Budapesti Műszaki és Gazdaságtudományi Egyetem
Tudományos Diákköri Konferencia

Prolog Server Pages
Extensible Architecture

Providing Web Interface for Prolog Applications

Prolog alkalmazások illesztése webes környezethez

Hunyadi Levente

Konzulensek:
Dr. Szeredi Péter, Számítástudományi és Információelméleti Tanszék
Szabó Péter, Számítástudományi és Információelméleti Tanszék

2005. október 19.

Kivonat

A Prolog logikai nyelv kifejezőereje lehetővé teszi általános programozási nyelvként történő alkalmazását. Webfelülettel rendelkező alkalmazások fejlesztése esetén azonban jelentős hátrány a nyelv konzol-orientáltsága, kérdés-válasz jellege. Bár a kérdések és válaszok sorozata HTTP kérések és válaszok sorozatába ágyazható, ez a megközelítés, ugyan egyszerű esetekben alkalmazható, ám nem illeszkedik a webes felhasználói felület-modellbe.

A dolgozat előbb áttekinti a webprogramozást elősegítő, rendelkezésre álló Prolog nyelvi kiterjesztéseket, különös tekintettel a PiLLoW könyvtárra, illetve az idegen nyelvekhez kapcsoló illesztési felületekre. Ezek alapján felvázol egy kétszintű rendszerarchitektúrát, amely összetett alkalmazások kialakítását teszi lehetővé. Az architektúra egyrészt támogatást nyújt a HTTP környezet átlátszó, a nyelvbe beépülő elérésére, a kérések közti állapotmegőrzésre, az egyidejű kérések hatékony kiszolgálására, másrészt egy bővíthető sablontechnológiát jelent, amely kifejezésnyelvével és strukturált JSP-szerű szerkezetével webhelyek olyan kialakítása felé vezérel, amelyben a megjelenítés és üzleti logika elválik. Cél, hogy ezáltal lehetővé váljon webfelülettel rendelkező alkalmazás pusztán Prolog nyelven történő megvalósítása, és ne kelljen azt egy webtámogatással bíró rendszerarchitektúrába ágyazni.

A HTTP protokollt és az alacsonyszintű kommunikációt a webkiszolgálóhoz kapcsolódó Prolog Web Container rejti el a programozó elől. Környezetet biztosít felhasználói program futtatására, felel a kérések lebonyolításáért, a kérések közti viszony megőrzéséért és a kérések okozta terhelés elosztásáért, hátteret nyújtva a Prolog Server Pages technológiának. A Prolog Server Pages egyszerű, ám kiterjeszthető struktúrális mechanizmus teszi lehetővé, hogy a megjelenítés és az üzleti logika egyértelműen szétválasztható legyen.

A dolgozat, felhasználva annak többszálú lehetőségeit és beépített alacsonyszintű külső adat-elérési képességeit, egy SWI-Prolog rendszerre írt referenciaimplementációval, illetve ennek segítségével megvalósított példaalkalmazással zárul, amely megvalósítja és bemutatja a felvázolt architektúrát.

Abstract

The expressive power of Prolog enables its use as a general programming language. However, in the case of applications with web interfaces, the console-oriented, question–answer nature of the language is a considerable drawback. Even though the series of questions and answers may be embedded in a series of HTTP requests and responses, this approach, while acceptable for simpler cases, does not integrate into the web user interface model.

First, the paper provides an overview of Prolog language extensions facilitating web programming available at the user's disposal, in particular, the PiLLoW library and foreign language interfaces. Based on these, it sketches a two-layered system architecture that allows constructing complex applications. On one hand, the architecture gives support to access HTTP in a transparent manner integrated into the language, to preserve state between requests, to serve concurrent requests efficiently. On the other hand, the architecture means an extensible template technology with an expression language and structured JSP-like formalism, which encourages the separation of view and business logic. It aims to make it possible to implement applications with web interfaces purely in Prolog, thereby making it unnecessary to embed the application into a system architecture with support for the web.

The HTTP protocol and low-level communication is hidden from the programmer by Prolog Web Container, which connects to the web server. It provides an environment for executing user programs, marshals request handling, preserving state between requests and load balancing, thereby giving background support for the Prolog Server Pages technology. Prolog Server Pages, a simple yet extensible structural mechanism, makes it possible to unambiguously separate view and logic.

The paper, utilizing its multi-threading capabilities and its built-in low-level external data access support, concludes with a reference implementation in SWI-Prolog, which realizes the presented system architecture. A sample application is provided for demonstration.

Contents

1 Introduction **5**

2 Existing libraries for the web environment **7**
2.1 PiLLoW . . . 7
2.1.1 Markup generation with templates . . . 7
2.1.2 User interaction . . . 10
2.1.3 Evaluation . . . 12
2.2 PrologBeans . . . 12
2.2.1 Querying Prolog from Java . . . 13
2.2.2 Receiving Java queries from Prolog . . . 14
2.2.3 Sessions . . . 14
2.2.4 Evaluation . . . 14

3 Prolog Server Pages Architecture **16**
3.1 Own contributions . . . 16

4 Prolog Web Container **18**
4.1 Design considerations . . . 18
4.1.1 Level of integration . . . 18
4.1.2 Process and thread model . . . 21
4.1.3 Synchronization and data sharing . . . 23
4.2 Differences from Java Servlets . . . 24

5 Prolog Server Pages documents **25**
5.1 Representing server page documents . . . 25
5.1.1 Term representation of an XML document . . . 25
5.1.2 The intermediate representation . . . 26
5.1.3 Target-independent representation . . . 27
5.1.4 Relationship of representations . . . 27
5.2 Variables . . . 27
5.3 Expression language (“EL”) . . . 28
5.3.1 Arithmetic expressions . . . 28
5.3.2 String expressions . . . 29
5.3.3 Boolean expressions . . . 29
5.3.4 Invalid expressions . . . 29
5.3.5 Variables in expressions . . . 29
5.3.6 Magic quoting . . . 29

5.3.7 Expression language evaluation . . . 30
5.4 Declarative semantics of functional elements . . . 30
5.4.1 Choice of elements . . . 31
5.4.2 Conditional inclusion . . . 31
5.4.3 Iteration . . . 33
5.4.4 Dynamic construction of elements . . . 34
5.4.5 Variable query and binding . . . 35
5.4.6 Miscellaneous tags . . . 36
5.4.7 Attributes of the root node . . . 37
5.5 Procedural semantics of special elements . . . 37
5.6 Extension mechanism . . . 37
5.7 Prolog Business Logic files . . . 39
5.8 Environment facts . . . 40
5.8.1 Facts for a single request . . . 40
5.8.2 Facts for session management . . . 40
5.8.3 Facts for managing application-level variables . . . 41

6 Reference implementation 42
6.1 Overview . . . 42
6.1.1 Character encoding . . . 43
6.1.2 Global configuration file . . . 44
6.2 Prolog Web Container . . . 44
6.2.1 Server and worker threads . . . 44
6.2.2 Handling requests . . . 45
6.3 Prolog Server Pages . . . 46
6.3.1 Concurrent importation . . . 46
6.3.2 Page preprocessing . . . 47
6.3.3 Page evaluation . . . 47
6.4 Performance evaluation . . . 49
6.5 Sample application . . . 49

7 Evaluation 51
7.1 Future work . . . 52

A Expression language functions 56
A.1 Comparison operators . . . 56
A.2 String comparison operators . . . 56
A.3 Arithmetic operators . . . 57
A.4 String functions . . . 58

B Deployment instructions 59

Chapter 1

Introduction

Prolog has successfully been utilized as a programming language for knowledge bases but its expressive power enables its use for general programming tasks. One serious limitation of the language, however, is its inherent question-answer interface: execution of a sequence of predefined goals is unsuitable in the quickly developing web environment. Attempts have been made to wrap the traditional console behavior into a series of HTTP requests and responses between which state is preserved [19], and even though this approach might be suitable for simple presentation of information, it is by no means a solution for complex web applications that deliver interactive content to a wide range of users. Such require a clear separation of view (user interface) and model (business logic), with adequate support for presenting visual information.

First, existing Prolog libraries to address the task of web interface development will be considered; their advantages and disadvantages will be weighed. It will be shown that while these tools support certain aspects of development for the web, they fail to offer a comprehensive, all-in-one solution. To remedy the situation, a novel system, Prolog Server Pages Extensible Architecture (*Prosper*) will be presented, which facilitates the use of Prolog in the Internet environment. Its aim is to provide a technology that targets common questions of web application development such as request variables, sessions, simultaneous requests, forms or repeatedly used page elements, thereby avoiding the need to wrap Prolog solutions into an existing web technology (such as JSP or ASP .NET) for the sake of web presentation.

