Labelled Variables in Logic Programming: A First Prototype in tuProlog

Roberta Calegari, Enrico Denti, and Andrea Omicini

Dipartimento di Informatica, Scienza e Ingegneria (DISI)
Alma Mater Studiorum–Università di Bologna, Italy
{roberta.calegari, enrico.denti, andrea.omicini}@unibo.it

Abstract. We present the first prototype of Labelled tuProlog, an extension of tuProlog exploiting labelled variables to enable a sort of multi-paradigm / multi-language programming aimed at pervasive systems.

1 Context & Motivation

To face the challenges of today pervasive system, which are inherently *complex*, *distributed*, *situated* [7] and *intelligent*, suitable models and technologies are required to effectively support *distributed situated intelligence*. To this end, in this paper we investigate a logic programming (LP) extension based on *labelled variables*, and test it by means of a prototype rooted in the tuProlog system [3, 9], called *Labelled* tuProlog. Our general aim is to define a unique blend of LP and labelled variables – a sort of multi-paradigm and multi-language programming framework for a distributed environment – where diverse computational models can be tailored to the *local* needs of situated systems by means of suitable labelled variables systems, and work together against the common LP background according to the original tuProlog aim [3].

Our work builds upon the general notion of *label* defined by Gabbay [4], and adopts the techniques introduced by Holzbaur [6] to develop a generalisation of LP where labels are exploited to define computations in domain-specific contexts, while retaining the conventional syntax of LP. More generally, our work moves from the observation that pervasiveness of today systems requires awareness of the environment as well as the ability of reacting to changes. This mandates for models promoting system intelligence, and for technologies making it possible to spread intelligence wherever needed. While logic-based approaches are natural candidates for intelligent systems, a pure LP approach seems not to fit the needs of situated systems. Instead, a hybrid approach would make it possible to exploit LP for what it is most suited – symbolic computation –, while possibly delegating other aspects (such as situated computations) to other languages, or to other levels of computation. This is precisely where the notion of labelled variables can be of help: by enabling some parts of the computation to be expressed at a separate level, while retaining the general coherence of the LP approach.

Being light-weight, intentionally designed around a minimal core, and Java-based, tuProlog is an ideal candidate to support the above goal—distributing situated intelligence while supporting labelled-variable-based computations.

While several works exists in this area – such as [8, 1] – they mostly focus onto specific scenarios and sorts of systems—e.g. modal logic, deductive systems, fuzzy systems, etc. Instead, our model aims at providing a general-purpose mechanism that could fit several relevant context-specific systems, while seamlessly rooted in the LP framework.

While in our first prototype labels are only applied to variables – and not generally to formulas like in Gabbay [4] – the proposed notion of label is already general enough to capture Holzbaur's attribute variables [6], opening the way towards the expressiveness and the computational power of Constraint Logic Programming (CLP) [10, 5]—which is why the two case studies presented below assume CLP as the reference label domain.

2 The Labelled Variables Model in Logic Programming

A *Labelled Variables Model for Logic Programming* is defined by:

- a set of *basic labels* $b_1, \ldots, b_n$, each with the form of a logic term, which represent entities in the *domain of labels*;
- a set of *labels*, each defined as a set of basic labels—i.e., $l_i = \{b_{1i}, .., b_{ni}\}$;
- a *labelling association* $\langle v, l \rangle$, associating label l to variable v: as a convenient shortcut, such association can be written as v^l;
- a *combining function* f_L, synthesising a new label from two given ones, combining the two labels according to some scenario-specific criteria.

The unification of two labelled variables is represented by the extended tuple *(true/false, θ, label)* where *true/false* represents the existence of an answer, θ represents the most general substitution, and *label* represents the new label associated to the unified variables defined by the *combining function* f_L. Figure 1 reports the unification rules for two generic terms T_1 and T_2: to lighten the notation, undefined elements in the tuple are omitted—i.e., labels or substitutions that do not exist/do not apply in a particular situation. Since, by design, only variables can be labelled here, the only case to be added to the standard unification table is represented by labelled variables.

Fig. 1. Unification rules summary

T_2 \ T_1	constant C_2	variable X_2	labelled variable X_2^{l2}	compund term S_2
constant C_1	true if $C_1 = C_2$	true - $\{X_2/C_1\}$	false	false
variable X_1	true - $\{X_1/C_2\}$	true - $\{X_1/X_2\}$	true - $\{X_1/X_2\} - l_2$	true - $\{X_1/S_2\}$
labelled variable X_1^{l1}	false	true - $\{X_1/X_2\}$ - l_1	true if not $\{\}$ - $\{X_1/X_2\} - f_L(l_1, l_2)$	false
compund term S_1	false	true - $\{X_2/S_1\}$	false	true if S_1 and S_2 unify

While this choice clearly confines the impact of labelling, keeping the label computational model well separate from the LP one, it is also too restrictive in practice: in fact, application scenarios often need to express some relevant properties via by suitable terms, so that they can influence the label computation. For this purpose, we introduce the notion of *label-interpreted terms*, as a set of semantically-relevant terms in the domain of labels, along with a *pre-processing phase*, where each label-interpreted term is intercepted and replaced with *an anonymous variable labelled with that term*. In this way, any special term in the domain of label can be treated normally by the f_L combining function, with no need to change the basic model above.

