

A Multi-Theory Logic Programming Language for the World Wide Web

Giulio Piancastelli and Andrea Omicini

ALMA MATER STUDIORUM – Università di Bologna
via Venezia 52, 47023 Cesena, FC, Italy

Abstract. Despite the World Wide Web recent architectural formalization in terms of Representational State Transfer (REST) architectural style and Resource-Oriented Architecture (ROA), current languages and tools for Web programming generally suffer from a lack of understanding of its design constraints and from an abstraction mismatch that makes it difficult to fully exploit the Web potential. Based on the insights gained by REST and ROA, we claim that logic languages are well-suited for promoting the Web architecture and principles: in particular, the straightforward mapping of REST and ROA abstractions onto elements of Contextual Logic Programming allows for directly executable logic-based resource representations, as well as dynamic modification of resource behaviour at runtime. Along this line, in this paper we present Web Logic Programming as a Prolog-based language for the World Wide Web embedding the core REST and ROA principles, intended to work as the basis of a framework for the rapid prototyping of Web applications. We define the language operational semantics and discuss some simple but significant programming examples.

Keywords: World Wide Web, Representational State Transfer, Resource-Oriented Architecture, Contextual Logic Programming, Prolog, Web Logic Programming.

1 Introduction

The field of Web application programming has always been dominated by the imperative paradigm, from the early years of procedural CGI scripts, to the modern days of object-oriented frameworks and platforms. In this landscape, the declarative paradigm was represented by few cases: functional and logic languages were never stably accepted into the Web mainstream.

In particular, research on the significance of logic declarative languages in the specific application domain of the World Wide Web focussed on three main themes: *(i)* the provision of libraries, such as PiLLoW [1], to manipulate HTML and XML documents and to operate with the HTTP protocol; *(ii)* the merge between agent- and Web-based technologies resulting in so-called *Internet agents* [2]; and *(iii)* the representation of information in the form of *logic pages*, as promoted, for instance, by the LogicWeb [3] language and system. Unfortunately, the issue of the relationship between logic programming and the Web pretty

much staled across these research lines before entering the new millennium. However, in the latest years, substantial achievements have been obtained in the description and understanding of the architectural principles and design criteria underlying the WWW. The insights gained from those achievements pave a new way toward the exploitation of declarative technologies.

First, Fielding introduced and elaborated the novel Representational State Transfer (REST) architectural style for distributed hypermedia systems [4]. Then, based on the formalization of REST, Richardson and Ruby [5] recently presented the Resource-Oriented Architecture (ROA) as a set of guidelines and best practices for implementing applications that follow the design principles of the Web. REST provides a set of architectural constraints that, when applied as a whole, emphasizes properties such as scalability, uniformity, modifiability, and interoperability. The *resource* is the main REST and ROA data abstraction, defined as any conceptual target of a hypertext reference; communication amongst resources, and between the client-side and the server-side of a Web application, occurs through a *uniform interface* by transferring a *representation* of a resource current state. Yet, languages and tools currently used for Web programming still focus on different abstractions such as *page*, *controller*, and more recently *service*, thus suffering from a mismatch that has made it difficult to fulfill the whole potential of the Web architectural properties.

When confronted with novel insights provided by REST and ROA, also known research on logic programming and the Web shows significant shortcomings. Libraries providing APIs for HTTP and markup languages are limited in scope, just interfacing logic technologies with the Web instead of promoting a wider and deeper integration. Internet agents superimpose the agent-oriented paradigm on the resource-oriented web architecture, without both a clear understanding of the principles underlying the WWW, and a deep treatment of the relationship between the two fundamental abstractions of agent and resource. Despite logic pages [3] being similar to resources, they are conceptually narrower, in the sense that a logic page could be viewed as a resource with one and only one representation. Besides, in spite of dealing with server-side entities and their relationships, logic pages are exclusively carved as a technology to be fully exploited on the client side, thus losing the benefits of a proper use of the Web architectural features. However, the REST focus on resource representations as the main driver of interaction, and the corresponding Web computational model, suggest that declarative languages could play a significant role in the construction of server-side resource-oriented Web applications.

The purpose of our research is to build a logic framework for engineering applications on the World Wide Web, designed so as to promote the architectural principles and constraints described by Fielding, Richardson, and Ruby, and aimed at easing rapid prototyping, also allowing the prototype to evolve while supporting properties such as scalability and modifiability. In this paper, we first show how to map Web concepts onto elements of Contextual Logic Programming [6] according to REST and ROA principles; then, based on that mapping, and on the programming model introduced in [7], we present *Web Logic Programming*

as the fundamental brick of our logic framework in terms of a Prolog-based logic language specific to the domain of Web applications, by defining its operational semantics and discussing some simple but significant programming examples.