Prosper is built up of two major components: Prolog Web Container and Prolog Server Pages. Prolog Web Container provides the background for server pages by automating communication tasks and HTTP-specific functionality with added support for persistence. Essential

In order to demonstrate that the architecture outlined is sound, a reference implementation in SWI-Prolog is given. Utilizing multi-threading, partial evaluation of server page documents, automated synchronized access to shared resources, the implementation is intended to be a suitable tool for middle-size applications with web interfaces. All functionality implemented is then demonstrated by a sample application.

Chapter 2

Existing libraries for the web environment

Prior to presenting Prolog Server Pages Architecture, an overview of existing web libraries available for Prolog is given. Two major representatives, PiLLoW, a web support library, and PrologBeans, a Java interface, are thoroughly analyzed. It will be shown that while these libraries are useful in themselves and back several aspects of developing applications for the web, neither of them can be considered a comprehensive solution. While this analysis of existing libraries does not strive for completeness and not every available library is listed, all key benefits and problems are outlined.

2.1 PiLLoW

Programming in Logic Languages on the Web (PiLLoW) is a public domain web programming library which simplifies the generation of HTML and XML pages by representing them as Herbrand terms, extracting information posted via forms through predefined predicates and access to HTTP header data. It provides a simple template mechanism that is capable of on-the-fly substitution of markup attribute values and of content at predefined places. In addition, it empowers Prolog to act as a Common Gateway Interface (CGI) application by providing simple utility predicates to access execution environment. [10]

2.1.1 Markup generation with templates

The authors of PiLLoW have realized that representing well-formed HTML and XML content as a stream of characters is impractical in the Prolog environment, but the hierarchical tree structure of these markup languages can easily be adopted to suit the needs of a Prolog programmer. An element can be directly mapped into a term comprising of the name

(generic identifier), attributes and contents of the element. Attributes are stored as a dictionary of name-value pairs in the order they occur in the source, while element content is a list of enclosed elements (mapped to terms) and as-is text (atoms or strings) in order of appearance.[1] It can be easily seen that the definition of this mapping is recursive. The example below shows a simple application of the transformation:[2]

```
<html>
  <body lang="hu">
    <h1>title</h1>
    text
    <a ref="http://dp.iit.bme.hu">hyperlink</a>
  </body>
</html>
```

```
env(html, [], [
  env(body, [lang=hu], [
    env(h1, [], [title]),
    text,
    env([ref='http://dp.iit.bme.hu'], [
      hyperlink
    ])
  ])
])
```

It can be realized that the transformation from HTML or XML is reversible, the mapping is a one-to-one correspondence. However, when generating HTML content using Prolog terms, this rigid representation is often inconvenient. PiLLoW extends the term representation to include simpler notation for frequently used HTML elements. Unlike *env*/3, they exploit the meaning of the HTML tags they represent:

- *img*(Source)
  img('http://dp.iit.bme.hu/dp.png')
  `<img src="http://dp.iit.bme.hu/dp.png">`

- *img*(Source, Attributes)
  img('http://dp.iit.bme.hu/dp.png', [alt='dp'])
  `<img src="http://dp.iit.bme.hu/dp.png" alt="dp">`

[1] The ISO standard for Prolog does not include a separate type for an efficient representation of strings. Hereafter, the term *string* refers to a list of character codes. Nevertheless, some Prolog implementations define a separate string type. It is explicitly noted if the term *string* is used in this sense.

[2] For the sake of clarity, whitespace has been removed from the term representation; in actual PiLLoW terms, it is included.

- *ref*(HRef, Text)
  ref('http://dp.iit.bme.hu/ets','ETS')
  `<a href="http://dp.iit.bme.hu/ets">ETS</a>`

- *heading*(Level, Text)
  heading(2, 'Subtitle')
  `<h2>Subtitle</h2>`

- *itemize*(Elements)
  itemize([apple,banana,cherry])
  `<ul><li>apple</li><li>banana</li><li>cherry</li></ul>`

- *enumerate*(Elements)
  itemize([apple,banana,cherry])
  `<ol><li>apple</li><li>banana</li><li>cherry</li></ol>`

- *verbatim*(Text)
  verbatim('`<!-- not to be interpreted as comment -->`')
  `&lt;!-- not to be interpreted as comment --&gt;`

The HTML form of the previous example might, for instance, be generated by the following set of convenience terms:

```
env(html, [], [
  env(body, [lang=hu], [
    heading(1, title),
    text,
    ref(http://dp.iit.bme.hu, hyperlink)
  ])
])
```

The discussed transformation from HTML and XML to Prolog would contribute in itself little to generation of markup content, however, through uninstantiated variables Prolog provides a straightforward way of representing templates. Templates are full-fledged HTML and XML structures except that certain branches of their tree representation are missing and some of their attribute assignments lack the value part. PiLLoW represents templates via special XML constructs: the element `v` replaces missing branches, lacking values in attributes are signaled by names beginning with an underscore (e.g. `_name`). Upon transformation into a term, these constructs in templates are replaced by uninstantiated variables rather than ground terms. A dictionary is built which contains name-variable

pairs in which the name part is derived from the construct itself: either the text content of the node `v` or the trailing characters after the underscore in attribute values. By instantiating variables, a template is transformed into a proper tree. An example is given below:led by names beginning with an underscore (e.g. `_name`). Upon transformation into a term, these constructs in templates are replaced by uninstantiated variables rather than ground terms. A dictionary is built which contains name-variable pairs in which the name part is derived from the construct itself: either the text content of the node `v` or the trailing characters after the underscore in attribute values. By instantiating variables, a template is transformed into a proper tree. An example is given below:

```
<html>
<body lang="_language">
<v>content</v>
</body>
</html>

env(html, [], [
  env(body, [lang=A], [B])
])
```

After the transformation has taken place, the dictionary will be instantiated to the list [language=`A`, content=`B`]. Unifying `A` with `hu` and `B` with *ref*('http://dp.iit.bme.hu', 'hyperlink caption'), a compound term representing a valid HTML page will come to existence:

```
<html>
<body lang="hu">
<a ref="http://dp.iit.bme.hu">hyperlink caption</a>
</body>
</html>
```

2.1.2 User interaction

A substantial amount of the HTML specification deals with forms. An HTML form is a section of a document containing textual content, markup, special elements called controls (checkboxes, radio buttons, inputs, dropdown lists, etc.), and labels on those controls. Users generally "complete" a form by modifying its controls (toggling a checkbox, entering text, selecting an item, etc.) prior to submitting the form to the HTTP server for processing. [21]

Common Gateway Interface

Form handling provided by PiLLoW is based on the assumption that the Prolog interpreter is operated as a CGI application. CGI applications are programs written in any programming language that conform to the interface prescribed by the specification: they are launched by the invoking program, read any HTTP client-related data from their standard input and environment variables, write results on their standard output (and standard error, when appropriate) and terminate. [5]

In a regular CGI scenario, an HTTP request is received by a web server, which makes request-related information available through environment variables, invokes the CGI application and passes the body of the request (if any) to its standard input. Most frequently, data of a request derive from an HTML form. Provided that the submission method for the form is GET, a list of URL encoded name=value pairs, adjoined by ampersands is available in the QUERY_STRING environment variable. For POST requests, this data appears on the standard input stream. Names correspond to HTML form element names supplied with the `name` attribute. Encoding ensures that illegal characters do not appear in the URL by converting them to `%xx` sequences, where `xx` are hexadecimal codes in the range `00-ff`. After having been launched, the application performs a custom operation and prints any results on the standard output, which is directed back by the web server to the client which initiated the request. The application should supply the appropriate HTTP headers, in particular, content type. [16]

Helper predicates

While the CGI protocol is simple in itself, one may easily see that it involves several repetitive tasks that need to be solved regardless of the particular application one is developing. When a Prolog program is launched to handle a request, PiLLoW provides utility predicates that extract data passed via forms for either request type, freeing programmers of the tiresome job of manually processing environment variables and parsing standard input to obtain values tied to specific form controls. Thereby, the particular details of the CGI protocol are hidden from the programmer, valuable development time is saved.