3 Prototype in tuProlog

In this Section we apply our model in the context of the tuProlog system, designing a tuProlog extension that enables users to define their own labelled applications for their specific domains of interest.

3.1 System architecture

Since tuProlog is Java-based, a language extension can be provided both as a Prolog meta-level or library, or via suitable Java methods [3]. We support both ways, defining first a Prolog-language level to denote labelled variables, then two language extensions (Prolog-based and Java-based) to be used in alterative to denote the labelled application (Subsection 3.2).

Moreover, practical reasons suggest to avoid the proliferation of anonymous variables in the pre-processing phase, since they would pollute the name space and make the code less readable: for this reason, the prototype adopts a language shortcut to specify the label of a term *directly*, with no need to actually replace terms with unbound variables explicitly. However, this is just a linguistic extension – not a model extension –, thus leaving the model properties untouched.

3.2 The language extension

In order to enable users to define their labelled application, a suitable set of Prolog predicates / Java methods is introduced to let users define: *(i)* the label/-variable association; *(ii)* the function f_L that specifies under which conditions two labels unify in the selected domain, returning the new label associated with the unified term; *(iii)* a shortcut for the pre-processing transformation, moving *label-interpreted* terms to the label world, making them labels of undefined variables. For the sake of brevity, we show here how to express the three entities just on the Prolog side only—the Java side being conceptually identical.

- the label/variable association is expressed by the `label_associate(+Var, +Term)` predicate, which associates variable *`Var`* to label *`Term`*; as a further convenience, the `°/2` infix operator is also provided for the same purpose.

– the function f_L is expressed by the `label_generate(+Label1, +Label2, -Label3)` predicate, which encodes how to build a new label from two given ones.
– the pre-processing phase is embedded in the `label_interpret(+Label, +Term)` helper predicate, which succeeds if *Term* can potentially be unified with the label *Label* in the label world, or fails otherwise. Only if the check succeeds, the labelled variable is actually unified with the term.

4 Case studies

In the following we present two application examples: Interval CLP [2] and Dress Selection based on colour-match rules. There, figures show both the syntax (left), and the prototype screenshot (right), which sometimes differs due to the current implementation shortcuts, as discussed below.

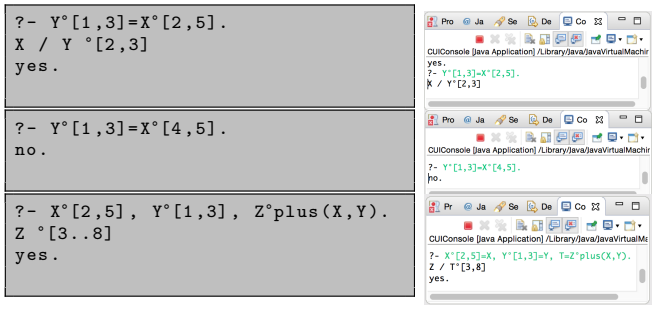
4.1 Interval CLP

In this application scenario, variables are labelled with their admissible numeric interval—that is, `X°[A,B]` means that `X` can span over the range $[A..B]$. Unification succeeds if two numeric intervals overlap, in which case the intersected interval is the newly-computed label. Being specific of the application scenario, the behaviour is defined by the f_L function.

In the code in Figure 2 (top), labelled variable `Y` is unified with labelled variable `X`: the operation succeeds (`X/Y`), with both variables receiving the new label `[2,3]`. Conversely, in the subsequent case the unification fails, because the intersected interval is empty.

In the third case, three aspects are worth highlighting. First of all, an explicit unification is needed to link an unlabelled logic variable (`X`) to a labelled variable (`X°[2,5]`). The second aspect concerns the last term: because the shortcut notation `Z°plus(X,Y)` is not yet supported by the prototype, the unification with a new (possibly anonymous) variable must be made explicit (`Z=_°plus(X,Y)`).

Fig. 2. Interval CLP in Labelled tuProlog



This is why an auxiliary variable (`T`) is introduced—but `Z` could have been reused in alternative, as above. Finally, `plus(X,Y)` is an example of a label-interpreted term (whose meaning is obvious) requiring pre-processing.

4.2 Dress Selection

Here, the goal is to select from the wardrobe all the shirts that respect some given colour constraints: the domain of labels includes then shirts and their colours.

The predicate `shirt(`*`Colour`*`,`*`Description`*`)` represents a shirt with of colour *`Colour`*, expressed in terms of its RGB composition – a triple like `rgb(`*`Red`*`,`*`Green`*`,`*`Blue`*`)` –, synthetically described by *`Description`*. This means that in standard LP the knowledge representation would include facts like `shirt(rgb(255,255,255), amy_white_blouse)`, and alike.

