this means that there is a $k \geq 0$ such that $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow k$.

Let us prove by induction on k. Again, if $k = 0$ the result holds trivially. Let us suppose now that $\langle p(\widetilde{t}), \widetilde{\ell} \rangle \in T_P \uparrow k + 1$. This means (by definition of T_P) that there is a rule r: $s_0 \leftarrow \Lambda', s_1, s_2, \ldots, s_m$ such that $s_0 = p(\widetilde{x})$ and there is a valuation u on $\mathcal{H}$ such that $u(s_0) = p(\widetilde{t})$ and that, in particular, $\langle s_1, \ell_1 \rangle, \ldots, \langle s_m, \ell_m \rangle \in T_P \uparrow k$ (and f_L can be computed and f_C is true on these arguments). By inductive hypothesis, for $i \in \{1, \ldots, m\}$ there is a derivation, say, of h_i steps for $\sigma_i = \langle u(s_i), v \circ u, \langle u(s_i), \ell_i \rangle \rangle$

Since f_C is true on such arguments and f_L can be computed, the same holds for T_P: so, we have a derivation of $\sum_{i=1}^{m} h_i + 1$ steps for $\langle p(\widetilde{t}), v \circ u \circ \theta, \langle \widetilde{t}, \widetilde{\ell} \rangle \rangle$.

This completes the proof of the inductive step. $\qquad\square$

4. Meta-interpreter

Listing 1. *The LVLP meta-interpreter: the* `solve/3` *predicate.*

```
%%%% solve(+Goals, +LVarsIn, -LVarsOut)
%% Goals is the list of goals to solve
%% LVarsIn is the labelling on goals variables
%% LVarsOut is the final labelling on output variables
% termination condition
solve([], LVars, LVars) :- !. % for efficiency
% goal iterator
solve([Goal|Goals], LVarsIn, LVarsOut):- !,
  solve(Goal, LVarsIn, LVarsTempOut),
  solve(Goals, LVarsTempOut, LVarsOut).
% solve core
solve(Goal, LVarsIn, LVarsOut):-
  clause(Goal, LVars, Body),
  mergeLabels(LVarsIn, LVars, LVarsTempOut),
  solve(Body, LVarsTempOut, LVarsOut).
```

Listing 2. *The LVLP meta-interpreter: the* `mergeLabels/3` *predicate.*

```
%%%% mergeLabels(+LVars1, +LVars2, -LVars3)
%% LVars1, LVars2, LVars3 are lists of labelled variables
%% LVars3 is obtained merging the labelled variables in LVars1 and LVars2
%% LVars1 and LVars2 are sorted according to the same criterion--e.g., alphabetically
%% For all the variables that appear both in LVars1 and LVars2, the resulting label
%% is obtained using the combining function embedded in label_generate/3
% termination conditions
mergeLabels(LVars, [], LVars) :- !.
mergeLabels([], LVars, LVars) :- !.
% variable Var1 propagation in LVars3 if it is contained only in LVars1
mergeLabels([Var1^L1|LVars1], LVars2, [Var1^L1|LVars3]):-
  not_in(Var1,LVars2), !,
  mergeLabels(LVars1, LVars2, LVars3).
% variable Var2 propagation in LVars3 if it is contained only in LVars2
mergeLabels(LVars1, [Var2^L2|LVars2], [Var2^L2|LVars3]):-
  not_in(Var2,LVars1), !,
  mergeLabels(LVars1, LVars2, LVars3).
% generation of a new label if variable Var is contained both in LVars1 and LVars2
mergeLabels([Var^L1|LVars1], [Var^L2|LVars2], [Var^L3|LVars3]):-
  label_generate(L1, L2, L3),
  mergeLabels(LVars1, LVars2, LVars3).
% utility func not_in(+Var, +LVars) checks if the list LVars does not contain Var
```

```
not_in(_,[]).
not_in(X,[Y^_|_]) :- !, fail.
not_in(X,[_|T]) :- not_in(X,T).
```

The operational semantics of LVLP is captured by the meta-interpreter shown in Listing 1: the code is developed in tuProlog [27], a light-weight Prolog system whose (Java-based) design inherently enables the injection of Prolog programs within pervasive systems, as well as their integration with diverse programming languages and paradigms, over computing platforms of any sort [20].