Limitations

Although it parses POST data that derives from forms, XML HTTP requests are not supported by PiLLoW. Used in client-side scripting, this technology allows sending XML embodied in HTTP POST requests. Constructing XML queries and processing replies of the same format, clients can provide a rich user interface without a need to requery the entire page. The technology can save considerable bandwidth and is used in web applications

such as GMail. [17]

2.1.3 Evaluation

Though a suitable solution for many simple applications, PiLLoW in itself fails to provide general support for developing for a web environment. While the template mechanism it defines facilitates the easier generation of HTML and XML content, it is not powerful enough to be employed widely. Conditional inclusion of markup fragments or iteration is not directly supported; Prolog code has to be implemented to instantiate a variable marked in the template beforehand to achieve conditional generation of content, while iteration requires constructing a list of ground terms which will replace a marked template element. While these tasks can be automated, PiLLoW leaves such issues to the library user. In general, PiLLoW provides no clear separation of model and view: the two become inherently intermixed as much effort is needed to provide control flow within the view. In the classical Model 2 architecture, model, which encapsulates business logic, and view, which provides the user interface to this logic are cleanly separated and may be developed independently. PiLLoW, a useful tool in itself, is insufficient to this end.

As a library which hides the CGI interface from the programmer, PiLLoW can be employed to provide simple web services. However, HTTP in itself is stateless and CGI includes no mechanism to provide application state. As CGI applications are initiated and terminated for every single request, the solution is difficult to scale: a serious speed penalty is incurred. PiLLoW, while it provides support for HTTP POST, XML HTTP requests, which are employed in client-side scripting to avoid having to requery the entire page, are not supported. PiLLoW is clearly not an optimum choice for applications that rely on sessions or consume substantial resources such as processing external files or maintaining database connections.

2.2 PrologBeans

PrologBeans is a SICStus extension to provide a Java interface to Prolog processes. The package is designed in a service-oriented view, it is aimed at providing a graphical Java front-end for complex Prolog applications. In a typical scenario, a Java-side client utilizes Prolog functionality by creating a query and binding Java variables to uninstantiated variables of the created goal. The query is then forwarded to a Prolog process for execution, when ready, computed results are returned to Java wrapped into an object. [3]

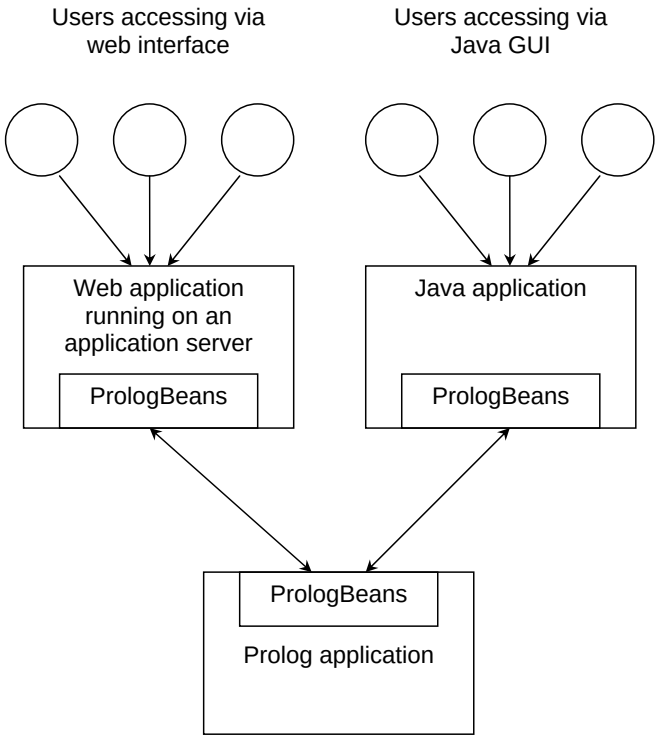

Figure 2.1: Java to Prolog interconnectivity: the PrologBeans architecture (adapted from [4])

2.2.1 Querying Prolog from Java

PrologBeans consists of two stubs: the Java package *se.sics.prologbeans* and the Prolog library *prologbeans*, transparently facilitating the TCP/IP-based communication between the Java Virtual Machine (JVM) and the Prolog engine (Figure 2.1). In order to execute Prolog goals, a Java object must seize a *PrologSession* object. This represents a connection between Java and Prolog: multiple Java threads are allowed to communicate with a single Prolog process and multiple Prolog processes are allowed.[3] Goals are executed by parameterizing the *executeQuery* method of a *PrologSession* object. The class *Bindings* ensures that Java variables are properly delimited when used in Prolog context, for instance, atoms are surrounded by single quotes. The example below shows how the Prolog query *evaluate*('PrologBeans', Out) is run from Java:

```
QueryAnswer answer =
      session.executeQuery("evaluate(In,Out)",
            new Bindings().bind("In","PrologBeans"));
```

[3]In this architecture, multiple Prolog processes cannot share data and can hence be utilized only for speed improvement of concurrent stateless requests.

Execution is synchronous: the Java thread is suspended until query results from Prolog arrive wrapped in a *QueryAnswer* object. This class provides methods to test whether the goal could be proved and indicate if an error occurred. Results are accessed as *Term* objects, which is the Java counterpart of the Prolog term.

2.2.2 Receiving Java queries from Prolog

Queries from Java are handled by hooks preregistered using *register_query*(+Query, :PredicateToCall, +SessionVar).[4] `Query` is a Prolog term that corresponds to that constructed when calling *executeQuery* in Java. `PredicateToCall` is an arbitrary predicate and is used to perform any Prolog-side operations. `SessionVar` signifies the session identifier associated with the request. Prolog, as an interpreted language, is susceptible to insertion and execution of malicious code via *call*/1. Preregistration increases security by allowing only predefined goals to be run from Java.

2.2.3 Sessions

PrologBeans supports sessions implemented through *assert*/1 and *retract*/1 at the Prolog side. When parameterizing a *getPrologSession* call, a session parameter is provided by the Java client implementation, most commonly as an *HTTPSession* object. This transmitted to the Prolog server along with each query: any registered queries receive this as an argument. Session identifiers are used in predicates *session_get*/4 and *session_put*/3. The hook *register_event_listener*/2,3 allows for arbitrary Prolog goals to be executed when a session is started or ends. [4]

2.2.4 Evaluation

PrologBeans, as a Java interface to Prolog, provides an embedding of Prolog technology into a Java environment, thereby exposing Prolog functionality to any Java-related technology. Socket-based communication allows independence from JVM memory management and enables Prolog to be run on a different machine. Multiple Prolog instances – if data does not persist through sessions – allow for decomposition of larger tasks. The platform-independence of Java and the availability of Prolog on numerous systems provides for architectural independence.

Nevertheless, a major drawback of employing PrologBeans in web application development is that knowledge of both Java and Prolog programming languages is assumed. As Java is often required only for the presentation layer and performs no actual processing,

[4]Following standard Prolog convention, + denotes an inbound argument and : a term that may occur as a legal argument to *call*/1.

it seems straightforward to remove it from the service chain. It would be more practical if Prolog itself could provide the view instead of relying on Java to this end. However, applications that heavily make use of Java's GUI capabilities could still benefit from PrologBeans.

In addition, PrologBeans does not automatically leverage multi-threading offered by Java. Multiple simultaneous requests, unless manual balancing is employed, are served by a single Prolog instance. For cases in which data persists on the Prolog side, sessions in particular, any gain exploited through multi-threading is inhibited by serialization of requests as they are served by a single Prolog engine.

Chapter 3

Prolog Server Pages Architecture

The shortcomings of existing web libraries for Prolog having been outlined, one can see that in order to battle the intricacy of developing web applications yet remaining in the Prolog domain, common aspects of generic tasks are to be automated and made transparent. In addition to hiding details of the Hypertext Transfer Protocol, it is practical to provide mechanisms that give access to information received via requests in a manner natural to Prolog. The stateless nature of HTTP is to be wrapped into session management that enables data to persist through requests. It is essential that while being convenient to use, the solution should not hinder scalability. Multi-threading capabilities, while remaining invisible to the Prolog programmer, should be exploited to achieve proper load-balancing of concurrent requests.

For applications that incorporate complex user interaction via web pages, view and logic are to be separated. Designing view should not demand expert knowledge of Prolog, while logic should not contain any visual information intended for presentation. On the other hand, view should offer mechanisms for conditional inclusion of fragments, repetition of fragments with multiple variable substitutions, dynamic creation of elements or content that is dependent upon the context in which pages are displayed.

In order to meet these goals, a multi-layered architecture, similar to that used in the Java 2 Platform, Enterprise Edition (J2EE) is proposed.

