A LOGIC PROGRAMMING MODEL FOR WEB RESOURCES

Giulio Piancastelli and Andrea Omicini
ALMA MATER STUDIORUM – *Università di Bologna, via Venezia 52, 47037 Cesena, FC, Italy*

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

Abstract: In the latest years, substantial achievements have been obtained in the description and formalization of the architectural principles and design criteria underlying the World Wide Web. First, the Representational State Transfer (REST) architectural style for distributed hypermedia systems was introduced, defining the *resource* as the key abstraction of information; then, the Resource-Oriented Architecture (ROA) was presented as a REST-based set of guidelines and best practices for implementing services on Web resources. However, a resource programming model is still missing, since procedural and object-oriented web programming focussed on different abstractions, such as *page*, *controller*, and *service*. Instead, we adopt the logic declarative paradigm to define our model for resource-oriented programming, also showing how its peculiar features lead to novel possibilities for dynamic modification and extension of resource behavior at runtime. In this paper, we first map novel REST and ROA abstractions onto elements of structured logic programming. Then, we present Web Logic Programming as a Prolog-based language for the World Wide Web embedding the core REST and ROA principles, by defining its computation model and discussing a bookshelf sharing web application example.

1 INTRODUCTION

Despite the World Wide Web being increasingly used as a successful platform for open distributed systems of heterogeneous nature, a proper description, understanding, formalization, and divulgement of its architectural principles and design criteria has been achieved only recently. In the latest years, first the Representational State Transfer (REST) architectural style for distributed hypermedia systems was introduced (Fielding, 2000); then, the Resource-Oriented Architecture (ROA) was presented as a REST-based set of guidelines and best practices for implementing services that exploit the full potential of the Web (Richardson and Ruby, 2007). Both REST and ROA focussed on the *resource* as the main data abstraction, defined as any conceptual target of a hypertext reference; they also prescribed communication amongst resources to happen through a *uniform interface* by transferring a *representation* of a resource's current state.

However, a resource programming model is still missing. From the early years of procedural CGI scripts to the modern days of object-oriented frameworks and platforms, web application programming focussed on abstractions such as *page* (Lerdorf et al., 2006), *controller* (Thomas et al., 2006), and more recently *service* (Alonso et al., 2003), which are different in nature from resources, despite seldom sharing some similarities. Instead, the World Wide Web computation model and the REST focus on resource representations as the main driver of interaction suggest that declarative languages could play a significant role in the construction of resource-oriented applications.

We adopt the logic declarative paradigm to define our model for resource-oriented programming for three main reasons. First, the mapping between logic programming elements and the World Wide Web computation model is natural and straightforward. Then, the foundational idea of treating programs as data, leading to resource representations that are directly executable, allows the abstractions to stay simple without reducing their expressiveness and computational power. Finally, this very same idea opens the novel possibility for dynamic modification and extension of resource behavior at runtime.

The purpose of our research is to build a logic framework for engineering web applications, design-

Piancastelli G. and Omicini A. (2008).
A LOGIC PROGRAMMING MODEL FOR WEB RESOURCES.
In *Proceedings of the Fourth International Conference on Web Information Systems and Technologies*, pages 158-164
DOI: 10.5220/0001524501580164

ing it so as to frame it within the principles and constraints of the World Wide Web architectural style. In this paper, we first recall WWW concepts that are relevant to our resource programming model. Then, we show how to map those concepts onto elements of structured logic programming (Monteiro and Porto, 1993; Brogi et al., 1994; Bugliesi et al., 1994) according to REST and ROA principles. Finally, based on this mapping, we present the first fundamental brick of a logic web framework in terms of a logic language extension specific to the domain of web applications, named *Web Logic Programming* (WebLP). Noteworthy programming examples from a bookshelf sharing application are delivered through the whole paper.

2 CONCEPTS AND PROPERTIES OF REST AND ROA

The Representational State Transfer style (Fielding, 2000) is an abstraction of the architectural elements within a distributed hypermedia system. The principal data element and key abstraction of information is characterized as a *resource*: any conceptual target of a hypertext reference. 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). More precisely, a resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time. Each resource is further identified by a unique name, used to reference to the entities involved in an interaction amongst components of a hypermedia distributed system architecture. It should be noted that, even in the context of the 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 whilst leaving the implementation of information sources free for the web application developer to design.