2 The Concepts and Properties of REST and ROA

The Representational State Transfer style [4] is an abstraction of the architectural elements within a distributed hypermedia system. The principal data element and key abstraction of information is defined as a *resource*: any conceptual target of a hypertext reference, identified by a unique name. Any information that can be named can be a resource, including a document, an image, a temporal service, a collection of other resources, and a non-virtual object (e.g. a person). When applied to the World Wide Web, the REST style only deals with the abstract definitions of a resource and its external representations, imposing constraints on the uniform interface of resources while leaving the implementation of information sources free for the Web application developer to design.

Building upon REST recommendations, ROA [5] newly proposes that a resource and its Uniform Resource Identifier [8] ought to have an intuitive correspondence; in other words, that URIs should be descriptive. According to ROA best practices, identifiers should also have a definite structure, and that structure should vary in predictable ways. This *addressability* property of Web applications is accompanied by the *connectedness* property, that is the quality of resources to be linked to each other in meaningful ways, so as to follow REST prescription to exploit hypermedia as the engine of the application state [4]. Also in the case of ROA, as it has been noted for REST before, the architectural guidelines do not impose any sort of constraint on the engineering of resource systems.

According to REST and ROA, the World Wide Web computation model revolves around transactions in the HyperText Transfer Protocol (HTTP), a document-oriented protocol aimed at transferring *representations* of a resource current state [9]. Each transaction, such as the one depicted in Fig. 1 (left), 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 operate the method [5]. On systems respectful of REST principles, the method information is contained in the HTTP request method (e.g. GET, POST, PUT, DELETE), and the scope information is the URI of the resource to which the request is directed. Computations, then, occur on the receiving side of the HTTP transaction, where the resource that is the request target needs to perform the operation represented by the method information. The result of a Web computation is a *response*, telling whether the request has been successful or not, and optionally carrying the representation of the new state of the target resource.

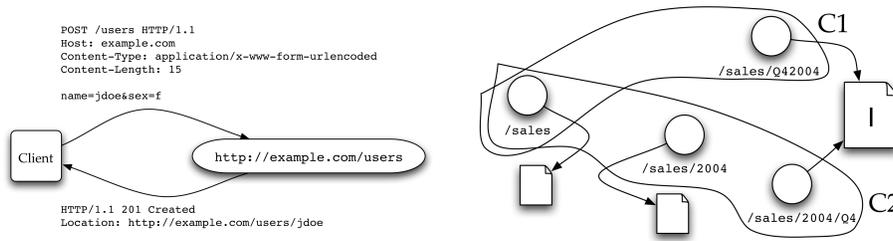


Fig. 1. (Left) A client starts a HTTP transaction asking the `/users` resource to create a new user in the Web application; the resource returns a HTTP response with the identifier of the `jdoe` user just created. (Right) The information *I* (data and behavior) of a resource representing sales for the fourth quarter of 2004 can be identified by two different names and therefore live in two different contexts.

3 Resources and Contexts

Starting from the abstract definitions described in Sect. 2, the main properties of resources can be immediately identified: resources have a name, which in the case of the Web is a unique¹ identifier as defined by the URI standard [8]; resources have data representing their state; and, finally, resources have behavior, to be used, for instance, to change their state, to build up their representations, or to manage interaction with other resources. In particular, when resource names are carefully designed following the ROA best practices about structure and predictability, 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 that resource name, as shown in Fig. 1 (right). Since more than one name can identify the same resource, the context of a resource

¹ The uniqueness is to be intended in the sense that the same identifier cannot be associated to two or more resources at the same time; however, more than one name can identify the same resource at any point in time. For example [5], the sales numbers available at `http://example.com/sales/2004/Q4` might also be available at `http://example.com/sales/Q42004`.

is associated with its name rather than directly with the resource itself. Thus, a resource may live in different contexts at the same time, and feature different behavior according to the context where the interaction with other elements of the system takes place.

From the point of view of logic programming, the properties of Web resources can be easily mapped onto elements of well-known languages such as Prolog [10]. For each resource R we specify its name $N(R)$ as the single quoted atom containing the resource URI identifier; data and behavior can be further recognized as facts and rules, respectively, in a logic theory $T(R)$ containing the knowledge base associated to the resource. The advantage of using logic programming elements lies in the representational foundations of the Web computation model. The declarative representation of resource data and behavior as logic axioms can be directly executed by an inferential interpreter when a resource is involved in a computation, given the procedural interpretation of Prolog clauses.