3.1 Own contributions

Analyzing the requirements for a suitable architecture, the author sketches a design that meets the conditions imposed. The basics of the HTTP protocol and low-level interaction with the environment are covered by the Prolog Web Container. It addresses questions of communicating with a web server, receiving requests and generating replies, providing persistence of sessions and a view of HTTP environment natural to the Prolog programmer,

thereby providing a background for Prolog Server Pages.

Prolog Server Pages are a simple yet powerful extensible mechanism to declaratively define dynamically created web pages that are separate from business logic. Modeling other similar technologies, they provide a basic set of control flow elements adopted to the Prolog programming environment. In addition, an expression language offers simple computed logic to be incorporated into Prolog Server Pages. A flexible extension mechanism complements the available basic set of elements.

Having outlined a novel approach to provide web interface for Prolog applications, a reference implementation shows that the design put forward is indeed reasonable. Just as Prolog Server Pages Extensible Architecture design, the implementation is entirely own contribution. As no comparable Prolog technology to provide extensive support for web applications exists to-date, the paper includes a sample application that illustrates the entire set of capabilities offered by the reference implementation.

Chapter 4

Prolog Web Container

A web container, in Java terminology, is a part of a web server or application server that provides the network services over which requests and responses are sent. It contains and manages the so-called servlets, components that generate dynamic content, through their lifecycle. [14]

Similarly, a Prolog Web Container serves as an execution environment for Prolog modules that respond to HTTP requests by printing output when invoked. The container hides network-specific tasks from encapsulated code by parsing incoming requests, asserting facts specific to the request environment (URL, form data passed via GET or POST, etc.), managing sessions, calling a well-defined entry goal in the user-defined module and wrapping the response generated by the module into an HTTP response.

4.1 Design considerations

In designing a web container, the questions of transparency, concurrency and persistence are answered with respect to the extent a web server and web container are integrated, how simultaneous requests are serviced and how shared data is stored.

4.1.1 Level of integration

Stand-alone server The strongest level of integration, a stand-alone server unifies web server and container functionality accepting HTTP connections bound to a predefined port. The primary advantage of this architecture is simplicity: no extra communication channels are required between processes while socket interfaces and HTTP processing features available to some Prolog implementations can be harnessed; hence only one system architecture is used in design. However, such implementations often lack advanced features (concurrent request handling, URL rewriting, etc.) and combining the approach

with existing web servers listening on the standard HTTP port may pose problems. Communication through a proprietary port might also be blocked by firewalls. In addition, common HTTP features, which already have optimized solutions for web servers, need to be reproduced. Security and reliability are two requirements that opt for re-using existing web servers. All these question if Prolog is suitable as a general-purpose web server.

Web server APIs Application Programming Interfaces (APIs) allow complementary functionality to be added to standard web server services. Script languages – PHP for instance – provide dynamic linking libraries to be used with web servers, thereby empowering them to interpret designated scripts. Dynamic linking libraries used in this manner represent a strong coupling to servers, they have to be carefully implemented not to provoke unstability for the entire web server. In addition, such libraries are intended for a single server only, servers by different vendors (and very likely, of different versions) may not reuse libraries. The primary advantage of APIs is speed and efficiency, this is clearly a crucial advantage that promotes the wide-spread use of this architecture.

CGI application A Common Gateway Interface (CGI) application is a short-lived process that is initiated by the web server to respond to a single request (see 2.1.2). It requires no direct implementation of HTTP-specific features on the Prolog side and maintains independence from the web server. This independence, however, comes at a cost: startup overhead upon initialization is considerable, especially in the case of Prolog code. Also, sharing data between requests requires additional effort: databases or temporary files need to be utilized to store persisting data such as sessions. On the contrary, CGI is supported by virtually every web server available and little programming is required to transform a solution into a CGI-compliant one. [5]

FastCGI application A FastCGI application is a long-lived process that combines the advantages of a stand-alone server and the CGI approach. Contrary to CGI, FastCGI applications are persistent, that is, they usually remain active after having served a request. They communicate with the web server via the FastCGI protocol, making them independent, hence the solution becomes flexible (Figures 4.2 and 4.3). They make both multi-process and multi-threaded solutions feasible.

In a typical scenario, an HTTP request from a given client is received by the web server. The web server – either by extension or by path – determines that a request is to be forwarded to a FastCGI application. If the application is not yet running, it is started by the server. A communication link is established: unlike CGI, FastCGI is not tied to a specific method, operating system pipes or TCP/IP sockets may both be used; FastCGI applications can thus run on a different machine than the web server.

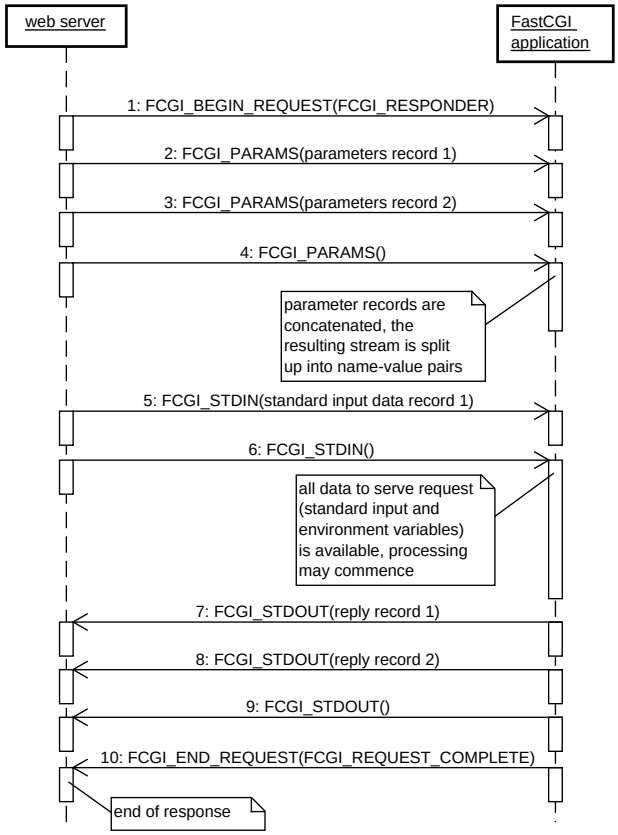

Figure 4.1: Sequence diagram for the FastCGI responder scenario

A notable distinction between the two protocols CGI and FastCGI is the availability of environment variables: in the case of FastCGI these cannot serve as a per-request way of transmitting information and are available for initial configuration of the application only. To overcome this difficulty, the FastCGI protocol defines an interface for sending name–value pairs, which serves the same purpose as environment variables in the case of CGI: submitting data associated with the HTTP request header. The body of the request is tagged accordingly and also transferred to the FastCGI application following the list of name–value pairs. The application responds with data and error streams wrapped into a FastCGI response, which after having been received by the web server, results in an HTTP response. This particular scenario is referred to as the responder role by the specification (Figure 4.1). [9]

Due to the benefits discussed above, the choice for the proposed architecture is FastCGI as it maintains independence from web server solutions yet remaining efficient in terms of overhead. Many popular web servers incorporate extensions for FastCGI support, notably Apache 1.x and 2.x series used in conjunction with mod_fcgi. [2]

browser		**web server**			**FCGI app.**
application	⟵ http ⟶	application	application	⟵ FastCGI ⟶	application
transport	⟵ tcp ⟶	transport	transport	⟵ pipe ⟶	transport
network	⟵ ip ⟶	network			
data link	⟵ ppp ⟶	data link			

Figure 4.2: Integration of a FastCGI application into a web environment in which the web server and the application run on the same machine (based on [23])

browser		**web server**			**FCGI app.**
application	⟵ http ⟶	application	application	⟵ FastCGI ⟶	application
transport	⟵ tcp ⟶	transport	transport	⟵ tcp ⟶	transport
network	⟵ ip ⟶	network	network	⟵ ip ⟶	network
data link	⟵ ppp ⟶	data link			

Figure 4.3: Integration of a FastCGI application into a web environment in which the web server and the FastCGI application run on separate machines

4.1.2 Process and thread model

Single-process, single-threaded model Being the simplest approach, all services provided are integrated into a single-threaded process in the model. While easiest to implement, concurrent requests are queued and a computation- or database-intensive request can block servicing others. In addition, non-responsive tasks can seriously harm performance. Therefore, this choice is inadequate to the needs of an efficient web-related system.