The `solve/3` predicate[1] has three arguments (Listing 1), namely:

- the list of the goals to be processed

- the current labelling

- the new labelling updated by the goal resolution process

The `solve/3` predicate recursively calls itself to process the goal list, and exploits `clause/3` and `mergeLabels/3` to, respectively, handle single goals and combine labels: in particular, `clause/3` finds a clause in the database whose head matches with `Goal` and returns both the `Body` of the clause and the labelling in the selected clause, `LVars`.

The core of the meta-interpreter is embedded in the `mergeLabels/3` predicate (detailed in Listing 2), which combines two sets of labels – the previous labelling, `LVarsIn`, and the labelling introduced by the current clause, `LVars` – into the new `LVarsTempOut`, or fails if no solution can be found. The generation of the new label is performed via the `label_generate/3` predicate, which embeds the combining function f_L, and is provided by the user according to the domain-specific features of the application scenario.

5. Case studies

In the following we discuss some LVLP computations based on our prototype rooted in Labelled tuProlog [29], which exploits the meta-interpreter presented in Section 4—available on Bitbucket [30].

5.1. WordNet network

This example extends and implements the case study of Example 3.2. A label is a network of related words describing the semantic net of the object represented by the associated variable according to the WordNet lexical database [22]. The listing in Figure 1 shows the tuProlog implementation of the `label_generate/3` predicate embedding the combining function f_L. Here, the `label_generate` predicate checks if ℓ_1 and ℓ_2 are contained in a common `wordnet_fact`.

Following our prototype syntax, `X^ label` is a labelled variable denoting a logic variable `X` labelled with `label`—where `label` is a term in the set of admissible labels defined by the user. In an LVLP clause, the list of the labelled variables precedes the remaining part of the body. So, given the program:

```
wordnet_fact(['dog','domestic dog','canis','pet','mammal','vertebrate']).
wordnet_fact(['cat', 'domestic cat', 'pet', 'mammal', 'vertebrate']).
wordnet_fact(['fish', 'aquatic vertebrates', 'vertebrate']).
wordnet_fact(['frog', 'toad', 'anuran', 'batrachian']).
animal(X) :- X^['pet'], X = 'minnie'.
```

---

[1] `solve/3` is designed according to the standard Prolog meta-interpreter [28]

```
animal(X) :- X^['fish'], X = 'nemo'.
animal(X) :- X^['cat'], X = 'molly'.
animal(X) :- X^['dog'], X = 'frida'.
animal(X) :- X^['frog'], X = 'cra'.
```

the following query, looking for a `pet` animal, generates four solutions:

```
?- X^['pet'], animal(X).

yes. X / 'minnie'
X^['dog', 'domestic dog', 'canis', 'pet', 'mammal', 'vertebrate'];

yes. X / 'minnie'
X^['cat', 'domestic cat', 'pet', 'mammal', 'vertebrate'];

yes. X / 'molly'
X^['cat', 'domestic cat', 'pet', 'mammal', 'vertebrate'];

yes. X / 'frida'
X^['dog', 'domestic dog', 'canis', 'pet', 'mammal', 'vertebrate']
```