The Resource-Oriented Architecture, *nomen omen*, recognizes resources, identifiers, and representations as key concepts to designing web applications respectful of the REST architectural style constraints (Richardson and Ruby, 2007). The most interesting novel ROA proposal is that a resource and its Uniform Resource Identifier (Berners-Lee et al., 1998) 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's prescription to exploit hypermedia as the engine of the application state (Fielding, 2000). 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 (Fielding et al., 1999). Each transaction starts with a *request*; the request contains 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 (Richardson and Ruby, 2007). 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, happen 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, triggered by the HTTP transaction, is a HTTP *response*, telling whether the request has been successful or not. Typically, the most significant HTTP responses contain the representation of the target resource new state as the result of the computation.

3 FROM RESOURCES TO CONTEXTS

Starting from the abstract definitions described in Section 2, the main properties of resources can be immediately identified: resources have a name, which in the case of the World Wide Web is a unique[1] identifier as defined by the URI standard (Berners-Lee et al., 1998); resources have data representing their state; and resources have behavior, to perform actions such as changing their state and building up their rep-

[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 (Richardson and Ruby, 2007), the sales numbers available at `http://example.com/sales/2004/Q4` might also be available at `http://example.com/sales/Q42004`.

resentations, or to manage interaction with other resources. In particular, when carefully designed following the ROA best practices about structure and predictability, resource names feature an interesting property on their own: each name typically encompasses the names of other resources, and ultimately the name of the resource associated with the domain at the root of the URI. For example, given a bookshelf sharing application, the name of the resource identifying a particular book on the shelf of a specific user:

```
http://example.com/users/jdoe/shelf/5
```

encompasses the following names:

```
http://example.com/users/jdoe/shelf
http://example.com/users/jdoe
http://example.com/users
http://example.com
```

respectively identifying the list of books (that is, the shelf) for the `jdoe` user, the `jdoe` user herself, the list of users in the system, and the web application.

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. Since more than one name can identify the same resource, the context of a resource has to be 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 (Sterling and Shapiro, 1986). For each resource R we specify its name $N(R)$ as the 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. In the bookshelf sharing example, the `http://example.com/users` resource is the target of HTTP POST requests aimed at creating new users in the system; as such, its logic theory contains a `create_user` rule, declaring the actions needed to perform the creation of a new user, and a series of `user` facts, storing data for each user of the application. The advantage of using logic programming elements lies in the representational foundations of the World Wide 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 resource 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:

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

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

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

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

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 (Fielding, 2000).

A computation in logic programming is a deduction of consequences of a set of facts and rules defining relationships between entities (Sterling and Shapiro, 1986). Sets of facts and rules are called *logic theories* (also known as *knowledge bases*), and *logic queries* (also called *goals*) are used to trigger the application of deduction rules on a theory. Answering a query with respect to a theory is determining whether the query is a logical consequence of the theory. Queries are also a means of retrieving information from a logic theory. A query asks whether a certain set of relations holds between the entities described in the theory. When a query contains free variables, the unification algorithm at the core of the logic programming computation model may bound them to entities that augment the boolean result of a deduction telling whether the query has been successful or not.

Adopting a logic programming view of the World Wide Web computation model described in Sect. 2, for each HTTP transaction the request gets translated to represent a deduction by retaining the request scope

information to indicate the target set of facts and rules, and by mapping the request method information onto a logic goal. Then, the computation takes place on the receiving side of the HTTP transaction, in the context associated to the resource target of the request, as already stated in Sect. 3. Finally, the information resulting from goal solution is translated again to a suitable representation, in order to be sent back as the payload of the HTTP response. Therefore, to invoke a computation represented by a goal G on a resource R, we adopt the following syntax:

$$N(R) : G \tag{1}$$

which, using logic symbols, can be expressed as:

$$C(R) \vdash G \tag{2}$$

implying the triggering of a query answering process by means of the application of the deduction rules on the theories composing the context.

In the bookshelf sharing case study, a HTTP POST request directed to the `http://example.com/users` resource represents the invocation of a computation to create a new user in the system. That request is translated to the logic query:

```
'http://example.com/users':post(Request,
                                Response,
                                View)
```

which, in the successful case, is handled by the following rule, contained in the theory associated to the `http://example.com/users` resource:

```
post(Request, Response, _) :-
    create_user(Request),
    param(Request, (user, User)),
    user_url(User, Url),
    header(Response, (location, Url)),
    status(Response, (201, created)).
```

where `user_url/2` is a commodity rule defined in the application resource.