Single-threaded, multi-process model A multi-process approach overcomes the difficulty of concurrent requests by launching multiple single-threaded processes. As most Prolog implementations are single-threaded, this solution can be defended on practical grounds, especially as existing performance management techniques can be re-used to achieve load-balancing. The major difficulty here one has to battle with is data sharing. Unlike threads, processes have distinct data space; inter-process communication needs to be incorporated into the system to implement document caches and sessions. An option is the usage of Linda: a single-server, multiple-client blackboard solution that serializes incoming requests. The server implements a global storage of tuples, which is read and written by connecting clients via primitives $in/1$, $rd/1$ and $out/1$ in a synchronized manner. It is possible to bring a FIFO queue into existence: the primitive $in/1$ not only reads but also removes data from the server blackboard. [12] Such mechanism can be used to dispatch individual tasks to a set of available clients and in this sense it substitutes the commonly used worker-thread model. However, serious speed penalties may be incurred: the efficiency gained by the caching may be lost due to the extra communication overhead.

Managing sessions may be overcome by using external databases, in this case, atomicity, consistency, integrity and durability are provided by a relational database.

Multi-threaded model In the classical multi-threaded model, each request is assigned a separate thread, which eliminates queuing and implicit serialization of requests. The most crucial advantage of this approach to multi-process models is the ability to integrate every operation in one process, allowing common data to be shared. For applications that rely heavily on data exchanged between threads, the difference is immeasurable. On the other hand, care must be taken to synchronize access to shared resources, which imposes additional difficulty on the programmer. Moreover, inadequate precaution in programming (dereferencing uninitialized pointers, etc.) may lead to the instability of the entire process in which threads are running; access violations or segmentation faults, in particular, will cause the process and thereby all threads to be terminated. In addition, few Prolog implementations support multi-threading; this restricts choice of implementation. Nevertheless, while superior to multi-process solutions in terms of performance, it may also respond poorly to a large number of concurrent requests. As each request initiates a new thread, heavy load implies numerous simultaneously existing threads; allocated CPU time slices quickly diminish. As there is associated cost with thread administration upon context switching, measured performance may easily decline.

Worker thread model In this architecture, concurrent requests are handled by parallel so-called worker threads. The major difference between this approach and the classical multi-threaded model is twofold: threads are not associated with a request but persist between them and the number of existing worker threads is limited. Requests are received by a server thread and dispatched to an idle worker thread. The worker thread processes the pending task; when ready, the it returns to available state. If no worker thread is idle when the server receives a request, it is placed in a first-in first-out queue. (Figure 4.4) While performance may be tuned by dynamically creating and destroying threads to reflect server load, extreme conditions do not result in severe decline of performance for tasks whose execution has already begun. The worker thread model approach is used in the case of the Apache 2 series and Microsoft .NET. [11] The proposed architecture takes advantage of the worker-thread model (without dynamic tuning of the number of threads) as it wishes to enjoy the benefits of multi-threading while remaining responsive to considerable load even in spite of increased programming effort required.

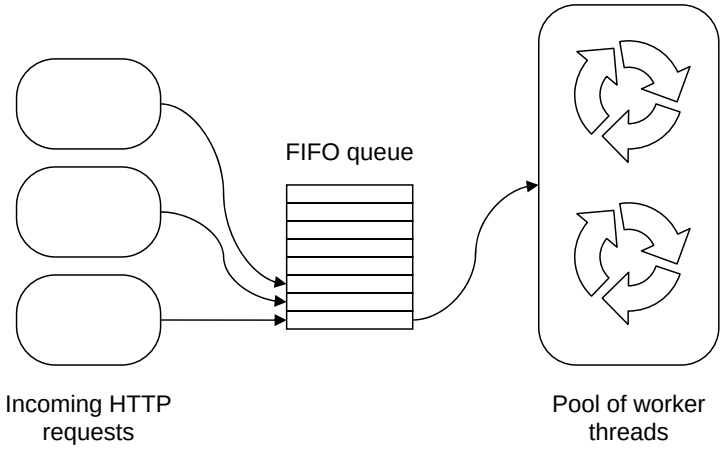

Figure 4.4: The worker-thread model

4.1.3 Synchronization and data sharing

Synchronization in Prolog

In a multi-threaded environment, access to shared resources needs to be synchronized. Two commonly known approaches provided by POSIX are mutual exclusion and condition variables. Mutual exclusion devices work by allowing only a single thread to hold a mutex at a time. Condition variables suspend thread execution until a certain condition becomes true: when other threads modify the variable, the waiting thread is notified so that it may re-examine the condition. Using these in a Prolog program, however, is obtrusive.

A feasible alternative is the application of FIFO queues: threads connect to a queue, and listen for incoming messages. In case data is put into the queue, listening threads awake and one seizes possession of the data, which is then removed from the queue (race condition). A worker-thread model can naturally rely on this technique for dispatching tasks received by the accepting thread.

Nevertheless, in some cases, the traditional approach using mutexes might also be utilized. If data that exists in a global store is to be modified, invariants have to be maintained. While a thread is modifying data, other threads may either see the initial or the final state of the data but not an intermediate (much likely inconsistent) state. This requires an indivisible test-and-set operation that prevents other threads from accessing data while it is being altered.

Data sharing in Prolog

The power of multi-threaded solutions can be harnessed only if an efficient method exists to share common data. Linda-based multi-process models accomplish this by serializing incoming requests at the server.[1] The multi-threaded alternative is critical sections or mutual exclusion. Certain data operations, however, require invariants to be maintained. For instance, while a template file is being read into memory by a thread, another thread that relies on the same file should not begin a distinct import operation. On the contrary, it should be suspended until the operation finished and should resume execution then.

4.2 Differences from Java Servlets

In the J2EE architecture, Java Server Pages heavily build upon the existing Java Servlet technology. Servlets already hide many of the details related to a web environment but provide only a modest, Model 1 architecture: model and view are not at all separated, following the design pattern common to conventional CGI applications while providing improved performance through persistence. Java Server Pages translate into servlets and hence encapsulate all functionality yet satisfying the requirements of Model 2 architecture. The translation is done at compile time. The Prolog solution is designed with requirements of model and view separation in mind and thus does not differentiate as clearly between web container and server pages architecture. As term representations are inherent to Prolog, no compilation of documents are done, rather, though preprocessed, they are interpreted on-the-fly, which allows for dynamic manipulation of content.

[1] SICStus Prolog offers library(linda). A SICStus-based implementation of Prolog Server Pages Architecture should consider using the library.

Chapter 5

Prolog Server Pages documents

A Prolog Server Pages (PSP) document is a text file that conforms to the XML 1.0 standard. [8] Thus, all PSP documents must be well-formed XML documents. On the other hand, elements belonging to a special namespace, some attributes of the root element and attribute values with a distinctive syntax are reserved or treated specially: a simple syntax molds to a powerful template.

First, a discussion of representing arbitrary XML content as Prolog terms is given, followed by the description of how PSP pages are stored in the so-called intermediate representation, which is a modification to the aforementioned XML term representation. Variables, which are named content with visibility constrained to the declaring element's siblings and their descendants, are introduced. Throughout PSP documents, a rich expression language is at the user's disposal, this is then discussed. Control flow in PSP documents is achieved via functional elements; their declarative semantics are given. The section terminates with the extension mechanism that allows incorporating new functional elements with user-defined behavior into the existing set of such elements.

5.1 Representing server page documents

5.1.1 Term representation of an XML document

As previously seen in the case of PiLLoW, it is straightforward to represent XML, a structured markup language as a Prolog term. XML itself corresponds to a tree, which is a severe limitation in some applications, inhibiting the representation of variously interconnected pieces of information, but handy in situations where the easiness of a well-defined structure can be exploited. Here, the Prolog representation of XML is always a ground term and has the following general form: *element*(ElementName, ListOfAttributePairs, ListOfEnclosedContent).

- `ElementName` represents the resolved name of the element, either as an atom or as a compound term `Namespace:Name`, whichever appropriate.
- XML attributes can be seen as additional information attached to an element and as in PiLLoW, they are stored as Name=Value pairs in `ListOfAttributePairs`.
- Element content is composed of a list of strings [1] and compound terms, the latter of which may have any of the following functors:
  - *sdata*/1 for string data,
  - *ndata*/1 for numeric data,
  - *pi*/1 for processing instructions or
  - *element*/3 which allows for nesting of elements.

This representation allows for a lossless mapping: the transformation into a Prolog term can be fully reversed, provided that whitespace collapsing is disabled.[2]

5.1.2 The intermediate representation

Contrary to PiLLoW, it is not uninstantiated variables that distinguish the intermediate term representation of a Prolog Server Page from a regular XML document. Prolog Server Pages use predefined naming conventions and notations to signify special meaning. While not discernible from the initial XML term representation, a second pass on the structure detects and transforms these elements.