Looking instead for a less specific `vertebrate` produces five solutions:

```
?- X^['vertebrate'], animal(X).

yes. X = 'minnie'
X^['dog', 'domestic dog', 'canis', 'pet', 'mammal', 'vertebrate'] ;

yes. X = 'minnie'
X^['cat', 'domestic cat', 'pet', 'mammal', 'vertebrate'] ;

yes. X = 'molly'
X^['cat', 'domestic cat', 'pet', 'mammal', 'vertebrate'] ;

yes. X = 'frida'
```

```
%% label_generate(+L1, +L2, -L3) embedding f_L behaviour for WordNet groups
label_generate(L1, L2, List):-
    wordnet_fact(List), sublist(L1, List), sublist(L2, List).
```
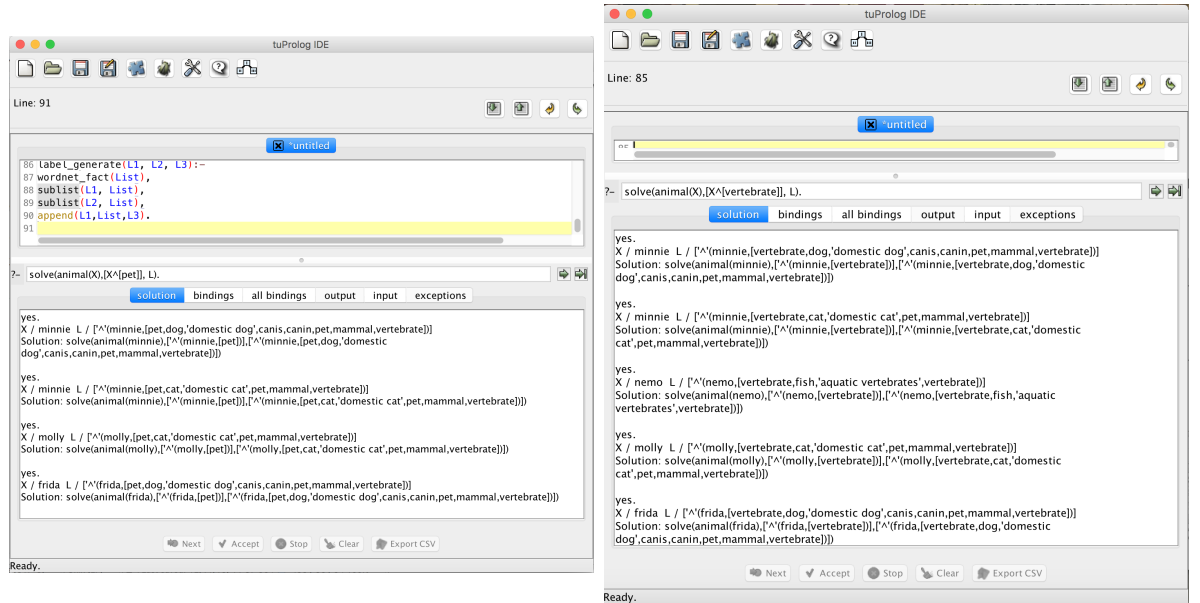

Figure 1.   label_generate/3 example: WordNet case study

```
X^['dog', 'domestic dog', 'canis', 'pet', 'mammal', 'vertebrate'] ;

yes. X = 'nemo'
X^['fish','aquatic vertebrates', 'vertebrate']
```

A relevant aspect of LVLP is that labels are not subject to the single-assignment assumption: each time two labelled variables unify, their labels are processed and *combined* according to the user-defined function that embeds the desired computational model, and the resulting label is associated to the unified variable. Thus, while the LP model *per se* is left untouched, diverse computational models can be associated to it, possibly influencing the result of a logic computation by *restricting* the set of admissible solutions according to each specific domain.

5.2. Dress selection

In the following example the application scenario is the selection from a wardrobe of a dress that is "similar enough" to a given colour. A fact `shirt(Description, Colour)` represents a shirt of colour *Colour*, expressed as a triple of the form `rgb(Red,Green,Blue)` in *Description*.

For instance, shirts in a wardrobe could be:
```
○  shirt(rgb(255,240,245), my_pink_blouse).
○  shirt(rgb(255,222,173), old_yellow_tshirt).
●  shirt(rgb(119,136,153), army_tshirt).
●  shirt(rgb(188,143,143), periwinkle_blouse).
○  shirt(rgb(255,245,238), fashion_cream_blouse).
```

Without any colour constraints, the following query would return all the shirts in the wardrobe:
```
?- [], shirt(Description, Colour).
```

Instead, by defining a *target colour* in the goal via labelled variables, the query can be refined in order to get only those dresses whose *dress_colour* is "similar enough" to the target—with similarity embedded through a suitably-defined combining function f_L.

In our example, two colours are considered as similar if their distance is below a given threshold. Thus, during the unification of labelled variables, if the *dress_colour* is similar to the *target_colour*, the returned label is *dress_colour* (that is, the colour of the selected shirt); otherwise, the empty label is returned, so unification fails.

As a first step, we assume that a colour c is represented as RGB ($c = rgb(r, g, b)$), the threshold is ≤ 30, and the colour distance is normalised and computed as a distance in a 3D Euclidean space. For instance, let us look for all the shirts similar to the papaya colour ○ through the following query, where `Colour` is labelled according to the papaya RGB triple (255, 239, 213):

```
?- Colour^[rgb(255,239,213)], shirt(Description, Colour).

yes. Description / my_pink_blouse, Colour / rgb(255,240,245)
Colour^[rgb(255,239,213)];

yes. Description / old_yellow_tshirt, Colour / rgb(255,222,173)
Colour^[rgb(255,239,213)];

yes. Description / fashion_cream_blouse, Colour / rgb(255,245,238)
Colour^[rgb(255,239,213)]
```

since the normalised distances d_N are:

d_N(○ $papaya = rgb(255, 239, 213)$, ○ $lightpink = rgb(255, 240, 245)$) = $\boxed{7.25}$
d_N(○ $papaya = rgb(255, 239, 213)$, ○ $lightyellow = rgb(255, 222, 173)$) = $\boxed{9.84}$
d_N(○ $papaya = rgb(255, 239, 213)$, ● $armyblue = rgb(119, 136, 153)$) = 40.95
d_N(○ $papaya = rgb(255, 239, 213)$, ● $periwinkle = rgb(188, 143, 143)$) = 30.88
d_N(○ $papaya = rgb(255, 239, 213)$, ○ $creamwhite = rgb(255, 245, 238)$) = $\boxed{5.82}$

```
%% label_generate(+L1, +L2, -L3) embedding f_L behaviour for integer intervals
label_generate([H1|([T1|T11])], [H2|([T2|T22])], [H3|T3]):-
  ((H1>H2) -> H3=H1; H3=H2), ((T1<T2) -> T3=T1; T3=T2), H3<T3.
```
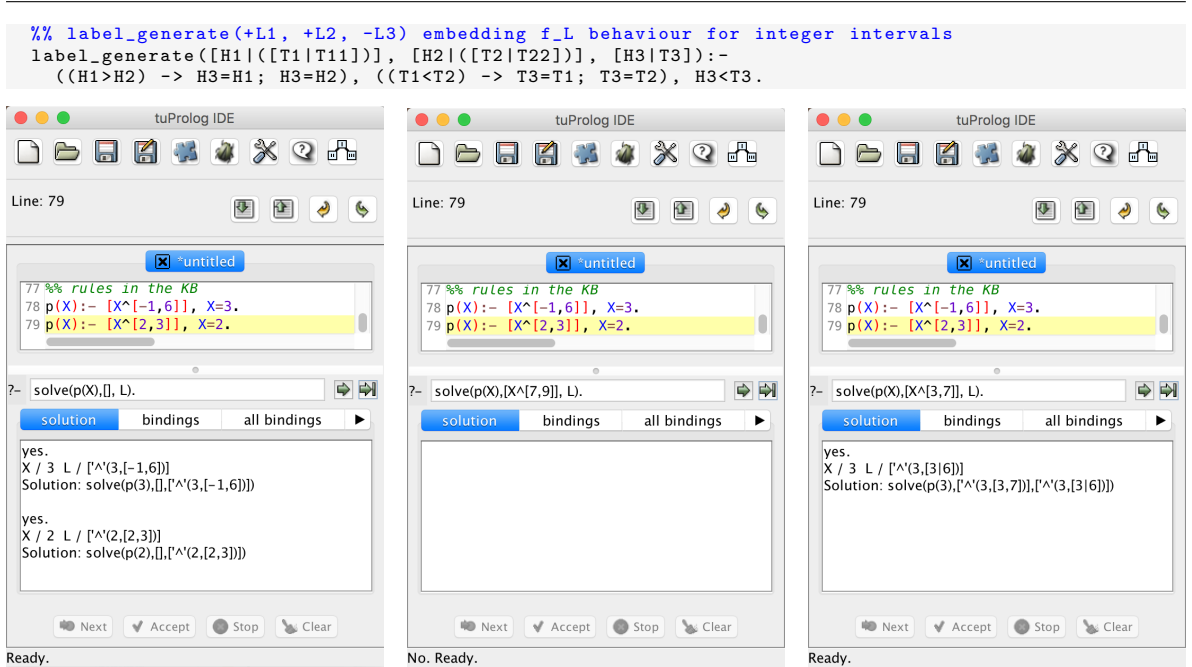

Figure 2. `label_generate/3` example: numeric interval intersection

Going one step further, the label can be enriched with the neighbourhood information (i.e., the admissible threshold), thus allowing the user to dynamically change the similarity criterion. For instance, the same query as the one in Listing 5.2 could be expressed as:

```
?- Colour^[rgb(255,239,213), d = 30], shirt(Description, Colour).
```

whereas a stricter constraint could be imposed by the following query:

```
?- Colour^[rgb(255,239,213), d = 6], shirt(Description, Colour).

yes. Description / fashion_cream_blouse, Colour / rgb(255,245,238)
Colour^[rgb(255,239,213), d = 6]
```

Once again, while LP is left untouched, LVLP captures a parallel computation on the domain of interest, which affects the final result.

