



UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

[Link to publication](#)

Citation for published version (APA):

Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Chapter 1

Introduction

Traditionally, Logic Programming is used first of all for *problem solving*. Although an important task, implementing the pure problem solving task of a program is typically a minor effort in the overall development of an application. Often, *interaction* is responsible for a much larger part of the code. Where the logic programming community typically stresses the *declarative* aspects of the language, imperative languages provide *control*, an important aspect of interaction. In practice, the distinction is not that rigid. In problem solving we want some control over the algorithms actually used while rules play a role in defining interaction policies.

Mixing languages is a commonly used way to resolve this problem. However, mixing languages harms rapid prototyping because it requires the developers to master multiple languages and to define an interface between the interaction and problem solving components of the application. In this thesis we investigate using the Logic Programming paradigm for both the application logic and the interaction under the assumption that a united environment provides a productive programming environment for knowledge intensive interactive applications.

This introduction starts with a description of the context in which the research was carried out and a characterisation of the software built, followed by a brief overview of and motivation for the use of Logic Programming. After that, we give a historical overview that ends with a timeline of projects described in this thesis, followed by the research questions and an outline of the thesis.

1.1 Application context

This thesis studies the construction of a software infrastructure for building tools that help humans in understanding and modifying knowledge. With software infrastructure, we refer to language extensions and libraries developed to support this class of applications. Our infrastructure is the result of experience accumulated in building tools, a process that is discussed in more detail in section 1.3. Examples of knowledge we studied are conceptual models created by knowledge engineers, ontologies and collections of meta-data that are

based on ontologies. If humans are to create and maintain such knowledge, it is crucial to provide suitable *visualisations* of the knowledge. Often, it is necessary to provide multiple visualisations, each highlighting a specific aspect of the knowledge. For example, if we consider an ontology, we typically want a detailed property sheet for each concept or individual. Although this presentation provides access to all aspects of the underlying data, it is poorly suitable to assess aspects of the overall structure of the ontology such as the concept hierarchy, part-of structures or causal dependencies. These are much better expressed as trees (hierarchies) or graphs that present one aspect of the ontology, spanning a collection of concepts or individuals. The user must be able to interact with each of these representations to *edit* the model. For example, the user can change the location of a concept in the concept hierarchy both by moving the concept in the tree presentation and changing the parent in the property sheet.

Next to visualisations, a core aspect of tools for knowledge management is the actual representation of the knowledge, both internally in the tool and externally for storage and exchange with other tools. In projects that pre-date the material described in this thesis we have used various frame-based representations. Modelling knowledge as entities with properties is intuitive and maps much easier to visualisations than (logic) language based representations (Brachman and Schmolze 1985). In the MIA project (chapter 9, Wielemaker et al. 2003a) we adopted the Semantic Web language RDF (Resource Description Format, Lassila and Swick 1999) as our central knowledge representation format. RDF is backed by a large and active community that facilitates exchange of knowledge with tools such as Protégé (Grosso et al. 1999) and Sesame (Broekstra et al. 2002). RDF provides an extremely simple core data model that exists of triples:

(Subject, Predicate, Object)

Informally, *(Predicate, Object)* tuples provide a set of name-value pairs of a given *Subject*. The power of RDF becomes apparent where it allows layering more powerful languages on top of it. The two commonly used layers are RDFS that provides classes and types (Brickley and Guha 2000) and OWL that provides *Description Logic* (DL, Horrocks et al. 2003; Baader et al. 2003). If the knowledge modelling is done with some care the same model can be interpreted in different languages, providing different levels of semantic commitment. Moreover, we can define our own extensions to overcome current limitations of the framework. RDF with its extensions turned out to be an extremely powerful representation vehicle. In this thesis we present a number of technologies that exploit RDF as an underpinning for our infrastructures and tools.