The intermediate representation is the simplest possible form that is not context- and target representation-dependent and as such is a sound architectural decision. Namespaces are resolved, built-in operations are identified, simplifications are made, existence checks for extensions are performed and constant expressions are evaluated when transforming an XML Prolog representation into an intermediate one; these are all one-time costs that are incurred only at page initialization, further requests do not pay these costs. On the other hand, the intermediate representation is flexible enough to allow being output in various formats (including HTML, XHTML and XML) with multiple variable substitutions.

Elements whose name is qualified with the namespace *psp* map to built-in behavior including conditional embedding, iteration, variable assignment, value insertion, dynamic element creation or caching; their term representation of *element*/3 is transformed into *builtin*/3, *cached*/2, variants of *capture*/2 or *insert*/1. Elements whose name is qualified

[1] Here, the term *string* refers to a separate type. For implementations that do not define it, atoms can be used. Strings are more efficient in terms of overhead when only single instances exists. Within the Prolog Server Pages chapter, strings are always used in this sense.

[2] Whitespace is always collapsed within tags but not within attribute values.

with a previously registered namespace [3] map to *extension*/3. Attribute values that are enclosed within braces are treated as expression language terms and evaluated or simplified based on whether they consist of purely constants or contain variables as well.

5.1.3 Target-independent representation

Though syntactically identical to XML representation in all aspects, for the sake of clarity, a distinction is made: target-independent representation is the basis for final markup output generation and thus contains no control flow instructions (conditionals, iterations, etc.).

5.1.4 Relationship of representations

The term representation of XML documents is *preprocessed* into intermediate representation. Intermediate representation terms are transformed into target-independent representation through the process page *evaluation*. The latter transformation requires context information, that is, variables are bound and propagated. Target-independent representation, which bears the most resemblance to the final format, is *output* upon request.

5.2 Variables

Throughout PSP documents, variables are utilized to associate a name with specified content. Variables fall into two distinct categories: implicit and explicit. Implicit variables are globally available [4] and used to query context information such as PSP document name or path. Explicit variables are defined by the PSP document author and local to the context in which they are defined. "Local context" refers to those descendants of the parent of the declaring element that follow the declaring element, that is, preceding siblings are excluded. This is identical to how variables are used in XSL transformations. [13] Variables with smaller scope cover the visibility of variables of the same name but broader scope, if such behavior is not intended, renaming should be employed.

Explicit variables are further classified into two categories. Simple variables represent numeric or string content, in other words, satisfy the predicate *atomic*/1. Compound variables come to existence by having assigned a name to an intermediate representation term. Implicit variables are always simple. When required, variables are coerced to match the context in which they are used. Simple variables become element lists of a single

[3] Namespaces that map to user-defined functionality are listed in the configuration file.

[4] Or, in other sense, locally available to the root element.

member, which corresponds to the simple variable, compound variables are "written into" a string as if by *write*/1.

5.3 Expression language ("EL")

Attribute values surrounded by braces ({}) distinguish usage of expression language (EL) syntax. Expression language allows placing simple expressions in attributes, which is evaluated prior to any further processing of the element. This behavior, combined with the power of functional tags, can be utilized to incorporate complex logic into PSP templates. Contrary to JSP, PSP expression language cannot occur outside attributes, notably, such constructs are ignored in element content.[5] [22]

Expression language constructs are typed. Three basic types are distinguished: numbers, strings and Booleans. Numbers include integers and floats and may be used as inputs for arithmetic expressions. Strings can be concatenated, checked for identify, searched and their length calculated. Boolean expressions include the special atoms true and fail, and may be preceded or adjoined by Boolean operators not (\+), and (comma), or (semicolon). Ways exist to convert one type into another: notably Boolean expressions often come into existence as results of comparisons.[6]

5.3.1 Arithmetic expressions

An arithmetic expression is a term that evaluates to a number, either integer or float. This includes numbers (expressions that cannot be simplified), special constants (e, pi), unary and binary operations and arithmetic functions (sqrt, sin, etc.). As a rule of thumb, one may identify an arithmetic expression by recognizing it as a legal second argument of *is*/2. The cast function `number` is provided to convert other types into numbers. Should the cast fail, the expression evaluates to the special constant *null*.

Arithmetic comparison

A comparison expression is of the form `Term Operator Term`, where `Term` is any arithmetic expression, and `Operator` is one of `eq`, `neq`, `lt`, `gt`, `lte` and `gte` corresponding to the Prolog operators `=:=`, `=\=`, `<`, `>`, `=<`, `>=`, respectively.

[5] This is a cleanliness consideration: dynamic content generation is thus constrained to elements. A distinct element `psp:insert` is provided to achieve the same functionality in Prolog Server Pages as in JSP.

[6] The appendix *Expression language* provides a complete list of arithmetic, string and Boolean operators and functions at the user's disposal.

5.3.2 String expressions

Atoms and data derived from globally-scoped variables (e.g. http_get and http_post) classify as strings. String expressions combine strings by concatenation or create substrings by slicing. Numbers are converted into strings by means of a cast, which always succeeds.

String comparison

Strings are compared using the operators `iden` and `niden` (corresponding to Prolog operators `==` and `\=`). Built-in collation is not supported.

5.3.3 Boolean expressions

In addition to the special constants `true` and `fail`, results of comparisons and string searches are Boolean expressions. Boolean expressions are combined using negation (\+), conjunction (comma) and disjunction (semicolon).

5.3.4 Invalid expressions

Expressions that would otherwise cause an exception are evaluated to the reserved constant *null*. This occurs when an expression of the wrong type is supplied (for instance, adding two strings by means of + rather than using concat/n) or an arithmetic expression cannot be computed (division by zero). Any expression containing *null* is itself *null*. If nulls are used in a condition, built-in tags output empty results, even if an else branch is present. This semantics relates closely to the three-state logic commonly used in relational database systems. The `is_null` functions is provided to detect *null*s. *null*s cannot be cast.

5.3.5 Variables in expressions

Previously assigned variables are available for use in EL as strings. Variables are referred to via the function `variable`(Name), Name corresponding to the name of a variable available in the local context. If no variable of the given name exists, the expression evaluates to *null*.

5.3.6 Magic quoting

As single and double quotation marks are disallowed in well-formed XML attributes, when required, atoms in attributes may be quoted by `|` and lists of character codes by `||`. For designers that construct PSP pages, the usage of `&apos;` and `&quot;` is the preferred

way of circumventing this limitation of XML. Magic quoting is globally controlled in the configuration file.

5.3.7 Expression language evaluation

Just as Prolog Server Pages have three representations based on their state: initial XML, intermediate and final (which contains purely output information), expression language terms share similar phases: unevaluated form, simplified form and evaluated form. The unevaluated form corresponds to the structure that results after having read the attribute value into a Prolog term and substituted any occurrences of *variable*/1 with an uninstantiated variable. (In this phase, a dictionary is built of the name-variable pairs.) During the transformation of a page into an intermediate form (preprocessing), a simplification of unevaluated expressions is attempted. In general, unevaluated expressions containing no variables or context references reduce to a constant, though reduction may also be possible in other cases; expressions containing variables remain unevaluated albeit in simpler form. During evaluation, all variable substitutions are made and all expressions must simplify to a constant. If this is not the case, the expression relies on an undeclared variable, which should indicate an error. Erroneous expressions evaluate to the special constant *null*.

5.4 Declarative semantics of functional elements

Elements belonging to the namespace *psp* are processed upon page evaluation, the element itself being replaced by the content it generates. This is done in a recursive manner: encountering a functional tag, based on the attributes it is labeled with, it optionally evaluates its content with a single or multiple variable bindings. Strings and static content evaluate to themselves, therefore the transformation will terminate. This mechanism is similar to but simpler than the one used by Extensible Stylesheet Language Transformations (XSLT). Some attributes have general meaning and are shared by multiple elements:

- The attribute `var` refers to a previously assigned variable. As a value, a variable name is required. The variable must be available in the local context.
- In the attribute `goal`, the user must supply a Prolog predicate or a series of predicates. Predicates should not contain a module specifier and should be available in the context of the attached logic module.
- The attribute `function` serves to refer to predicates that return a value. This attribute is typically used in iterative constructs. As a value of `function`, only a single

predicate can be used, for which a counterpart must be available in the context of the corresponding logic module. This counterpart must have an additional argument than specified in the attribute, which is an uninstantiated variable that will be unified with the return value. This semantics is similar to functions as used in the Mercury language.