To account for the possible complexity of Web computations that may involve more information than it is enclosed in a single isolated resource, we introduce the context $C(R)$ as the locus of computation associated with each resource. Following the suggestions given by ROA best practices with respect to resources naming structure, a resource context is defined by the composition of the theories associated with the resources linked to names which are encompassed by the name of that resource, including the theory associated with the resource itself. Given a resource R with a name $N(R)$ for which it holds that:

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

where the inclusion operator follows the encompassment semantics previously defined, then the associated context $C(R)$ is generated by the composition:

$$C(R) = T(R) \cdot T(R_1) \cdot \dots \cdot T(R_n) \tag{1}$$

where the theories $T(R_i)$ could be imagined as occupying the slots of a stack structure, with $T(R)$ at the top and $T(R_n)$ at the bottom.

Alongside resources associated with a URI, we have identified four particular resources corresponding to recurring concepts in the domain of Web applications development. We define these four resources as *implicit* resources, in the sense that they are part of the context of any application resource even if their names are not included in the set of names that are encompassed by the name of that resource. The four implicit resources are: *(i)* the environment resource R_E (identified by the special atom **environment**) representing the environment where the Web application lives, that can entail the operating system and Web server/container where the application has been deployed; *(ii)* the application resource R_A (identified by the special atom **application**) representing the application itself, and containing knowledge that can be applied to every resource belonging to the application; *(iii)* the user resource R_U (identified by the special atom **user**) representing a user of the application; *(iv)* the session resource R_S (identified by the special atom **session**) representing an interaction session of a

user with the Web application. The generation of the context associated to the generic resource R described in (1) has thus to be augmented as:

$$C(R) = T(R) \cdot T(R_1) \cdot \dots \cdot T(R_n) \cdot T(R_S) \cdot T(R_U) \cdot T(R_A) \cdot T(R_E) \quad (2)$$

The context of implicit resources is constructed by composing their theories in the same order as (2); that is, for instance, the context for the user resource is built using the following composition:

$$C(R_U) = T(R_U) \cdot T(R_A) \cdot T(R_E)$$

and the context for the session resource can be constructed by augmenting the user context $C(R_U)$ with the session theory $T(R_S)$ in the following way:

$$C(R_S) = T(R_S) \cdot C(R_U)$$

Note that, as far as the environment resource R_E is concerned, the information contained in its $C(R_E)$ context is the same as the information contained in the corresponding $T(R_E)$ theory.

4 Web Logic Programming

Web Logic Programming (WebLP) is a language to program resources, as the key abstraction of the World Wide Web, and their interaction, in application systems following the Resource-Oriented Architecture. After the characterization of the structure of our main data type offered in Sect. 3, we now need to define the resource computation model underlying the language, while maintaining compatibility with the constraints of the REST architectural style [4]. To this aim, it must be noted that the REST style prescribes every resource to be accessed through a uniform interface, composed by the set of methods defined in the HTTP specification [9]. The advantage of interface uniformity lies in the fact that the semantics of each operation can be defined on a per application basis, instead of being dictated by the architecture once and for all. Besides, REST only deals with resource access from the outside of a Web application, leaving space for languages such as WebLP to define their own computation model within the boundaries of the application.

Adopting a logic programming view of the Web computation model described in Sect. 2, for each HTTP transaction the request gets translated to represent a deduction by retaining the request's scope information to indicate the target set of facts and rules, and by mapping the request's method information onto a logic goal (e.g. `get/3`). Then, the computation takes place on the receiving 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 to a suitable representation, in order to be sent back in the HTTP response. Therefore, to invoke a computation represented by a goal G on a resource R , we adopt the syntax $\mathbb{N}(R) : \mathbb{G}$, which, ultimately, means $C(R) \vdash G$.

Let $C(R)$ be the composition of a number of theories, the query G is asked in turn to each theory. The goal succeeds as soon as it is solved using the knowledge base contained in a theory $T(R_i)$, by exploiting context search to locate the unifying predicate; otherwise, the goal fails if no solution is found in any theory. Furthermore, when the goal G gets substituted by the subgoals $S_j(G)$ of the matching rule in the theory, the computation proceeds from the context of the resource R_i rather than being restarted from the original context. The computation steps can be roughly expressed as follows:

$$T(R_i) \vdash G$$

$$C(R_i) \vdash S_1(G) \wedge \dots \wedge S_n(G)$$

The structure of identifiers and resources in the Web architecture simplifies computations in that no need for a dynamic context augmentation is envisioned. When a resource R_i needs to ask a goal on a resource R_{i-1} on the same path, it has to invoke that computation directly on the R_{i-1} resource. As a consequence, computations are self-contained in the context where they are resolved rather than invoked, making every goal callable from every resource in the application space, without performing artificial inclusion of (or extension to) the knowledge base of other resources. The order in the composition of theories forming a context imposes the direction of computations within the context: from the outermost theory (associated with the resource on which the computation has been invoked) to the innermost, finally involving the theories from the implicit resources, that is session, user, application, and environment, in this order.