Most of the work that is presented in this thesis was executed in projects that aim at supporting *meta data* (annotations) for items in collections. Initially we used collections of photos, later we used museum collections of artworks. Knowledge management in this context consists of creating and maintaining the ontologies used to annotate the collection, creating and maintaining the meta data and exploring the knowledge base. Our knowledge bases consist of large RDF graphs in which many edges (relations) and vertices (concepts and instances) are only informally defined and most ‘concepts’ have too few attributes for

computers to grasp their meaning. In other words, the meaning of the graph only becomes clear if the graph is combined with a commonsense interpretation of the *labels*. This implies we have little use for formal techniques such as description logic reasoning (see section 10.5.0.5). Instead, we explore the possibilities to identify relevant sub-graphs based on a mixture of graph properties (link counts) and the semantics of a few well understood relations (e.g., identity, is-a) and present these to the user.

In older projects as well as in the photo annotation project (chapter 9) we aimed at applications with a traditional graphical user interface (GUI). Recently, our attention has switched to web-based applications. At the same time attention shifted from direct manipulation of multiple views on relatively small models to exploring vast knowledge bases with relatively simple editing facilities.

From our experience with projects in this domain we assembled a number of core requirements for the tools we must be able to create with our infrastructure:

1. *Knowledge*

For exchange purposes, we must be able to read and write the standard RDF syntaxes. For editing purposes, we need reliable and efficient storage of (small) changes, together with a history on how the model evolved and support for *undo*. As different applications need different reasoning facilities, we need a flexible framework for experimenting with specialised reasoning facilities. Considering the size of currently available background knowledge bases as well as meta-data collections we estimate that support for about 100 million triples suffices for our experimental needs.

2. *Interactive web applications*

Current emphasis on web applications requires an HTTP server, including concurrency, authorisation and session management. Creating web applications also requires support for the document serialisation formats that are in common use on the web.

3. *Interactive local GUI applications*

Although recent projects needs have caused a shift towards web applications, traditional local GUI applications are still much easier to build, especially when aiming for highly interactive and complex graphical tools.

1.2 Logic Programming

Although selecting programming languages is in our opinion more a matter of ‘faith’¹ than science, we motivate here why we think Logic Programming is especially suited for the development of knowledge-intensive interactive (web) applications.

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be

¹The Oxford Concise English Dictionary: “strong belief [in a religion] based on spiritual conviction rather than proof.”

traced at least as far back as John McCarthy's (McCarthy 1969) advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.

Wikipedia, oct. 2008

Defined as above, logic programming is obviously the perfect paradigm for expressing knowledge. However, when applying this paradigm we are faced with two problems: (1) there is no sufficiently expressive logic language with an accompanying efficient theorem prover and (2) although control may not be of interest for theorem proving, it *is* important when communicating with the outside world where ordering communication actions often matters.

An important break through in the field of logic programming was the invention of the *Prolog* programming language (Colmerauer and Roussel 1996; Deransart et al. 1996). The core of Prolog provides a simple resolution strategy (SLD resolution) for *Horn clauses*. The resulting language has a declarative reading, while its simple resolution strategy provides an imperative reading for a Prolog program at the same time. For example,² the program below can be read as “a(X,Z) is true if b(X,Y) and c(Y,Z) are true”. Particularly if the literals b and c have exactly one solution, the first argument is *input* and the second *output*, it can also be read as “To create Z from X, first call b to create Y from X and then call c to create Z from Y.

```
a (X, Z) :-
    b (X, Y) ,
    c (Y, Z) .
```

In our view, this dual interpretation of the same code greatly contributes to the value of Prolog as a programming language for interactive knowledge-intensive applications. Notably in chapter 4 we see the value of the declarative reading of RDF expressions to achieve optimisation. Declarative reading also helps in many simple rules needed both as glue for the knowledge and as rules in the interactive interface. On the other hand I/O, associated to interactivity, often asks for an imperative reading: we do not want to *prove* `write('hello world')` is true but we want to *execute* this statement.

