Verso un framework e un linguaggio logico per la programmazione Web
Towards a Logic Language and Framework for Web Programming

Giulio Piancastelli Andrea Omicini Enrico Denti

SOMMARIO/*ABSTRACT*

Nonostante la popolarità del World Wide Web come piattaforma di sviluppo, una adeguata descrizione dei suoi principi architetturali e criteri di progettazione è stata ottenuta solo recentemente, grazie alla introduzione dello stile architetturale REST (Representational State Transfer), che definisce la *risorsa* come la fondamentale astrazione della informazione. Difatti, i linguaggi e gli strumenti correntemente usati per programmare il Web soffrono in genere della mancanza di una corretta comprensione dei suoi vincoli architetturali e progettuali, e di una difformità tra le astrazioni di programmazione che rende difficile sfruttare appieno le potenzialità del Web.
I linguaggi dichiarativi sono particolarmente adatti alla costruzione di sistemi di programmazione rispettosi della architettura e dei principi del Web. Tra le tecnologie di programmazione logica, tuProlog è espressamente progettato per essere uno dei componenti abilitanti di infrastrutture basate su Internet: le sue proprietà ingegneristiche lo rendono peraltro adatto per il Web, dove la programmazione logica permette la modifica del comportamento delle risorse a tempo di esecuzione. Di conseguenza, questo articolo presenta un modello di programmazione logica per risorse Web basato su Prolog e delinea un framework per sviluppare applicazioni Web fondato su quel modello.

Despite the popularity of the World Wide Web as a development platform, a proper description of its architectural principles and design criteria has been achieved only recently, by the introduction of the Representational State Transfer (REST) architectural style which defines the resource *as the key abstraction of information. In fact, languages and tools currently used for Web programming generally suffer from a lack of proper understanding of its architecture and design constraints, and from an abstraction mismatch that makes it hard to exploit the Web potential.*
Declarative languages are well-suited for a programming system aimed at being respectful of the Web architecture and principles. Among logic technologies, tuProlog *has been explicitly designed to be one of the enabling components of Internet-based infrastructures: its engineering properties make it suitable for use on the Web, where logic programming allows modification of resource behaviour at runtime. Accordingly, in this paper we present a Prolog-based logic model for programming Web resources, and outline a framework for developing Web applications grounded on that model.*

Keywords: World Wide Web, REST, Contextual Logic Programming, tuProlog, Prolog.

1 Motivation and Background

Despite the World Wide Web steadily gaining popularity as the platform of choice for the development and fruition of many kinds of Internet-based systems, a proper description of the Web architectural principles and design criteria has been achieved only recently, by the introduction of the Representational State Transfer (REST) architectural style for distributed hypermedia systems [4]. REST defines the *resource* as the key abstraction of information, and prescribes communication and interaction among resources to occur through a *uniform interface* by transferring a *representation* of a resource current state.

Yet, from the early years of procedural CGI scripts to the modern days of object-oriented frameworks, Web application programming has always focussed on different abstractions, such as *page* [5], *controller* [11], and more recently *service* [7], thus suffering from a mismatch that has made it difficult to exploit the full potential of the Web architectural properties. In fact, a page is just the result of a computation involving one or more resources, and deals only with representation issues on the client side. A controller happens to be a programming framework abstraction, sharing almost nothing with the underlying Web platform. Finally, services disregard Web standards such as

URI and HTTP, so they do not get the benefits of the REST architecture in terms of cacheability, connectedness, addressability, uniformity, and interoperability [10].

Declarative programming has never been accepted into the Web mainstream, even though logic languages have shown they could effectively handle both communication and co-ordination in a network-based context [1], and logic technologies have been successfully used to engineer intelligent components at the core of Internet-based infrastructures [2]. However, the REST focus on resource representations as the main driver of interaction, and the corresponding Web computation model, suggest that declarative languages could play a significant role in the construction of resource-oriented applications. The advantage of using elements from logic programming languages such as Prolog lies in the representational foundations of the Web computation model: a declarative representation of resources may be manipulated and, given the procedural interpretation of Prolog clauses, directly executed by an interpreter when a resource is involved in a computation.