However, a computation may need to be performed onto a group of different resources: think, for example, to a comparison of search results for similar queries, or to a filtering from two (or more) distinct sets of photographs. Those are the cases when the context of a computation needs to be composed of more than one resource context; but, since that composed context could play only a part in a broader computation, the requirement to encapsulate the knowledge it contains by keeping it separate from other contexts still retains its validity.

Extending the syntax for invoking a computation, we may express the composition of contexts and the invocation of a computation on it by using the syntax $[N(R_1), \dots, N(R_n)] : G$, which, ultimately, has the following meaning:

$$C_C(R_1, \dots, R_n) = C(R_1) \cup \dots \cup C(R_n)$$

$$C_C(R_1, \dots, R_n) \vdash G$$

The semantics of this computation is given by an aggregated view where the list of contexts can be thought as representing a context composition that behaves as the union of the component contexts. This semantics roughly corresponds to the *union* operator in Modular Logic Programming [11], also included in the LogicWeb system [3]. The goal G succeeds as soon as it is solved on at least a context $C(R_i)$, or it fails when no solution is found in any context. Furthermore, when the goal gets substituted by the S_m subgoals of the matching rule in one of the theories of the context $C(R_i)$, the computation restarts from the original

context union, rather than proceeding from the isolated $C(R_i)$ context. These computation steps can be approximately expressed as follows:

$$C(R_i) \vdash G$$

$$C_C(R_1, \dots, R_n) \vdash S_1(G) \wedge \dots \wedge S_m(G)$$

This sequential approach can be better understood and best illustrated by an example. Suppose to have a context $C(R_1)$ where the rule $p:-q$ is contained, a context $C(R_2)$ containing the q fact, and query $[N(R1), N(R2)]:p$ asked. By the aggregated semantics, the goal is asked in turn to each context corresponding to the resource names involved in the computation. It immediately unifies in $C(R_1)$, leaving the subgoal q to be resolved. Then, q is asked in turn to each context again: it first fails on $C(R_1)$, since this context contains clauses for the predicate p only; but it succeeds in $C(R_2)$, thanks to the presence of the q fact; therefore, the overall query succeeds.

4.1 Operational Semantics

The previous subsections contain a sketch of the computation model of the proposed multi-theory logic language, illustrated by means of short examples and informal explanations. We now specify the operational semantics of Web Logic Programming in the classical form of a set of inference rules, with the aim of further aiding the precise comprehension of the mechanisms and the abstractions already introduced. We start by premising some definitions that will come handy when describing the inference rules.

A logic theory T is defined as a 2-tuple $\langle L_T, K_T \rangle$ containing the identifier L_T for the theory, also called the *label*, and the knowledge base K_T , comprising the logic predicates specified in the theory. Formally, we define L_T as

$$L_T = \{a \mid a \in atom, a \in urispace\}$$

that is, a label is a particular atom belonging to the space of URI identifiers [8]. The knowledge base K_T is defined as the following set of clauses

$$K_T = \{h \leftarrow B \mid h \in atom, B \in goal\}$$

where the clauses have the form $h \leftarrow B$, the clause head h is an atom, and the clause body B is a goal. A goal can be a basic goal (e.g. `goal`) or a *labeled* goal (e.g. `label:goal`, with $label \in L_T$) or a set of basic and labeled goals.

We also explicitly specify the predicates defined in a theory T by the set

$$defined(T) = \{\hat{p} \mid \exists p \leftarrow Q \in K_T\}$$

where the \hat{p} notation is used to represent the principal symbol of the predicate p . For the WebLP language, it also holds that $available(T) \equiv defined(T)$, that is, the set of available predicates in a theory T corresponds to the set of defined predicates in T .

Directly borrowing from Contextual Logic Programming [6], we intend to define derivations in a declarative style, by considering a derivation relation and introducing a set of inference rules for it. We write a tuple in a derivation relation as $C \vdash G[\theta]$, where C is a context, G is a goal, and θ is a set of equalities representing a substitution. We also write an inference rule in the following form

$$\frac{\textit{Antecedents}}{\textit{Consequent}} \{ \textit{Conditions} \}$$

where the *Consequent* is a derivation tuple, the *Antecedents* are zero, one, or two derivation tuples, and the *Conditions* are a (possibly empty) set of arbitrary prepositions. Declaratively, the *Consequent* holds if the *Conditions* are true and the *Antecedents* hold. Procedurally, to establish the *Consequent*, if the *Conditions* are true, the *Antecedents* need to be established.