Since its birth, the Prolog language has evolved in several directions. Notably with the introduction of the WAM (Hassan Aït-Kaci 1991) compiler technology has improved, providing acceptable performance. The language has been extended with extra-logical primitives, declarations, modules and interfaces to other programming languages to satisfy software engineering requirements. At the same time it has been extended to enhance its power as a declarative language by introducing extended unification (Holzbaur 1990) which initiated

²Throughout this thesis we use ISO Prolog syntax for code.

constraint logic programming (CLP) and new resolution techniques such as SLG resolution (Ramakrishnan et al. 1995) which ensures termination of a wider class of Horn clause programs.

Practical considerations In addition to the above motivation for a language that combines declarative and imperative notions there are other aspects of the language that make it suitable for prototyping and research. Many AI languages, both functional- and logic-based, share *reflexiveness*, *incremental compilation* and *safe execution* and lack of *destructive operations*.

Reflexiveness is the ability to process programs and goals as normal data. This feature facilitates program transformation, generation and inspection. Based on this we can easily define more application oriented languages. Examples can be found in section 3.2, defining a language to match XML trees and section 7.2.2.1, defining a language to generate compliant HTML documents and chapter 5 where Prolog syntax is translated partly into code for an external object oriented system. In chapter 4 we exploit the ability to inspect and transform goals for optimising RDF graph matching queries.

Incremental compilation is the ability to compile and (re-)load files into a running application. This is a vital feature for the development of the interactive applications we are interested in because it allows modifying the code while the application is at a certain state that has been reached after a sequence of user interactions. Without incremental compilation one has to restart the application and redo the interaction to reach the critical state again. A similar argument holds for knowledge stored in the system, where reloading large amounts of data may take long. Safe execution prevents the application from crashing in the event of program errors and therefore adds to the ability to modify the program under development while it is running. Prolog comes with an additional advantage that more permanent data is generally stored in clauses or a foreign extension, while volatile data used to pass information between steps in a computation is often allocated on the stacks, which is commonly discarded after incremental computation. This makes the incremental development cycle less vulnerable to changes of data formats that tend to be more frequent in intermediate results than in global data that is shared over a larger part of the application.

The lack of destructive operations on data is shared between functional and logical languages. Modelling a computation as a sequence of independent states instead of a single state that is updated simplifies reasoning about the code, both for programmers and program analysis tools.

Challenges Although the above given properties are useful for the applications we want to build, there are also vital features that are not or poorly supported in many Prolog implementations. Interactive applications need a user interface; dealing with RDF requires a scalable RDF triple store; web applications need support for networking, the HTTP protocol and document formats; internationalisation of applications needs UNICODE; interactivity and scalability require concurrency. Each of these features are provided to some extent in some implementations, but we need integrated support for all these features in one environment.

1.3 Project context

Our first project was Shelley (Anjewierden et al. 1990). This tool provided a comprehensive workbench managing KADS (Wielinga et al. 1992) models. The use of (Quintus-)Prolog was dictated by the project. We had to overcome two major problems.

- Adding graphics to Prolog.
- Establishing a framework for representing the complex KADS models and connecting these representations to the graphics layer.

Anjo Anjewierden developed PCE for Quintus Prolog based on ideas from an older graphics interface for C-Prolog developed by him. PCE is an object oriented foreign (C-)library to access external resources. We developed a simple frame-based representation model in Prolog and used the MVC (Model-View-Controller) architecture to manage models using direct manipulation, simultaneously rendering multiple views of the same data (Wielemaker and Anjewierden 1989).

At the same time we started SWI-Prolog, initially out of curiosity. Quickly, we identified two opportunities provided by—at that time—SWI-Prolog’s unique feature to allow for recursive calls between Prolog and C. We could replace the slow pipe-based interface to PCE with direct calling and we could use Prolog to define new PCE classes and methods as described in chapter 5 (Wielemaker and Anjewierden 2002).