Accordingly, we define a resource programming model (called Web Logic Programming [9]), which exploits elements of the logic paradigm and suitable logic technologies (i.e. the tuProlog engine [2]) so as to build a Web application framework aimed at easing rapid prototyping, and allowing the prototype to evolve while supporting Web architectural properties such as scalability or modifiability.

2 Web Logic Programming

Web Logic Programming (WebLP) [9] is a Prolog-based logic model to program resources and their interaction in application systems following the constraints of the World Wide Web architecture. To describe WebLP, we need both to characterise its main data type abstraction and to define its underlying computation model.

2.1 Resources

REST defines a resource as any conceptual target of an hypertext reference. Any information that can be named can be a resource, including virtual (e.g. a document) and non-virtual (e.g. a person) objects. Starting from this abstract definition, the main properties of resources can be easily determined: a name (in the form of an URI); data, representing the resource state; and behaviour, to be used, for instance, to change state or manage the interaction with other resources. The defining elements of resources can be easily mapped onto elements of well-known logic programming languages such as Prolog. For each resource R we specify its name $N(R)$ as the single quoted atom containing the resource URI identifier; data and behaviour can be further recognised as facts and rules, respectively, in a logic theory $T(R)$ containing the knowledge base associated to the resource.

In particular, if adopted resource names are descriptive and have a definite structure varying in predictable ways [10], they feature an interesting property on their own: any path can be interpreted as including a set of resource names. More precisely, we say that resource names such as the following:

```
http://example.com/sales/2004/Q4
```

encompass the names of other resources, and ultimately the name of the resource associated with the domain at the root of the URI:

```
http://example.com/sales/2004
http://example.com/sales
http://example.com
```

This naming structure suggests that each resource does not exist in isolation, but lives in an information *context* composed by the resources associated to the names *encompassed* by the name of that resource.

To account for the possible complexity of Web computations that may involve more information than it is enclosed in a single isolated resource, the context $C(R)$ is introduced as the locus of computation associated with each resource. The context of a resource is defined by the composition of the theories associated with the resources linked to names which are encompassed by that resource name, including the theory associated with the resource itself. Given a resource R with a name $N(R)$ so that:

$$N(R) \subseteq N(R_1) \subseteq \ldots \subseteq N(R_n)$$

the associated context $C(R)$ is generated by composition:

$$C(R) = T(R) \cdot T(R_1) \cdot \ldots \cdot T(R_n)$$

where any theory $T(R_i)$, containing the knowledge base associated to the resource R_i, can be empty – for instance when there is no entity associated to the name $N(R_i)$.

2.2 Computation Model

According to REST, the Web computation model revolves around transactions in the HTTP protocol. Each transaction starts with a *request*, containing the two key elements of Web computations: the *method information*, that indicates how the sender expects the receiver to process the request, and the *scope information*, that indicates on which part of the data set the receiver should apply the method [10]. On the Web, the method information is contained in the HTTP request method (e.g. GET, POST), and the scope information is the URI of the resource to which the request is directed. The result of a Web computation is a *response*, telling whether the request has been successful or not, and optionally containing the representation of the new state of the target resource.

Adopting a logic programming view of the Web computation model, for each HTTP transaction the request can be translated to represent a deduction by retaining the scope information to indicate the target theory, and by mapping the method information onto a proper logic goal. Then, the computation takes place on the server side of the HTTP transaction, in the context associated to the resource target of the request. Finally, the information resulting from goal solution is translated again into a suitable representation and sent back in the HTTP response.

A computation invoked by a goal G on a resource R triggers the deduction of G on the context $C(R)$. The composition of theories forming $C(R)$ is then traversed in a very similar way as units in Contextual Logic Programming (CtxLP) [6]. The goal G is asked in turn to each theory: the goal fails if no solution is found in any theory, or succeeds as soon as it is solved using the knowledge base in a theory $T(R_i)$. Furthermore, when the goal G is substituted by the subgoals of the matching rule in the theory, by default the computation proceeds from $C(R_i)$ rather than being restarted from the original context.