The notion of derivation is then formalized as a tree such that: (i) any node is labeled with a derivation tuple; (ii) all leaves are labeled by an empty goal; and (iii) the relation between any node and its children is the one between the consequent and antecedents of an instance of an inference rule whose conditions are true. We also characterize the operation of the WebLP system as follows: given a context C and a goal G , the system will try to construct a derivation whose root is labeled by the $C \vdash G[\theta]$ tuple, giving θ as the computed answer substitution result if the derivation succeeds.

The Inference Rules The first three rules are similar to the rules for the Prolog programming language, typically reified in the form of the three clauses in the vanilla meta-interpreter [10]. The first rule is the *Null Rule*, stating that the null goal is derivable in any context, with the empty substitution ϵ .

$$\overline{C \vdash \emptyset[\epsilon]} \quad (N)$$

The second rule deals with goal conjunctions, and it is appropriately called *Conjunction Rule*.

$$\frac{C \vdash G_1[\theta] \wedge C \vdash G_2\theta[\sigma]}{C \vdash G_1, G_2[\theta\sigma[\textit{variables}(G_1, G_2)]]} \quad (C)$$

This rule dictates that, to derive a conjunction of goals² in a context, you need to derive the first conjunct, and then the other conjunct in the very same context (with updated substitutions). So, even if the context may change during the derivation of the first goal, the derivation of the second goal must start from the original context, and not from the context where the derivation of the first goal may have possibly ended.

The third rule is called *Reduction Rule*.

$$\frac{T \cdot C \vdash B\theta[\sigma]}{T \cdot C \vdash g[\theta\sigma[\textit{variables}(g)]]} \left\{ \begin{array}{l} h \leftarrow B \in \textit{variant}(K_T) \\ \theta = \textit{mgu}(g, h) \end{array} \right\} \quad (R)$$

² The notation $\rho[V]$ means the restriction of the substitution ρ to the variables in V .

This rule describes how goals are reduced: if the predicate of an atomic goal is defined in the current theory, that is, a corresponding clause is found, the goal is reduced by using a variant of the clause. As usual in logic programming, in order to reduce goals in derivations we have to use clause variants to respect the local quantification of bound variables.

A second set of three inference rules describes how to navigate contexts during the derivation of a goal, and how it is possible to explicitly switch to a specific context instead of relying on the navigational mechanisms of the language. The first rule of this second set, similar to the context search rule in Contextual Logic Programming, is called *Implicit Up Rule*.

$$\frac{T_b \cdot C \vdash g[\theta]}{T_a \cdot T_b \cdot C \vdash g[\theta]} \{\hat{g} \notin \text{available}(T_a)\} \quad (iU)$$

This is perhaps the most important rule in the second set, since it describes how a predicate not available in a theory can be derived by using the context, and therefore accounts for the dynamic binding of the WebLP language. When a predicate g is not available in the current theory (here represented as T_a) the derivation process moves up to the next available theory in the context. It must be noted that the moving direction strictly follows the path in the URI identifying the resource context where the derivation has started. Given a resource and its URI, the resource ancestors in the URI path are always known, because of an architectural constraint in the naming system of the Web; on the contrary, the resource descendants are unknown, unless it is the resource itself that stores those data, because of a specific requirement of a particular Web application. Therefore, the only moving direction between theories within the same context that makes sense to enforce at the language level is the one coherent with the Web architecture, that is the direction following a resource ancestors up to the root of its path.

The behavior of resources can be regarded as dynamic under two independent aspects. First, two or more URIs can be associated to the same resource at any point in time: the names $N_1(R), \dots, N_i(R)$ may identify the same resource R , thus the same knowledge base contained in the theory $T(R)$ associated to the resource. However, each different name also identifies a different context that the same resource may live within; therefore, predicates that are used in $T(R)$, but are not defined there, may behave in different ways following the definition given by the context where the resource is called to perform a computation. The second dynamic aspect of a resource sprouts from the ability to express behavioral rules as first-class abstractions in a logic programming language: on one hand, it is thus possible to exploit well-known stateful mechanisms to change the knowledge base associated to a resource; on the other hand, the HTTP protocol itself allows changing a resource by means of the PUT method, wherein the 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.

The next inference rule describes an explicit *Context Switch*, where a goal is asked to be derived in a specific context, different from the current one.

$$\frac{C_R \vdash G[\theta]}{C \vdash n_R : G[\theta]} \{n_R \in L_T\} \quad (cS)$$

To derive a goal G labeled with a resource identifier, the system switches from the current context to the context associated with that identifier, then starts the derivation of G in the new context. This is the preferred method to invoke a computation on a resource external to the path associated with the current context. Switching context instead of merging preserves the encapsulation of information that the representation of resources as separated logic theories encourages. The assumption underlying both the Web and the WebLP system is that resources encompassed by a single path form a set of entities so strictly related that they get composed in a new entity called context, where knowledge sharing and behavioral influence are favored.