XPCE as it was called after porting the graphics to the X11 windowing system was the basis of the CommonKADS workbench, the followup of Shelley. The CommonKADS workbench used XPCE objects both for modelling the GUI and the knowledge. Extending XPCE core graphical classes in Prolog improved the design significantly. Using objects for the data, replacing the Prolog-based relational model of Shelley, was probably a mistake. XPCE only allows for non-determinism locally inside a method and at that time did not support logical variables. This is not a big problem for GUI programming, but loses too much of the power of Prolog for accessing a knowledge base.

In the MIA project (chapter 9, Wielemaker et al. 2003a; Schreiber et al. 2001) we built an ontology-based annotation tool for photos, concentrating on what is depicted on the photo (the *subject matter*). We decided to commit to the Semantic Web, which then only defined RDF and RDFS. The MIA tool is a stand-alone graphics application built in XPCE. All knowledge was represented using `rdf(Subject, Predicate, Object)`, a natural mapping of the RDF data model. `Rdf/3` is a pure Prolog predicate that was implemented using several dynamic predicates to enhance indexing. With this design we corrected the mistake of the CommonKADS Workbench. In the same project, we started version 2 of the RDF-based annotation infrastructure. The pure semantics of the `rdf/3` predicate was retained, but it was reimplemented in C to enhance performance and scalability as described in chapter 3 (Wielemaker et al. 2003b). We developed a minimal pure Prolog Object layer and associated this with XPCE to arrive at a more declarative approach for the GUI, resulting in the Triple20 ontology tool (chapter 2, Wielemaker et al. 2005).

In the HOPS project³ we investigated the opportunities for RDF query optimisation (chapter 4, Wielemaker 2005) and identified the web, and particularly the Semantic Web, as a new opportunity for the Prolog language. We extended Prolog with concurrency (chapter 6, Wielemaker 2003a) and international character set support based on UNICODE and UTF-8 and added libraries to support this application area better. Motivated in chapter 7 (Wielemaker et al. 2008), we decided to provide native Prolog implementations of the HTTP protocol, both for the server and client. This infrastructure was adopted by and extended in the MultimediaN E-culture project that produced ClioPatria (chapter 10, Wielemaker et al. 2008). Extensions were demand driven and concentrated on scalability and support for a larger and decentralised development team. Scalability issues included concurrency in the RDF store and indexed search for tokens inside RDF literals. Development was supported by a more structured approach to bind HTTP paths to executable code, distributed management of configuration parameters and PIDoc, an integrated literal programming environment (chapter 8, Wielemaker and Anjewierden 2007).

Figure 1.1 places applications and infrastructure on a timeline. The actual development did not follow such a simple waterfall model. Notably ClioPatria was enabled by the previously developed infrastructure, but at the same time initiated extensions and refinements of this infrastructure.

1.4 Research questions

Except for section 1.2 and section 1.3, we consider the choice for Prolog and RDF a given. In this thesis, we want to investigate where the Prolog language needs to be extended and what design patterns must be followed to deploy Prolog in the development of *knowledge-intensive interactive (web) applications*. This question is refined into the three questions below.

1. *How to represent knowledge for interactive applications in Prolog?*

The RDF triple representation fits naturally in the Prolog relational model. The size of RDF graphs and the fact that they are more limited than arbitrary 3-argument relations pose challenges and opportunities.

- (a) *How to store RDF in a scalable way?*

Our RDF store must appear as a pure Prolog predicate to facilitate reasoning and at the same time be scalable, allow for reliable persistent storage and to allow for concurrent access to act as a building block in a web service.

- (b) *How to optimise complex queries on RDF graphs?*

Naively executed, queries to match RDF graph expressions can be extremely slow. How can we optimise them and which knowledge can the low-level RDF store provide to help this process?