In the labelled context, however, the objective is to move relevant information to the domain of labels. Since labels can only be attached to variables, a different knowledge representation is adopted, where *`Colour`* is seen as a labelled variable, having the `rgb/3` term above as its label. Accordingly, the wardrobe content representation is as follows:

```
shirt(A^rgb(255,240,245), pink_blouse).
shirt(B^rgb(255,222,173), yellow_tshirt).
shirt(C^rgb(119,136,153), army_tshirt).
shirt(D^rgb(188,143,143), periwinkle_blouse).
shirt(E^rgb(255,245,238), cream_blouse).
```

To obtain all the shirts in the wardrobe, a query such as

```
?- shirt(Colour, Description)
```

would obviously generate all the possible solutions in backtracking (Figure 3, top). By suitable exploiting labelled variables, however, the query can be refined by defining a *target colour* in the goal, and exploiting the combining function to get only the dresses whose *dress_colour* is "similar enough" to the target.

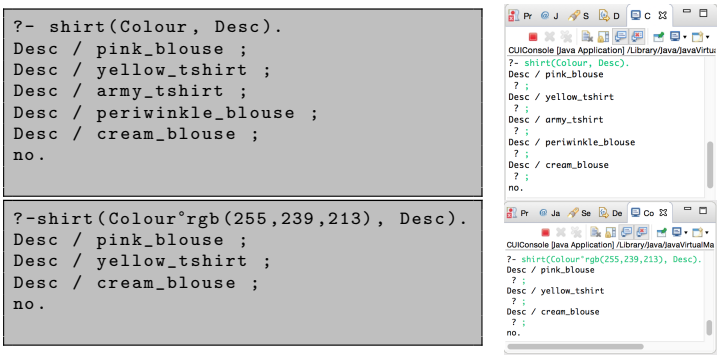
To this end, two RGB colours $c_1 = rgb(r_1, g_1, b_1)$ and $c_2 = rgb(r_2, g_2, b_2)$ are considered similar – so, the corresponding labels unify – if their distance is below a given threshold. For the sake of concreteness, we assume the threshold to be ≤ 100, and the colour distance to be normalised and computed as a distance in a 3D Euclidean space.

The f_L function, checking whether two given labels (c_1=*target colour* and c_2=*dress colour*) are to be considered as similar, can be defined as:

$$f_L(c_1, c_2) = \begin{cases} c_2 & \text{if } \mathit{distance}(c_1, c_2) \leq \mathit{threshold} \in [0..100] \\ \{\} & \text{if } \mathit{distance}(c_1, c_2) > \mathit{threshold} \in [0..100] \end{cases}$$

During the unification of the two labelled variables whose label represent the *target colour* and the *dress colour*, the following steps are performed: if the *dress_colour* is similar to the *target_colour*, the returned label is *dress_colour*—that is, the colour of the selected shirt; if, instead, the two colours are not similar, the empty label is returned, causing the unification process to fail.

Fig. 3. Colour Constraints in Labelled tuProlog



Now let us look for all the shirts similar to the papaya colour ◯: assuming a normalised similarity threshold of 30 (max is 100), the system returns just three solutions (Figure 3, bottom) instead of the five available in un unconstrained world (Figure 3, top), thus actually pruning the solution tree.

References

1. Alsinet, T., Chesñevar, C.I., Godo, L., Simari, G.R.: A logic programming framework for possibilistic argumentation: Formalization and logical properties. Fuzzy Sets and Systems 159(10), 1208–1228 (2008)
2. Benhamou, F.: Interval constraint logic programming. In: Podelski, A. (ed.) Constraint Programming: Basics and Trends, LNCS, vol. 910, pp. 1–21. Springer (1995)
3. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in tuProlog. Science of Computer Programming 57(2), 217–250 (Aug 2005)
4. Gabbay, D.M.: Labelled Deductive Systems, Volume 1. Clarendon Press, Oxford Logic Guides 33 (Sep 1996)
5. Gavanelli, M., Rossi, F.: Constraint logic programming. In: Dovier, A., Pontelli, E. (eds.) A 25-year Perspective on Logic Programming, LNCS, vol. 6125, pp. 64–86. Springer (2010)
6. Holzbaur, C.: Metastructures vs. attributed variables in the context of extensible unification. In: Bruynooghe, M., Wirsing, M. (eds.) Programming Language Implementation and Logic Programming, LNCS, vol. 631, pp. 260–268. Springer (1992)
7. Mariani, S., Omicini, A.: Coordinating activities and change: An event-driven architecture for situated MAS. Engineering Applications of Artificial Intelligence 41, 298–309 (May 2015)
8. Russo, A.: Generalising propositional modal logic using labelled deductive systems. In: Baader, F., Schulz, K.U. (eds.) Frontiers of Combining Systems, Applied Logic Series, vol. 3, pp. 57–73. Springer (1996)
9. tuProlog: Home page. `http://tuprolog.unibo.it/`
10. Van Hentenryck, P.: Constraint logic programming. The Knowledge Engineering Review 6, 151–194 (Sep 1991)