The last inference rule belonging to the second set is called *Explicit Up*, and describes a convenient shortcut to invoke a derivation of a goal directly on the immediate ancestor of a resource.

$$\frac{T_b \cdot C \vdash g[\theta]}{T_a \cdot T_b \cdot C \vdash \text{parent} : g[\theta]} \quad (eU)$$

The WebLP language offers the special **parent** identifier to let programmers refer to the current theory's parent in the composition representing a context. When compared with the *Implicit Up* inference rule (*iU*), the only difference is that, in the *Explicit Up* case, the check for \hat{g} to belong to the set of available predicates in the current theory (represented as T_a in the rule) is entirely missing. So, even if T_a contains a definition for \hat{g} , the *Explicit Up* rule completely disregards it, and force the system to use whatever definition of \hat{g} is contained in the parent theory T_b to derive the goal.

Another inference rule completes the operational semantics of the WebLP language. The *Context Composition* rule accounts for dynamic context composition by exploiting the *union* operator from Modular Logic Programming [11].

$$\frac{C_1 \cup C_2 \vdash B\theta[\sigma]}{C_1 \cup C_2 \vdash g[\theta\sigma[\text{variables}(g)]]} \left\{ \begin{array}{l} h \leftarrow B \in \text{variant}(K_T) \exists T \in C_1 \\ \theta = mgu(g, h) \end{array} \right\} \quad (cC1)$$

$$\frac{C_1 \cup C_2 \vdash B\theta[\sigma]}{C_1 \cup C_2 \vdash g[\theta\sigma[\text{variables}(g)]]} \left\{ \begin{array}{l} h \leftarrow B \in \text{variant}(K_T) \exists T \in C_2 \\ \theta = mgu(g, h) \end{array} \right\} \quad (cC2)$$

This couple of rules, quite similar to the reduction rule (*R*), describes indeed that a goal to be derived on a composition of context C_1 and context C_2 is reduced by using a clause either from C_1 or from C_2 .

4.2 Programming Examples

We now consider examples making use of some language features, to show how Web systems can be decomposed in resources, how resources can be implemented, and how to perform computations and dynamic behavioral changes.

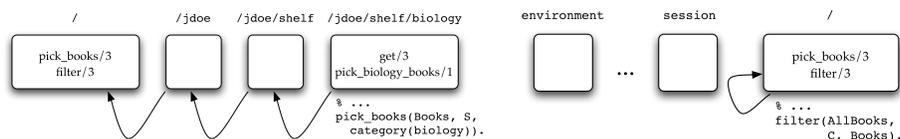


Fig. 2. (Left) The `/jdoe/shelf/biology` resource responds to a HTTP GET request by eventually invoking the `pick_biology_book/1` predicate, which in turn calls `pick_books/3`; the context is traversed until a proper definition for it is found in the `/` resource. (Right) Predicates on which `pick_books/3` depends are further searched starting from the current context rather than where the computation originally started.

Weather Service The first case study is a weather Web application: the main resource R , representing the home page, needs to show: (i) an image of the current weather condition in a specific area (e.g. as a still photograph taken from a webcam), updated on each request; (ii) the time at which the photograph has been shot; and (iii) the current temperature as measured by an external sensor. When a GET request is received, it is translated to a `get/3` goal, to be solved in the context associated with the name $N(R)$ identifying the home page resource. The theory $T(R)$ contains clauses for the `get/3` predicate: to solve that goal, information is dynamically retrieved from the webcam and the sensor, and a logic in-memory representation of those data is built. The time of the shot is taken by invoking a suitable `time/1` predicate, defined in the knowledge base associated to the environment resource R_E . Afterwards, all the data are made available to a template engine that builds a proper representation of the state of R , and finally that representation is sent back in the HTTP response.

Bookshelf Sharing As a second example, imagine a bookshelf sharing Web application. When the user `jdoe` is logged in, her shelf is represented by the S resource, identified by the URI `http://example.com/jdoe/shelf`. Each book is filed under one of more category subjects. The resource B for biology books, for instance, lives at `/jdoe/shelf/biology`. When B receives a GET request, a predicate to pick the list of biology books is ultimately invoked on it:

```
pick_biology_books(Books) :-
    parent_id(Shelf),
    pick_books(Books, category(biology), Shelf).
```

where `parent_id/1` is a predefined predicate returning the identifier of the parent resource. The `pick_books/3` predicate is defined neither in B nor in S , since it has a wider scope. As illustrated in Fig. 2 (left), the theory chain in the context for B is then traversed backwards up to the `/` resource, where a suitable definition for `pick_books/3` is found:

```
pick_books(Books, category(C), Shelf) :-
    findall(B, Shelf : book(B), AllBooks),
    filter(AllBooks, C, Books).
```