³<http://www.bcn.es/hops/>

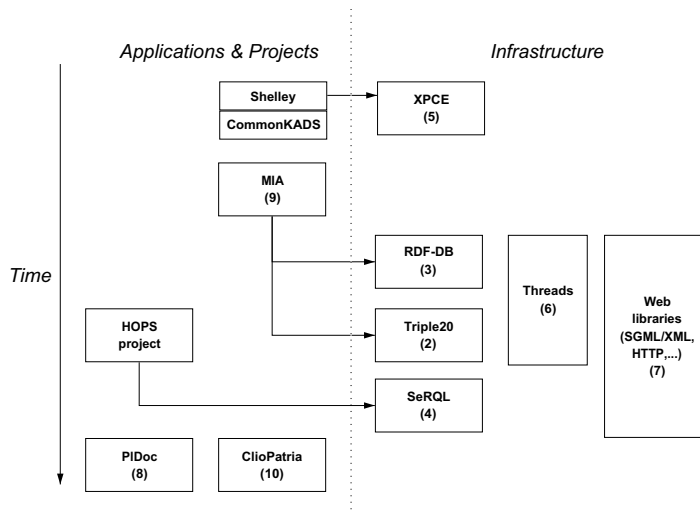


Figure 1.1: This diagram shows all projects and components in chronological order. Numbers between brackets are the chapter numbers describing the component. The arrows indicate the most important influence relations.

2. How to support web applications in Prolog?

Since approximately 2004 our attention in projects shifted from stand-alone GUI applications to web-based applications. As we show in the related work sections of chapter 7, web support is recognised as an issue in the Prolog community, but the solutions are only partial.

(a) How to represent web documents?

The web defines data formats such as HTML, XML and RDF. What is the appropriate way to read, process and write these using Prolog?

(b) How to support web services?

A web service must bind HTTP requests to executable code that formulates a reply represented as a web document. What architecture is needed if this executable code is in Prolog? How do we realise an environment that allows for debugging, is scalable and simple to deploy?

3. How to support graphical applications in Prolog?

As we have seen above, visualisation plays a key role in applications that support the user in managing knowledge. Graphics however does not get much attention in the

Prolog community and intuitively, like I/O, does not fit well with the declarative nature of Prolog.

- (a) *How can we interface Prolog to external object oriented GUI systems?*
Virtually all graphics systems are object oriented and therefore a proper interface to an external object oriented system is a requirement for graphics in Prolog.
- (b) *How to create new graphical primitives?*
Merely encapsulating an object oriented system is step one. New primitives can often only be created by deriving classes from the GUI base classes and thus we need a way to access this functionality transparently from Prolog.
- (c) *How to connect an interactive GUI to the knowledge*
The above two questions are concerned with the low-level connection between Prolog and a GUI toolkit. This question addresses the interaction between knowledge stored in an RDF model and the GUI.

1.5 Approach

Development of software prototypes for research purposes is, in our setting, an endeavour with two goals: (1) satisfy the original research goal, such as evaluating the use of ontologies for annotation and search of multi-media objects and (2) establish a suitable architecture for this type of software and realise a reusable framework to build similar applications. As described in section 1.3, many projects over a long period of time contributed to the current state of the infrastructure. Development of infrastructure and prototypes is a cyclic activity, where refinements of existing infrastructure or the decision to build new infrastructure is based on new requirements imposed by the prototypes, experience in older prototypes and expectations about requirements in the future. Part II of this thesis describes three prototype applications that both provide an evaluation and *lessons learned* that guide improvement of the technical infrastructure described in Part I. In addition to this *formative evaluation*, most of the infrastructure is compared with related work. Where applicable this evaluation is quantitative (time, space). In other cases we compare our design with related designs, motivate our choices and sometimes use case studies to evaluate the applicability of our ideas. Externally contributed case studies are more neutral, but limited due do lack of understanding of the design patterns with which the infrastructure was developed or failure to accomplish a task (efficiently) due to misunderstandings or small omissions rather than fundamental flaws in the design. The choice between self-evaluation and external evaluation depends on the maturity of the software, where mature software with accompanying documentation is best evaluated externally. Chapter 7 and 8 include external experience.