Being the context $C(R)$ the composition of a number of theories, the computation is carried on so that the query G is asked in turn to each theory. The goal fails if no solution is found in any theory; the goal succeeds as soon as it is solved using the knowledge base contained in a theory $T(R_i)$. Furthermore, when the goal G gets substituted by the subgoals $S_i(G)$ 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 expressed as follows:

$$T(R_i) \vdash G \tag{3}$$

$$C(R_i) \vdash S_1(G) \wedge \ldots \wedge S_n(G) \tag{4}$$

As a more elaborated example involving the bookshelf sharing case study, we consider a computation on a typical resource pattern that is one of the bases of modern web applications, involving lists and list items. 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 `http://example.com/jdoe/shelf/biology`. When a GET request is issued for that resource, a predicate to pick the list of biology books is ultimately invoked on B, such as:

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

where `parent_id/1` is a predefined predicate returning the identifier of the parent resource in the path of the current context. The `pick_books/3` predicate is defined neither in B nor in S, since it is an application-wide functionality. The theory chain in the context for B is then traversed backwards up to the `http://example.com` resource, where a suitable definition for `pick_books/3` is found:

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

and works by supposing that the shelf stores books by means of `book/1` facts. According to the computation steps expressed in (3) and (4), a working definition for the `filter/3` predicate is searched starting from the content of the `http://example.com` resource rather than the context of B where the computation originally started. The final representation of the biology books in the shelf further depends on some information stored in the user resource, mapped on the URI `http://example.com/jdoe`: for example, the name of the user, and a setting to decide how the book view is ordered.

Contexts as compositions of theories can be seen as having a layered structure. The definition of a generic computation G as given in (2), (3), and (4) dictates a unique direction in which those layers can be traversed: from the outermost (the theory associated with the resource on which the computation has been invoked) to the innermost, passing through the theory belonging to each of the composing resources, as the social bookmarking computation example shows, passing from the `http://example.com/jdoe/shelf/biology` resource to the `http://example.com/` resource first and to the `http://example.com/jdoe` resource

afterwards. It must be noted that the moving direction strictly follows the path in the URI identifying the resource context where the computation 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 World Wide 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 moving direction between theories within the same context that makes sense to enforce as a default at the language level is the one coherent with the WWW architecture, that is the direction following a resource ancestors up to the root of its path.

The $N(R) : G$ syntax described in (1) is also 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. Resources external to a context do not enjoy the same treatment with respect to that context.

4.1 Dynamic Resource Behavior

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 by using the notation described in (1). 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 outer resources. The order in the composition of theories forming a context imposes the direction of computations within it.

However, the behavior of a resource can be regarded as dynamic under two independent aspects. First and foremost, two or more URIs can be associated to the same resource at any point in time: that is, the names $N_1(R), \ldots, N_i(R)$ may identify the same resource R, thus the same knowledge base contained in the theory $T(R)$ associated to the resource. Each different name 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 (e.g. the `assertz/1` and `retract/1` predicates) 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 entity enclosed as the request payload should be considered as a modified version of the one residing on the origin server (Fielding et al., 1999). Hence, it becomes possible to imagine behavioral changes triggered at runtime not only from peer resources, but also from external actors by using the resource uniform interface, according to the architectural principles of the Web.