- The attribute `expr` is generally used in conjunction with expression language syntax. The value is used as-is, without any further transformation or dereferencing. (Note that expression language is always evaluated prior to any further processing, except for special contexts. An expr="{1+2}" is hence equivalent in this sense to expr="3")
- Source files are referred to by the `src` attribute. Expression language cannot be used to dynamically generate source file names, this is a security consideration, which applies to `psp:import` and `psp:include` elements.

5.4.1 Choice of elements

While the choice of elements may seem arbitrary, the intention was to choose a basic set of elements and allow definition of added functionality through an extension mechanism. The PHP templating engine, Smarty offers as built-in functions `if`, `foreach` and `section` for control flow, `include` for inclusion of external page fragments and `capture` and `insert` to assign data to variables and access variables and context information, respectively. [20] The JSP core tag library contains `set` and `remove` for variable support, `if`, `choose`, `forEach` and `forTokens` (a variant of `forEach`) for flow control, `import` for inclusion of fragments, `redirect` for redirection, `url` for URL-rewriting (to support propagating session identifiers in URLs), `capture` for handling errors and `out` for direct generation of code. [7] Velocity, a Java-based template engine builds on `#if`, `#foreach`, `#set`, `#include` and `#parse` as standard directives. [6] The core functionality provided by the templating engines seems to heavily overlap with slight variations according to the environment in which the different mechanisms are used. Prolog Server Pages functional elements reproduce the majority of the constructs with attention to the Prolog environment in which they are to be used.

5.4.2 Conditional inclusion

psp:if goal="..."

Attempts to prove the specified goal and embeds the content defined by the enclosing tags in the result upon success, otherwise, it is replaced by empty content. The element performs an implicit cut and does not backtrack.

psp:if function="predicate(A, B, C, ...)" [binding="..."]

Appends an uninstantiated argument to the predicate specified and calls the resulting predicate. If this single-predicate goal can be proved, the enclosed content is embedded. The difference between the `if` element as used in conjunction with `goal` and `function` is that in the latter case, the substitution of the uninstantiated argument is available through the local variable `binding`. In order to avoid name clashes for nested constructs, the attribute `binding` can be used to rename this local variable.

psp:if expr="..."

Evaluates the given expression and embeds the content defined by the enclosing tags in the result if the expression is identical to `true`. An error is generated if expression does not evaluate to `true` or `fail`. If braces are omitted, only the two Boolean constants can be used.

psp:if-else

Containing two child elements, `if` and `else`, this element is functionally identical to `if` except that upon failure, the `else` part is included in the result. It is compulsory that the node have exactly two child nodes, one of which is `psp:if`, the other `psp:else`, other child nodes (including character data not enclosed by any of the aforementioned two) are ignored.

psp:choose

Encloses `psp:when` elements and the `otherwise` element.

psp:when

Used in conjunction with the `psp:choose` element, it is provided as an alternative to nested `psp:if` constructs to improve readability. Upon evaluation, the condition of the first `psp:when` child node of a `psp:choose` element is tested, if it fails, evaluation proceeds with the next condition. A `psp:otherwise` node allows an alternative in case all others fail. Syntactically equivalent to the Prolog construct:

```
( predicate_1(A, B, C) -> action_1
; predicate_2(D, E) -> action_2
; action_otherwise
).
```

`psp:when` accepts exactly the same attributes as `psp:if` and results are identical in all aspects. `when` constructs can always be – and internally are – rewritten as nested `if-else` constructs:

```
<psp:choose>
    <psp:when goal="goal1">
        appears if goal1 can be proved
    </psp:when>
    <psp:when expr="expr2">
        appears if expr2 evaluates to true
    </psp:when>
    <psp:otherwise>
        appears otherwise
    <psp:otherwise>
</psp:choose>

<psp:if-else>
    <psp:if goal="goal1">
        appears if goal1 can be proved
    </psp:if>
    <psp:else>
        <psp:if-else>
            <psp:if expr="expr2">
                appears if expr2 evaluates to true
            </psp:if>
            <psp:else>
                appears otherwise
            </psp:else>
        </psp:if-else>
    </psp:else>
</psp:if-else>
```

5.4.3 Iteration

psp:for-each function=”predicate(A, B, C, . . .)” [iterator=”. . . ”]

Appends an uninstantiated variable to the n-ary predicate specified and calls the resulting $(n+1)$-ary predicate. The appended uninstantiated variable is to return a proper list upon success. The content of the element is evaluated for each member of the list, indi-

vidual members are available in the local context of the element by accessing the variable `iterator`. The iterator variable is renamed by specifying an attribute of the same name. Nothing is included in the result if either the predicate fails or the returned list is empty.

psp:for-each-else

In addition to `psp:for-each`, this construct allows specification of action in case the predicate fails. Two child nodes are permitted: `psp:for-each` and `psp:else`, the rest is ignored. The result is empty contents should the predicate provided as an attribute to `for-each` return an empty list.

psp:for-all function="predicate(A, B, C, ...)"

The element can be considered as syntactic sugar and is identical to for-each, except that results are extracted through backtracking rather than returned as a list. Calling *findall*(X, predicate(A, B, C, ..., X), List) prior to handing results to PSP documents, usage of this tag can be completely avoided. Nevertheless, for this very reason, no for-all-else is provided: if predicate does not succeed at least once, *findall*/3 returns an empty list.

5.4.4 Dynamic construction of elements

psp:element tag="..."

Enclosing `attribute` and `content` child elements, this element provides a means of dynamically creating additional elements during page evaluation. It is particularly useful if the set of attributes is unknown beforehand and should be generated on-the-fly or the element name depends on some condition. As this may lead to infinite loops, once brought into existence, the created element is not preprocessed again, that is, one is not able to exploit dynamic generation functionality to produce recursion.

psp:attribute name="..."

One of the processed child nodes of `psp:element`, this construct appends an attribute to the element generated. The name of the attribute is as specified, while its value will contain the contents of the `attribute` element with leading and trailing whitespace removed. Dynamically generated elements may have non-dynamically generated attributes that occur in its tag as regular name-value pairs. These always precede dynamically generated tags.

psp:content

The other child node of `psp:element`, it serves to provide a further level of nesting for the actual content of the dynamically generated element. As a result of evaluation, the dynamically created element will have content identical to the content enclosed by this element.

5.4.5 Variable query and binding

psp:capture var="varname"

Creates and unifies the uninstantiated variable `varname` with the string comprising of the characters enclosed by the start and end tags with leading and trailing whitespace removed. Enclosed content is evaluated prior to being assigned to the variable. The resulting string may have zero length. The variable name may be capitalized to conform to Prolog notational convention. The assignment is local to the context, that is, parent tags are unaffected.

psp:capture content="varname"

Creates the local context variable `varname` and unifies it with the target-independent representation of the enclosed content. Essential

psp:insert var="Variable" [arg="ArgumentNumber" — coerce="true"]

The value of the local `Variable` replaces the element upon evaluation of the page. The variable must be atomic or an `arg` attribute is required to extract an attribute that is simple. On the other hand, the attribute `coerce` may be provided to indicate that compound term need to be laid out as if by printed by *write*/1.

psp:insert list="List" nth="Nth"

`List` is a local variable that satisfies the predicate *is_list*(List), that is, it is a proper ground list. The list must consist of atomic elements, the `Nth` member of which is extracted and inserted at the place of the element. `Nth` is one-based.

psp:insert expr="Expression"

Evaluates the given `Expression`, which replaces the element upon evaluation.

psp:insert goal="..."

Calls the specified goal as if by *call*/1 and inserts any output it produces into the result. Success or failure of the goal is ignored but it must not produce an exception.

psp:insert function="predicate(A, B, C, ...)"

Appends an uninstantiated argument to the specified predicate, calls the resulting single-predicate goal and inserts the value of the appended argument into the result. Acts as if it were defined as

```
<psp:capture var="temporary">
  <psp:insert goal="predicate(A,B,C,...,X), write(X)" />
</psp:capture>
<psp:insert var="temporary" />
```

5.4.6 Miscellaneous tags

psp:cached

The element identifies the enclosed content as cached content. Such content is stored as target-independent terms in memory between requests and is returned upon need. It serves to declare dynamically generated but infrequently changing sections, which saves valuable processing time by avoiding repetition of unnecessary computation-intensive tasks. It is guaranteed that enclosed content is evaluated once after having imported the server page document but it may be evaluated multiple times.

psp:import src="sourcefile"

The page specified by sourcefile is imported into a self-contained module space and its implementation is thus hidden from the importing page. Upon preprocessing, the tag is replaced by the identified content.

psp:include src="sourcefile"

The contents of sourcefile replaces the tag. Acts as if the contents of the specified source were located at the place of the element. Encoding schemes used by the including and included documents may not differ.