5.3. Integer intervals

Standard domains for logic languages – including the CLP ones [23] – are also supported. For instance, labels could be used to represent the integer interval over which the logic variable values span: accordingly, the label syntax could take the form $X^[min,max]$, and a simple interval program could look like:

```
interval(X):- X^[-1,4].
interval(X):- X^[6,10].
```

The unification of two variables labelled with an interval would then result in a variable labelled with the intersection of the intervals. Accordingly, the following simple query generates two solutions:

```
?- X^[2,7], interval(X).

yes.
X^[2,4] ;

yes.
X^[6,7]
```

However, the expressiveness of LVLP makes it possible to easily move from the domain of integer intervals to more articulated domains, thus going beyond the reach of constraint logic languages.

For instance, the above example could be easily extended to the domain of *integers with a neighbourhood*, as in the following program:

```
neighbourhood(X):- X^[-1,4], X=3.
neighbourhood(X):- X^[6,10], X=8.
```

There, constant values unify with labelled variables if they belong to the interval in the label. Accordingly, the following query would result in just one solution:

```
?- X^[2,7], neighbourhood(X).

yes.  X / 3
X^[2,4]
```

because the second clause would set X out of the interval specified in the query.

6.   Related Work

Surveying the literature reveals a large number of diverse proposals pushing computational logic towards *distributed situated intelligence* [31] in pervasive systems—to exploit domain knowledge, understand local context, and share information in support of intelligent applications and services [32, 33]. There, systems are expected to respond intelligently to contextual information about the physical world acquired via sensors and information about the computational environment.

The declarative approach and the explicit knowledge representation of LP enable knowledge sharing at the most adequate level of abstraction while supporting modularity and separation of concerns [34], which are especially valuable in open and dynamic distributed systems (*serendipitous interoperability*, [35]). As a further element, LP formal semantics naturally enables logic-based intelligent agents to reason and infer new information.

Many languages extension have been proposed in order to allow intelligent agents to interact with the environments and deal with specific situation, highlighting the benefits of LP for reasoning in pervasive systems. XLOG [36] is a hybrid programming environment where predicate logic is integrated into an object-oriented computational model, specially adequate for working with reactive agents to enable the principles of emergence and situatedness. Along this line, CIFF [37] is a system implementing a novel extension of Fung and Kowalski's IFF abductive proof procedure [38] aimed at building intelligent agents that can construct plans and react to changes in the environment. The proposed solution improves on more conventional abductive theories for planning by adding the possibility to interact with the environment, by observing environment properties as well as actions executed by other agents, thus enhancing agent situatedness.

Moreover, many researches exploit LP extensions to model context and situations. In the works by Ranganathan and Campbell [39] and Katsiri and Mycroft [40], first-order logic (FOL) is used for representing and reasoning with context, whereas Henricksen [41] exploits FOL to describe and reason with situations. On the other hand, the above approach do not adopt a modular approach or meta-reasoning as in [42], where an extension of Prolog (LogicCAP) is presented: the notion of *situation*

program is introduced, thus highlighting the primacy of the situation issue for building context-aware pervasive systems.

Orthogonally, Labelled Deductive Systems (LDS) have been proposed for providing logics from different families with a uniform presentation of their derivability relations and semantic entailments to deal with domain-specific situations [3]. The main idea there is to provide a new unifying methodology, replacing the traditional view of logic, manipulating sets of formulas by the notion of structured families of labelled formulas. Detailed investigations have been undertaken to explore the benefits of using the LDS methodology to reformulate intuitionistic modal logics [43] and substructural logics [44, 45]. Specialised frameworks based on LDS have been also proposed [46, 47, 48]. Among the others, the Compiled Labelled Deductive Systems (CLDS) approach demonstrated how LDS techniques facilitate the reformulation and generalisation of a large class of modal logics and conditional logics [48, 49].

Our work builds upon the general notion of *label* as defined by Gabbay [3], and adopts the techniques introduced by Holzbaur [4] to develop a generalisation of LP where labels are exploited to define computations in domain-specific contexts. Our characterisation can be viewed as a generalisation of the aforementioned approaches, blending the benefits of labels and LP so as to enable the very intrinsic nature of distributed situated intelligence. Indeed, LVLP allows heterogeneous devices in the IoT to have specific application goals and manage specific sorts of information, enabling reactivity to environment change while capturing diverse logic and domains.

7.   Conclusions & Future Work

The primary results of this paper is the definition of the LVLP theoretical framework, where different domain-specific computational models can be expressed via labelled variables, capturing suitably-tailored labelled models. The framework is aimed at extending LP to face the challenges of today pervasive systems, by providing the models and technologies required to effectively support distributed situated intelligence, while preserving the features of declarative programming. We present the fixpoint and operational semantics, discuss correctness, completeness, and equivalence, and test the effectiveness of our approach through some case studies.