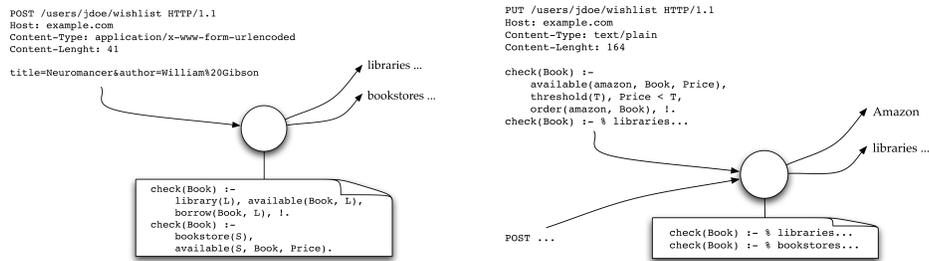


Fig. 3. (Left) POST-ing a new book in the reading wish list triggers an availability check first on local libraries then on online bookstores. (Right) The wish list resource behavior may be dynamically changed by a PUT request carrying new logic rules; afterwards, new books in the list are first checked on Amazon, then only on libraries.

Definitions for other predicates invoked by `pick_books/3` are then searched starting from the current context, rather than $C(B)$ where the computation originally started, as shown in Fig. 2 (right). The final representation of the biology books in the shelf may further depend on some information to be found in the user resource R_U (in this case, mapped on both `user` and the URI `http://example.com/jdoe`): for example, a setting to decide how the book view is organized (e.g. ordered by book insertion time).

Dynamic Reading Wish List As an example of dynamic resource behavior, imagine a reading wish list placed alongside a bookshelf in the previously discussed application. Under usual circumstances, 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. This behavior, depicted in Fig. 3 (left) may be codified by the following rules:

```
check(Book) :- library(L), available(Book, L), borrow(Book, L), !.
check(Book) :- bookstore(S), available(S, Book, Price).
```

Now imagine an online bookstore (e.g. Amazon) offering discounts for a specific period of time. During that period, the wish list resource should react to the insertion of new books so as to check that store first instead of libraries, directly placing an order if the possibly discounted price is inferior to a certain threshold, and to avoid checking other online stores. The new behavior, relative to the store offering discounted prices, is represented by the following rule:

```
check(Book) :- available(amazon, Book, Price),
  threshold(T), Price < T, order(amazon, Book), !.
```

The Web application could then be instructed to change the behavior of wish list resources on a per user basis³ by sending HTTP PUT requests that modify the computational representation of those resources. As shown in Fig. 3 (right), those PUT requests would carry as their content the new rule and the rule dealing with libraries, so that wish list resources would accordingly modify their `check/1` predicate by adopting that new definition. The Web application could then programmatically restore the old behavior at the end of the discount period, by sending another PUT request for each wish list, with a payload adequately set up to the previous `check/1` rule set.

5 Discussion

Starting from the logic programming model for Web resources introduced in [7], we have extended it toward resource contexts composition, and have rigorously defined WebLP as a Prolog-based logic language for Web programming on top of the extended model, by fully describing its operational semantics. Our primary concern has been to follow the principles and capture the key abstractions of the Web as described by REST [4] and ROA [5]. We have mapped the *resource* abstraction to a logic theory, and maintained the *addressability* property by using URIs [8] with the purpose of identifying theories and labeling queries to be asked to specific resources. We have respected the *uniform interface* and let it access logic theories by triggering deductions as a means of exchanging information. Finally, we have embraced the *connectedness* property by tightly binding together, in the notion of *context*, all resources along a single URI path.

By using contexts as its primary computation metaphor, the WebLP language is heavily indebted with previous treatments on the topic, especially Contextual Logic Programming (CtxLP) [6]. Despite being a well-known abstraction, logic programming contexts on the World Wide Web are a complete novelty when built on resources encompassing URIs as in WebLP fashion. Their use in this fashion would have not been possible without the insights and best practices gathered by ROA. The constraints of REST allowed several simplifications of contexts with respect to their original definition. For example, there is no need for dynamic context augmentation, because the structure of resource identifiers is fixed, as per an architectural constraint in the naming system of the Web. Moreover, asking a query by using a relative URI to extend the current context of a resource would have the same computational effect as invoking the goal by directly using the resulting absolute URI. Thus, also for reasons of syntax uniformity, CtxLP context augmentation has been abandoned altogether.