1.6 Outline

Where figure 1.1 shows projects and infrastructure in a timeline to illustrate how projects influenced the design and implementation of the infrastructure, figure 1.2 shows how libraries,

tools and applications described in this thesis depend on each other.

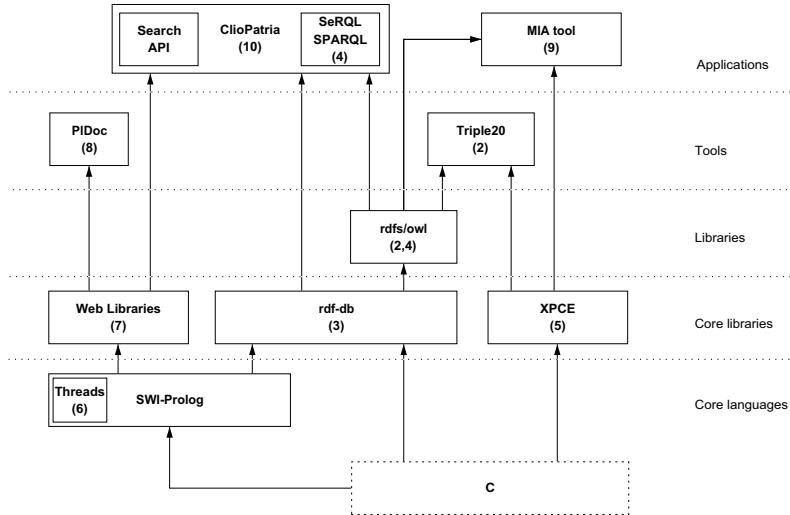


Figure 1.2: Overview of software described in this thesis. The arrows indicate ‘implemented on top of’, where we omitted references from the higher level libraries and applications written in Prolog. Numbers between brackets refer to the chapter that describe the component.

We start our survey near the top with *Triple20* (chapter 2, Wielemaker et al. 2005), a scalable RDF editor and browser. The paper explains how the simple and uniform RDF triple-based data model can be used as an *implementation* vehicle for highly interactive graphical applications. *Triple20* can both be viewed as an application and as a library and is discussed first because it illustrates our approach towards knowledge representation and interactive application development. The paper is followed by four papers about enabling technology: chapter 3 (Wielemaker et al. 2003b) on RDF storage, chapter 4 (Wielemaker 2005) on query optimisation and RDF query language implementation, chapter 5 (Wielemaker and Anjewierden 2002) on handling graphics in Prolog and finally chapter 6 (Wielemaker 2003a) on a pragmatic approach to introduce concurrency into Prolog. The first part is completed with chapter 7 (Wielemaker et al. 2008), providing a comprehensive overview of the (semantic) web support in SWI-Prolog. Being an overview paper it has some overlap with earlier chapters, notably with chapters 3, 4 and 6.

In part II we describe three applications. The first application is *PIDoc* (chapter 8, Wielemaker and Anjewierden 2007), a web-based tool providing literate programming for Prolog. Although *PIDoc* itself is part of the infrastructure it is also an application of our Prolog-based

libraries for building web applications. Chapter 9, (Schreiber et al. 2001; Wielemaker et al. 2003a) describes the role of ontologies in annotation and search, an application that involves knowledge and a traditional GUI. This chapter ends with a lessons learned section that motivates a significant part of the infrastructure described in part I. We conclude with chapter 10 (Wielemaker et al. 2008), which discusses architectural considerations for Semantic Web applications aiming at handling heterogenous knowledge that is only in part covered by formal semantics. The described application (ClioPatria) involves large amounts of knowledge and a rich web-based interactive interface.