While the first LVLP prototype [29] is currently implemented over tuProlog [27] via the described meta-interpreter, the full integration of the LVLP model in the tuProlog code is currently in advanced stage of development.

The next stage is represented by the design and implementation of a full-fledged logic-based *middleware* for LVLP, which could be exploited to test the effectiveness of LVLP in real-world pervasive intelligence scenarios. As far as the formal aspects are concerned, future work will be devoted to deeper exploration and better understanding of the consequences of applying labels to formulas, as suggested by Gabbay [3]. Other research lines will possibly include the application of the LVLP framework to different scenarios and approaches—such as probabilistic LP [9], the many CLP approaches [23], distributed ASP reasoning [50], and action languages [51].

References

[1] Mariani S, Omicini A. Coordinating activities and change: An event-driven architecture for situated MAS. Engineering Applications of Artificial Intelligence. 2015 May;41:298–309. doi:10.1016/j.engappai.2014.10.006.

[2] Maes P. Situated agents can have goals. Robotics and Autonomous Systems. 1990;6(1):49–70. doi:10.1016/S0921-8890(05)80028-4.

[3] Gabbay DM. Labelled Deductive Systems, Volume 1. vol. 33 of Oxford Logic Guides. Clarendon Press; 1996. Available from: `http://global.oup.com/academic/product/labelled-deductive-systems-9780198538332`.

[4] Holzbaur C. Metastructures vs. attributed variables in the context of extensible unification. In: Bruynooghe M, Wirsing M, editors. Programming Language Implementation and Logic Programming. vol. 631 of Lecture Notes in Computer Science. Springer; 1992. p. 260–268. doi:10.1007/3-540-55844-6_141.

[5] Calegari R, Denti E, Dovier A, Omicini A. Labelled Variables in Logic Programming: Foundations. In: Fiorentini C, Momigliano A, editors. CILC 2016 – Italian Conference on Computational Logic. Proceedings of the 31st Italian Conference on Computational Logic. vol. 1645 of CEUR Workshop Proceedings. Milano, Italy; 2016. p. 5–20. Available from: `http://ceur-ws.org/Vol-1645/paper_7.pdf`.

[6] Alsinet T, Chesñevar CI, Godo L, Simari GR. A logic programming framework for possibilistic argumentation: formalization and logical properties. Fuzzy Sets and Systems. 2008;159(10):1208–1228. doi:10.1016/j.fss.2007.12.013.

[7] Barany V, ten Cate B, Kimelfeld B, Olteanu D, Vagena Z. Declarative Probabilistic Programming with Datalog. In: Martens W, Zeume T, editors. 19th International Conference on Database Theory (ICDT 2016). vol. 48 of Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2016. p. 7:1–7:19. doi:10.4230/LIPIcs.ICDT.2016.7.

[8] Russo A. Generalising Propositional Modal Logic Using Labelled Deductive Systems. In: Baader F, Schulz KU, editors. Frontiers of Combining Systems. vol. 3 of Applied Logic Series. Springer; 1996. p. 57–73. doi:10.1007/978-94-009-0349-4_2.

[9] Skarlatidis A, Artikis A, Filippou J, Paliouras G. A Probabilistic Logic Programming Event Calculus. Theory and Practice of Logic Programming. 2015 Mar;15(2):213–245. Special Issue on Probability, Logic and Learning. doi:10.1017/S1471068413000690.

[10] Hofstedt P. Multiparadigm Constraint Programming Languages. Cognitive Technologies. Springer; 2011. doi:10.1007/978-3-642-17330-1.

[11] Kifer M, Subrahmanian VS. Theory of generalized annotated logic programming and its applications. The Journal of Logic Programming. 1992 Apr;12(4):335—367. doi:10.1016/0743-1066(92)90007-P.

[12] Kowalski R. Logic Programming. In: Mason REA, editor. Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23. North-Holland/IFIP; 1983. p. 133–145.

[13] Bramer M. Logic Programming with Prolog. 2nd ed. Springer; 2013. doi:10.1007/978-1-4471-5487-7.

[14] Castelfranchi C. Modelling social action for AI agents. Artificial Intelligence. 1998 Aug;103(1-2):157–182. doi:10.1016/S0004-3702(98)00056-3.