As an example of dynamic resource behavior, imagine a bookshelf placed alongside of a reading wish list. Under usual circumstances, when a book is added to the wish list, the resource representing the wish list could check local libraries for book availability, and eventually borrow it on user's behalf; if it is not possible to find the book, the resource could check its availability in online bookstores, reporting its price to the user for future purchase. This behavior is 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. For 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),
  Price < Threshold,
  place_order(amazon, Book), !.
```

The bookshelf sharing web application could then be instructed to change the behavior of wish list resources on a per user basis by issuing HTTP PUT requests that modify the computational representation of those resources. Those PUT requests would carry

the new rule and the rule dealing with libraries in the payload, 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.

Finally, note that, with a proper hierarchy of identifiers, the behavioral changes described in the example could also be carried out in an application-wide fashion, by issuing PUT requests to a root resource common to all wish list contexts, and relying on the WebLP compositional computation model to have wish list resources find the appropriate actions to perform when a new book is added.

5 RELATED WORKS IN LOGIC PROGRAMMING

The primary concern of the Web Logic Programming (WebLP) language was to follow the principles and capture the key abstractions of the World Wide Web as described by the REST architectural style (Fielding, 2000) and the Resource-Oriented Architecture (Richardson and Ruby, 2007). We mapped the *resource* abstraction to a logic theory, and maintained the *addressability* property by using URIs with the purpose of identifying theories and labeling queries to be asked to specific resources. We respected the *uniform interface* and let it access logic theories by triggering deductions as a means of exchanging information. Finally, we embraced the *connectedness* property by tightly binding together, in the notion of *context*, all resources along a single URI path.

The representation of resources as logic theories has been analyzed at the programming model level; yet, in the construction of a WebLP framework, that representation could just play an intermediate role between the resource execution environment and the data persistence system. Resource data could be stored in a variety of different forms; for this purpose, the use of a deductive database system (Ramamohanarao and Harland, 1994) could also be envisioned. However, those systems are almost exclusively based on a restricted logic programming model, which could be suitable for some particular application domains, but neither for the general case of heterogeneous web applications, nor for the WebLP extension of the logic computation model.

Since the resource programming model embodied in WebLP is rooted in logic programming, we consider most relevant for the remain of the present discussion to compare it with computation models that extended the basic logic model described in Sect. 4 with abstractions such as contexts, modules, and objects. 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) (Monteiro and Porto, 1993). 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 introduction would have not been possible without the insights and best practices gathered around the definition of Resource-Oriented Architecture.

The constraints of the REST architectural style allowed several simplifications of contexts with respect to their original definition. For example, there is no need of including logic variables in identifiers. In CtxLP, the parameterization of names influences unit identification and configuration. However, in WebLP any identification problem is already intrinsically resolved, since names are already unique for each resource without the need for parameters; besides, resource configuration mechanisms should be exploited at the moment of constructing new resources, without having any impact on the identification process.

The requirement for context isolation lets WebLP also drop many of the characteristics related to logic modules (SC22, 2000; Brogi et al., 1994; Bugliesi et al., 1994), 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 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, when applied to the domain of the World Wide Web, rather than an extension, such as languages adding features from concurrency (Mello and Natali, 1992) or object-orientation (Omicini and Natali, 1994). 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 bring any notion of inheritance with itself: 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.

LogicWeb (Loke, 1998) is an example of web-oriented logic language that predates the simpler

Modular Logic Programming model (Brogi et al., 1994) without dealing with more complex software engineering paradigms. LogicWeb is also the web-oriented logic language most resembling Web Logic Programming. However, it is designed to be exploited on the client side of the web: instead of *resources*, it models *pages* as HTML documents, which are but just one possible representation of a resource; besides, it unfortunately lacks the insight on the intrinsic relationship amongst resources encompassed by a single URI that have been only achieved so recently, and that have been included in modeling the WebLP language.

6 ONGOING AND FUTURE WORK

We presented a logic programming language extension 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 the Representational State Transfer style (Fielding, 2000), and from the insights and principles of the Resource-Oriented Architecture (Richardson and Ruby, 2007).

The Web Logic Programming language is intended to exploit the full potential of declarative technologies by representing the foundation of a logic programming framework for engineering applications on the World Wide 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; on the other hand, to explore the possible ways of integrating interpreters of logic languages and server-side web technologies, and to study suitable patterns of application architecture to lay on top of the WebLP language.

In the future, more complex web applications will be constructed, in order to iterate on the framework building process and eventually find useful refinements of the WebLP programming model. Applications will also serve the purpose to both perform an expressiveness and usability comparison with other languages (Lerdorf et al., 2006) and frameworks (Thomas et al., 2006) now popular in the mainstream, and showcase the full potential of the novel and peculiar features of Web Logic Programming.

REFERENCES

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2003). *Web Services: Concepts, Architectures and Applications*. Springer-Verlag.

Berners-Lee, T., Fielding, R. T., and Mainster, L. (1998). Uniform Resource Identifiers (URI): Generic Syntax. Internet RFC 2396.

Brogi, A., Mancarella, P., Pedreschi, D., and Turini, F. (1994). Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(3):1361–1398.

Bugliesi, M., Lamma, E., and Mello, P. (1994). Modularity in logic programming. *Journal of Logic Programming*, 19-20:443–502.

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

Fielding, R. T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. Internet RFC 2616.

Lerdorf, R., Tatroe, K., and MacIntyre, P. (2006). *Programming PHP*. O'Reilly.

Loke, S. W. (1998). *Adding Logic Programming Behaviour to the World Wide Web*. PhD thesis, University of Melbourne, Australia.

Mello, P. and Natali, A. (1992). Extending Prolog with Modularity, Concurrency and Metarules. *New Generation Computing*, 10(4):335–360.

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

Omicini, A. and Natali, A. (1994). Object-oriented computations in logic programming. In Tokoro, M. and Pareschi, R., editors, *Object-Oriented Programming*, volume 821 of *LNCS*, pages 194–212. Springer-Verlag. 8th European Conference (ECOOP'94), Bologna, Italy, 4–8 July 1994. Proceedings.

Ramamohanarao, K. and Harland, J. (1994). An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122.

Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. O'Reilly.

SC22, J. T. C. I. J. (2000). Information technology — Programming languages — Prolog — Part 2: Modules. International Standard ISO/IEC 13211-2.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. The MIT Press.

Thomas, D., Heinemeier Hansson, D., Breedt, L., Clark, M., Davidson, J. D., Gehtland, J., and Schwarz, A. (2006). *Agile Web Development with Rails*. Pragmatic Bookshelf.