The requirement for context isolation lets WebLP also drop much of the characteristics related to logic modules [12, 11, 13], which were extensively used by CtxLP. The need for a restriction to forbid arbitrary imports from a resource to any other resources (perhaps external to its living context) led us to decide that the subdivision of the application in logic theories corresponding to web

³ By using a proper hierarchy of identifiers and relying on the WebLP computation model, the behavioral changes could also be issued in an application-wide fashion.

resources, and the navigation mechanisms offered by our notion of context, were good enough as modularization features for the WebLP language.

Indeed, resources are an abstraction simple enough to consider WebLP as a radical simplification of CtxLP applied to the domain of the Web rather than an extension, such as languages adding features from concurrency [14] or object-orientation [15]. In particular, resources are not objects in the object-oriented sense, no more than object method calls follow message passing in the distributed systems sense. For instance, the resource abstraction does not carry any notion of inheritance: accounting for polymorphism, or adding explicit lazy and eager binding operators, would have meant to forcibly superimpose other programming metaphors on a Web-oriented language. As an example of such a kind of language, in fact, LogicWeb [3] predates the simpler Modular Logic Programming model [11] without dealing with more complex software engineering paradigms.

LogicWeb is also the Web-oriented logic language most resembling WebLP. However, it is designed to be exploited on the client side: instead of resources, it models HTML documents, which are but just one possible representation of a resource; besides, it lacks the insight on the intrinsic relationship amongst resources encompassed by a single URI that has been only achieved so recently, and that have been nonetheless included in modeling the WebLP language.

6 Conclusions and Future Work

We presented a multi-theory logic programming language called *Web Logic Programming*, designed to model resources, as the key abstraction of the World Wide Web, and their interaction. WebLP is the first Web-oriented logic language to benefit from the architectural specification of hypermedia distributed systems as described by REST [4], and from the insights and guidelines of ROA [5].

The WebLP language is intended to represent the foundation of a logic programming framework for prototyping and engineering applications on the Web so as to follow its architectural principles and design criteria. To fulfill this broader aim, ongoing work is devoted on the one hand to achieve a clear integration between the WebLP language and Prolog, of which WebLP has been designed as an extension, and especially to deal with explicit state issues; on the other hand, to explore possible refinements of the programming model. In the future, more complex Web applications will be constructed, in order to iterate on the framework building process and to showcase its full potential.

References

1. Cabeza, D., Hermenegildo, M.: Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming* **1**(3) (May 2001) 251–282
2. Denti, E., Natali, A., Omicini, A.: Merging Logic Programming into Web-based Technology: A Coordination-based Approach. In De Bosschere, K., Hermenegildo, M., Tarau, P., eds.: *ICLP'97 Post-Conference 2nd International Workshop on Logic Programming Tools for Internet Applications*, Leuven (B) (11 July 1997) 117–128

3. Loke, S.W.: Adding Logic Programming Behaviour to the World Wide Web. PhD thesis, University of Melbourne, Australia (1998)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
5. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (May 2007)
6. Monteiro, L., Porto, A.: A Language for Contextual Logic Programming. In: Logic Programming Languages: Constraints, Functions, and Objects. The MIT Press (1993)
7. Piancastelli, G., Omicini, A.: A Logic Programming model for Web resources. In Cordeiro, J., Filipe, J., Hammoudi, S., eds.: 4th International Conference on Web Information Systems and Technologies (WEBIST 2008), Funchal, Madeira, Portugal, Institute for Systems and Technologies of Information, Control and Communication (INSTICC) (4-7 May 2008) 158-164
8. Berners-Lee, T., Fielding, R.T., Mainster, L.: Uniform Resource Identifiers (URI): Generic Syntax. Internet RFC 2396 (August 1998)
9. Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. Internet RFC 2616 (June 1999)
10. Sterling, L., Shapiro, E.: The Art of Prolog. The MIT Press (1986)
11. Brogi, A., Mancarella, P., Pedreschi, D., Turini, F.: Modular Logic Programming. ACM Transactions on Programming Languages and Systems **16(3)** (1994) 1361-1398
12. Joint Technical Committee ISO/IEC JTC1 SC22: ISO/IEC 13211-2 (June 2000) Information technology — Programming languages — Prolog — Part 2: Modules.
13. Bugliesi, M., Lamma, E., Mello, P.: Modularity in logic programming. Journal of Logic Programming **19-20** (1994) 443-502
14. Mello, P., Natali, A.: Extending Prolog with Modularity, Concurrency and Metarules. New Generation Computing **10(4)** (1992) 335-360
15. Omicini, A., Natali, A.: Object-oriented computations in logic programming. In Tokoro, M., Pareschi, R., eds.: Object-Oriented Programming. Volume 821 of LNCS., Springer-Verlag (1994) 194-212 8th European Conference (ECOOP'94), Bologna, Italy, 4-8 July 1994. Proceedings.