[15] Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems. 2013 Sep;29(7):1645–1660. doi:10.1016/j.future.2013.01.010.

[16] Atzori L, Iera A, Morabito G. The Internet of Things: A survey. Computer Networks. 2010 Oct;54(15):2787–2805. doi:10.1016/j.comnet.2010.05.010.

[17] Fortino G, Guerrieri A, Russo W, Savaglio C. Integration of agent-based and Cloud Computing for the smart objects-oriented IoT. In: IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2014). Hsinchu, Taiwan: IEEE; 2014. p. 493–498. doi:10.1109/CSCWD.2014.6846894.

[18] Lippi M, Mamei M, Mariani S, Zambonelli F. Coordinating Distributed Speaking Objects. In: 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017). Atlanta, GA, USA: IEEE Computer Society; 2017. p. 1949–1960. doi:10.1109/ICDCS.2017.282.

[19] Arsénio A, Serra H, Francisco R, Nabais F, Andrade J, Serrano E. Internet of Intelligent Things: Bringing Artificial Intelligence into Things and Communication Networks. In: Inter-cooperative Collective Intelligence: Techniques and Applications. vol. 495 of Studies in Computational Intelligence. Springer; 2014. p. 1–37. doi:10.1007/978-3-642-35016-0_1.

[20] Denti E, Omicini A, Calegari R. tuProlog: Making Prolog Ubiquitous. ALP Newsletter. 2013;Oct. Available from: http://www.cs.nmsu.edu/ALP/2013/10/tuprolog-making-prolog-ubiquitous/.

[21] Robinson JA. A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM. 1965 Jan;12(1):23–41. doi:10.1145/321250.321253.

[22] Fellbaum C. WordNet(s). In: Brown K, editor. Encyclopedia of Language and Linguistics. vol. 13. 2nd ed. Elsevier; 2006. p. 665–670.

[23] Cohen J. Constraint Logic Programming Languages. Communications of the ACM. 1990 Jul;33(7):52–68. doi:10.1145/79204.79209.

[24] Imbert JL, Cohen J, Weeger MD. An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-Interpreter for CLP. The Journal of Logic Programming. 1993;16(3):235–253. doi:10.1016/0743-1066(93)90044-H.

[25] Jaffar J, Maher MJ. Constraint logic programming: a survey. The Journal of Logic Programming. 1994 May–Jul;19–20, Supplement 1:503–581. Special Issue: Ten Years of Logic Programming. doi:10.1016/0743-1066(94)90033-7.

[26] Colmerauer A. An Introduction to Prolog III. In: Lloyd JW, editor. Computational Logic. Symposium Proceedings, Brussels, November 13/14, 1990. ESPRIT Basic Research Series. Springer; 1990. p. 37–79. doi:10.1007/978-3-642-76274-1_2.

[27] Denti E, Omicini A, Ricci A. Multi-paradigm Java-Prolog Integration in tuProlog. Science of Computer Programming. 2005 Aug;57(2):217–250. doi:10.1016/j.scico.2005.02.001.

[28] Sterling L, Shapiro EY, Warren DHD. The Art of Prolog. Advanced Programming Techniques. vol. 1994. MIT Press; 1986. Available from: http://mitpress.mit.edu/books/art-prolog.

[29] Calegari R, Denti E, Omicini A. Labelled Variables in Logic Programming: A First Prototype in tuProlog. In: Bellodi E, Bonfietti A, editors. AI*IA 2015 DC Proceedings. vol. 1485 of CEUR Workshop Proceedings. Ferrara, Italy: AI*IA; 2015. p. 25–30. Available from: http://ceur-ws.org/Vol-1485/paper5.pdf.

[30] tuProlog. Home Page [Web Site]. Università di Bologna, Italy; 2017. Available from: http://tuprolog.unibo.it.

[31] Parker LE. Distributed Intelligence: Overview of the Field and its application in Multi-robot Systems. Journal of Physical Agents. 2008;2(1):5–14. doi:10.14198/JoPha.2008.2.1.02.

[32] Chen H, Finin T, Joshi A. An Ontology for Context-Aware Pervasive Computing Environments. The Knowledge Engineering Review. 2003 Sep;18(3):197–207. doi:10.1017/S0269888904000025.

[33] Smart P. Situating Machine Intelligence Within the Cognitive Ecology of the Internet. Minds and Machines. 2017 Jun;27(2):357–380. doi:10.1007/s11023-016-9416-z.