As an example, consider the user `jdoe` in a bookshelf sharing application, where her shelf is represented by the resource S, identified by the URI `http://example.com/jdoe/shelf`. Suppose that, according to a proper naming scheme, the resource B for biology books lives at `/jdoe/shelf/biology`. When a GET request is issued for that resource, a predicate `pick_biology_books/1` is ultimately invoked on B, depending on a `pick_books/3` predicate that is neither defined in B nor in S. The theory chain in $C(B)$ is then traversed backwards up to the `http://example.com` resource, as depicted in Figure 1, where a suitable definition for `pick_books/3` is finally found. Definitions for other predicates invoked by it are then searched starting from the context of the root resource, rather than $C(B)$ where the computation originally started.

The fixed structure of URIs as resource identifiers makes the composition of theories forming a context static, differently from CtxLP, where it was possible to push or pop units from the context stack at runtime. The structure of identifiers and resources in the Web architecture also dictates a unique direction in which the theories associated to resources composing a context can be traversed: from the outermost (associated with the resource on which the computation has been invoked) to the innermost, passing through the theories belonging to each of the composing resources, until the host resource is finally involved.

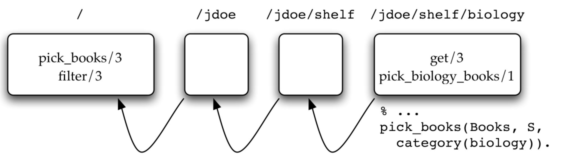

Figure 1: The `/jdoe/shelf/biology` resource responds to a HTTP GET request by eventually invoking the `pick_biology_book/1` predicate, which in turn calls `pick_books/3`. The context is traversed until a proper definition for it is found in the / resource.

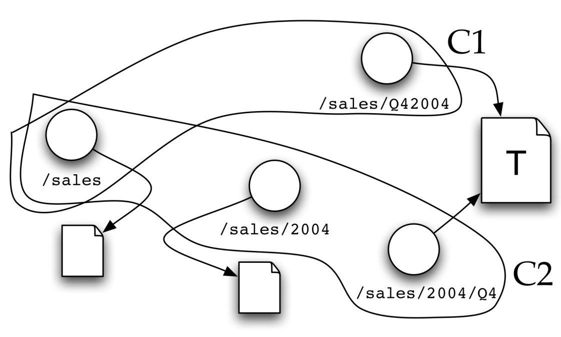

Figure 2: The logic theory of a resource representing sales for the fourth quarter of 2004 can be identified by two different names and therefore live in two different contexts.

2.3 Dynamic Resource Behaviour

The behaviour of a resource can be regarded as dynamic under two independent aspects. First, two or more URIs can be associated to the same resource at any time: that is, the names $N_1(R), \ldots, N_m(R)$ may identify the same resource R, thus the same knowledge base contained in the theory $T(R)$ associated to the resource. Each different name $N_i(R)$ also identifies a different context $C_i(R)$ that the same resource R may live within, (see Figure 2); therefore, predicates that are used in $T(R)$, but are not defined there, may behave in different ways based on the definition given by the context where the resource is called.

The second dynamic aspect of a resource comes from the ability to express behavioural rules as first-class abstractions in a logic programming language: on the one hand, well-known logic mechanisms for state manipulation (the `assertz/1` and `retract/1` predicates) can be exploited to change the knowledge base associated to a resource; on the other, the HTTP protocol itself allows changing a resource by means of a PUT request, whose content should be considered as a modified version of the target resource that has to replace (or be merged with) the original version residing on the server.

As an example, imagine a reading wish list in the previous bookshelf application. Usually, when a book is added, the resource representing the wish list could check local libraries for book availability, and possibly borrow it on user's behalf; if no book can be found, the resource could check its availability in online bookstores, reporting its price to the user for future purchase. During the period of time when an online bookstore offers discounts, the wish list resource should react to the insertion of new books so as to check that store first instead of libraries.

The Web application could then be instructed to change the behaviour of wish list resources by issuing HTTP PUT requests that modify the computational representation of those resources. The PUT requests would carry the new rules in the content, so that wish list resources would accordingly modify their knowledge base. The application could programmatically restore the old behaviour at the end of the discount period, by sending another PUT request containing the previous rule set for each wish list resource.

3 tuProlog: Logic Technology for the Web

tuProlog is a minimal Java-based system explicitly designed to integrate configurable and scalable Prolog components into standard Internet applications, and to be used as the core enabling technology for the provision of basic coordination capabilities into complex Internet-based infrastructures [2]. Alongside configurability and scalability, tuProlog has been designed to offer additional engineering properties suitable for distributed systems and architectures: ease of deployment, lightness, and interoperability in accordance with standard protocols (RMI, CORBA, TCP/IP). Those properties are a good match for the architectural properties described by REST, so that tuProlog can be reasonably employed as the core inference engine taking care of resource computations and interactions.

With the aim of sketching a WebLP framework, a minimal Prolog engine such as tuProlog would need to be augmented with a construct very similar to logic contexts, for which various implementation techniques exist, ranging from the least intrusive meta-interpretation to the most effective virtual machine enhancing. With a similar intent, the architecture of tuProlog has been recently re-engineered to feature the malleability property [8], especially important in allowing a light-weight Prolog technology to be extended with similar ease as the Prolog basic execution model can be extended on the pure linguistic side.

The pervasive integration with Java featured by tuProlog [3] is so much important as we consider how much an established platform for Web development Java has become in the latest years. In order to build a WebLP framework, some Java technology which has proven itself effective for some parts of the Web computation model can be exploited, provided that the abstractions underlying the technology are sound within the Web architectural style. As a first example of such technology, the existing Apache Tomcat web server/container can be considered a multi-threaded efficient environment where tuProlog can be integrated, exploiting component life-cycle management and an HTTP uniform interface implementation. JavaServer Pages are a further example, as an extensible technology to produce resource representations to be consumed on the client side of Web applications. Where instead abstractions suffer from mismatch with respect to the REST architectural style, as the case is for Java servlets used as application controllers, they can be dismissed or re-used with a different purpose, for instance as mere HTTP dispatchers.

4 Future Work

The development of tuProlog as a server-side Web technology and of the WebLP framework will be the main focus of our activity in the near future. Afterwards, we also plan to explore possible extensions of the programming model, mostly based on experience in building a variety of applications on the framework.

REFERENCES

[1] E. Denti and A. Omicini. Engineering multi-agent systems in LuCe. In *International Workshop on Multi-Agent Systems in Logic Programming (MAS '99)*, Las Cruces, NM, USA, 30 November 1999.

[2] E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001.

[3] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 57(2):217–250, August 2005.

[4] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[5] R. Lerdorf and K. Tatroe. *Programming PHP*. O'Reilly, April 2002.

[6] L. Monteiro and A. Porto. A Language for Contextual Logic Programming. In *Logic Programming Languages: Constraints, Functions, and Objects*. The MIT Press, 1993.

[7] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison-Wesley, 2005.

[8] G. Piancastelli, A. Benini, A. Omicini, and A. Ricci. The architecture and design of a malleable object-oriented Prolog engine. In *23th ACM Symposium on Applied Computing (SAC 2008)*, Fortaleza, Ceará, Brazil, 16–20 March 2008.

[9] G. Piancastelli and A. Omicini. A Logic Programming model for Web resources. In *4th International Conference on Web Information Systems and Technologies (WEBIST 2008)*, Funchal, Madeira, Portugal, 4–7 May 2008.

[10] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[11] D. Thomas, D. Heinemeier Hansson, L. Breedt, M. Clark, T. Fuchs, and A. Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2005.