[34] Oliya M, Pung HK. Towards Incremental Reasoning for Context Aware Systems. In: Abraham A, Lloret Mauri J, Buford JF, Suzuki J, Thampi SM, editors. Advances in Computing and Communications: First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings, Part I. vol. 190 of Communications in Computer and Information Science. Springer; 2011. p. 232–241. doi:10.1007/978-3-642-22709-7_24.

[35] Niezen G. Ontologies for interaction: Enabling serendipitous interoperability in smart environments. Journal of Ambient Intelligence and Smart Environments. 2013 Jan;5(1):135–137. doi:10.3233/AIS-120194.

[36] Feijó B, Bento J. A logic-based environment for reactive agents in intelligent CAD systems. Advances in Engineering Software. 1998;29(10):825–832. doi:10.1016/S0965-9978(97)00066-5.

[37] Endriss U, Mancarella P, Sadri F, Terreni G, Toni F. Abductive Logic Programming with CIFF: System Description. In: Alferes JJ, Leite J, editors. Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings. vol. 3229 of Lecture Notes in Computer Science. Springer; 2004. p. 680–684. doi:10.1007/978-3-540-30227-8_56.

[38] Fung TH, Kowalski R. The IFF proof procedure for abductive logic programming. The Journal of Logic Programming. 1997;33(2):151–165. doi:10.1016/S0743-1066(97)00026-5.

[39] Ranganathan A, Campbell RH. An Infrastructure for Context-awareness Based on First Order Logic. Personal and Ubiquitous Computing. 2003 Dec;7(6):353–364. doi:10.1007/s00779-003-0251-x.

[40] Katsiri E, Mycroft A. Knowledge-Representation and Scalable Abstract Reasoning for Sentient Computing using First-Order Logic. In: Colton S, Gow J, Sorge V, Walsh T, editors. 1st Workshop on Challenges and Novel Applications for Automated Reasoning. CADE-19, Miami, FL, USA; 2003. p. 73–87.

[41] Henricksen K, Indulska J, Rakotonirainy A. Modeling Context Information in Pervasive Computing Systems. In: Pervasive Computing. vol. 2414 of Lecture Notes in Computer Science. Springer; 2002. p. 167–180. doi:10.1007/3-540-45866-2_14.

[42] Loke SW. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. The Knowledge Engineering Review. 2004 Sep;19(3):213–233. doi:10.1017/S0269888905000263.

[43] Sympson AK. The Proof Theory and Semantics of Intuitionistic Modal Logics [PhD Thesis]. University of Edinburgh, UK; 1994.

[44] Broda K, Finger M, Russo A. Labelled natural deduction for substructural logics. Logic Journal of the IGPL. 1999;7(3):283–318. doi:10.1093/jigpal/7.3.283.

[45] D'Agostino M, Gabbay DM, Broda K. Tableau Methods for Substructural Logics. In: D'Agostino M, Gabbay DM, Hähnle R, Posegga J, editors. Handbook of Tableau Methods. Dordrecht: Springer Netherlands; 1999. p. 397–467. doi:10.1007/978-94-017-1754-0_7.

[46] Artosi A, Governatori G, Rotolo A. Labelled Tableaux for Nonmonotonic Reasoning: Cumulative Consequence Relations. Journal of Logic and Computation. 2002 Dec;12(6):1027–1060. doi:10.1093/logcom/12.6.1027.

[47] Blackburn P. Internalizing labelled deduction. Journal of Logic and Computation. 2000;10(1):137–168. doi:10.1093/logcom/10.1.137.

[48] Russo AM. Modal Labelled Deductive Systems [PhD Thesis]. Department of Computing, Imperial College London, UK; 1996.

[49] Broda K, Gabbay DM, Lamb LC, Russo A. Labelled Natural Deduction for Conditional Logics of Normality. Logic Journal of the IGPL. 2002;10(2):123–163. doi:10.1093/jigpal/10.2.123.

[50] Dovier A, Pontelli E. Present and Future Challenges for ASP Systems. In: Erdem E, Lin F, Schaub T, editors. Logic Programming and Nonmonotonic Reasoning. 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. vol. 5753 of Lecture Notes in Computer Science. Springer; 2009. p. 622–624. doi:10.1007/978-3-642-04238-6_70.

[51] Dovier A, Formisano A, Pontelli E. Autonomous Agents Coordination: Action Languages Meet CLP(FD) and Linda. Theory and Practice of Logic Programming. 2013 Sep;13(2):149–173. doi:10.1017/S1471068411000615.