5.4.7 Attributes of the root node

logic-file="file"

Identifies the file containing the business logic for the page. The file specification should either be absolute or relative, relative paths are resolved with respect to the directory in which the related PSP document is located. Paths should follow the UNIX notational convention, that is, forward slashes are to be used.

output-method="method"

Specifies the output method for the page. The output method is one of the following: `html` (an HTML 4.0 compliant page is generated), `xml` (for an XML-compliant page) or `xhtml` (for an XHTML 1.0 compliant page). No action is taken to verify that the document actually conforms to the document type given.

5.5 Procedural semantics of special elements

Just as expression language elements, special elements are simplified upon preprocessing and substituted by the content they generate upon evaluation. Simplification does not apply to all elements but can typically be used to verify syntactic conformance. For instance, conditional expressions may test for a goal or an expression but not both. Checking validity at preprocessing time saves repeated verification at evaluation phase: the intermediate representation is clearly not subject to further changes that would invalidate syntactic compliance.

5.6 Extension mechanism

While the built-in elements of the `psp` namespace enable the construction of powerful web pages, an extension mechanism is provided to support additional user-defined features. Extensions are loaded and registered in prosper_configuration.pl as facts in the following manner: *extension_module*(Namespace). `Namespace` is an XML-conforming namespace specification an is identical to the name of a Prolog module that encapsulates the extension. The filename of the extension module and its name must match and all relevant predicates that have counterparts as tags need to be exported. Registered extensions are loaded automatically, no explicit *use_module*/1 declarations are required.

By registration, a binding is established between the exported predicates in the module and matching tags: upon page evaluation, elements of the name `module:predicate_name`

will cause a call to be generated to *module:predicate_name*(Attributes, Variables, Content, Terms).

- `Attributes` is a list of name-value pairs,
- `Variables` is a list of all current variable bindings available in the context of the element,
- `Content` is a ground term representing the enclosed content as discussed in Prolog XML term representation and
- `Terms` should be instantiated to the content that replaces the element.

In general, both `Content` and `Terms` are lists with a single member, *element*/3, in which other elements are nested. For instance, the predicate *table*/4 in the module *csv* might convert a comma-separated list into an HTML table:

```
<csv:table>
  1,first item
  2,second item
</csv:table>

<table>
  <tr>
    <td>1</td>
    <td>first item</td>
  </tr>
  <tr>
    <td>2</td>
    <td>second item</td>
  </tr>
</table>
```

In order to register this extension, one includes the following line in prosper_configuration:

```
extension_module(table).
```

In addition to evaluation time hooks, an extension may provide preprocessing time hooks. These have the signature *module:predicate_name*(Attributes, Content, Terms). The most significant difference is noted in the absence of the argument Variables: at preprocessing time, these are unavailable. Nevertheless, preprocessing time hooks work identical to evaluation time hooks and must accept and return arguments of the same format. Preprocessing time hooks are useful in

- checking syntax of attributes. Elements may have exclusive attributes that cannot be used simultaneously. It is wise to check for correct usage in code that executes once rather than multiple times. For instance, the built-in element `if` verifies at this phase that only one of its `goal`, `function` or `expr` attributes is specified and issues an error otherwise.

- evaluation and preprocessing of content. In some cases, simplification can be performed even if no context information is at one's disposal. For instance, parts that depend on conditions that always evaluate to false can be omitted.

- dynamic transformation of content. For instance, the built-in element `if`, depending on which argument it was invoked with, transforms itself into an `if_with_goal`, `if_with_function` or `if_with_expression` element.

Evaluation time extension hooks must call *terms_to_elements*(Content, Variables, Terms) on their content to cause an evaluation thereof. `Variables` may be identical to the list received by the hook or may be prepended by additional variables. In order to ensure proper visibility rules, `Variables` may not be changed in other ways. Preprocessing time hooks do not need to call the aforementioned predicate. Preprocessing and evaluation are handled differently as preprocessing is a greedy operation: it preprocesses all child elements prior to preprocessing the parent, while evaluation is a lazy one: evaluation of the parent precedes that of children. Lazy evaluation is necessary as content may rely on variables that are made available by the parent. For instance, the built-in element `for-each` provides the variable `iterator`.

Extensions are loaded upon container startup. Dynamic asserting of extensions is not supported.

5.7 Prolog Business Logic files

Prolog Business Logic files encapsulate logic related to a Prolog Server Page document. Multiple PSP documents may share the same logic file. Logic files are conventional Prolog modules which are compiled for faster processing. They may use external modules and foreign extensions provided that these are available either in the Prolog library or the search path but may not use any module with the prefix *prosper*. In addition, a special module is available to business logic files, namely *psp*, which provides access to automatically asserted facts such as data obtained via HTTP requests or sessions. This is also described in this section.

5.8 Environment facts

Associated with an HTTP request to retrieve a document, environment information is often available. Notably, GET requests carry information in the query string, which can be decoded into name-value pairs, whereas POST requests transfer data in the request body. Context information, such as the currently processing document or the name of the tied logic module, may be of value. Of paramount importance are session variables which represent bindings that span over multiple requests but belong to a single connection. It is thus feasible to make these data available to logic files through Prolog predicates and facts. This is achieved by the module *psp*, which is available from every module regardless of level of nesting and is accessed by prefixing; for instance, GET name-value pairs are obtained through the predicate *psp:http_get*(Name, Value). This is similar to how request variables are accessed in PHP through superglobal arrays. [18]

5.8.1 Facts for a single request

HTTP environment can be accessed through predicates of the form *http_*/*n in the module *psp*. Name-value pairs gained via HTTP GET and POST requests are short-lived in the sense that they are specific to a single request only: *http_get*/2 and *http_post*/2 facts are asserted into the program database upon initiation of evaluation and retracted upon completion thereof. *http_get*/2 and *http_post*/2 facts are local to the thread into which they are asserted; they are invisible to the outside. The predicate *http_request*/2 is provided as a syntactic sugar if the source of data does not matter. All clauses have the signature *http_name*(?Name, ?Value), that is, they can be used for listing, query or verification. For consistency, it is not recommended to dynamically alter these facts in logic files.

5.8.2 Facts for session management

Long-lived name-value pairs associated with a session are accessed through the predicate *session*(?Name, ?Value). In order to modify session variables, *session_set*/2, *session_update*/3 and *session_remove*/2 are provided. Modification is non-backtracking (destructive) and requires the value argument to be bound.

Facts related to sessions are treated differently from facts *http_get*/2 and *http_post*/2. Prior to initiating page evaluation, session variables are asserted into the thread-local program database of the executing thread using a data store that contains all session-related information. Upon completion, however, modifications via *session_*/*2,3 predicates are not discarded but replace any data previously available. Sessions do not persist between Prolog Web Container restarts.

5.8.3 Facts for managing application-level variables

Application-level variables provide another persisting mechanism. Contrary to session variables, these are neither tied to any request, nor any session and globally available from any threads at any time. Elementary synchronization is provided by means of *application_update*(+VariableName, :Predicate) to avoid one thread accessing an inconsistent state while another is performing an operation on an application variable. `VariableName` is the name of the application variable, while `Predicate` must be callable and have the signature *Predicate*(+OldValue, -NewValue). `OldValue` is unified with the current value of the application variable, the predicate must unify `NewValue` with the future value. It is guaranteed that no other thread accesses the application variable while the new value is being computed. The following example illustrates how this mechanism can be used:

```
updater_predicate(OldValue, NewValue) :-
    NewValue is OldValue + 1.

psp:application_update(counter, updater_predicate).
```

Similarly to session variables, they provide destructive assignment and do not survive system restarts.

Chapter 6

Reference implementation

Demonstrating the capabilities outlined, the paper concludes with a reference implementation in SWI-Prolog. SWI-Prolog is a Prolog implementation compliant to part one of the ISO standard with additional functionality compatible to other Prolog systems, notably Quintus and SICStus. It has been targeted for building large applications and has hence support for multi-threading, provides an extensive foreign interface to C, is bounded by few system limits and has a rich set of supplementary library modules. [24] As multi-threading is key to efficient servicing of Prolog Server Pages and inter-process communication via sockets is heavily relied on, SWI-Prolog, which has built-in support for these features, seems a straightforward choice.

6.1 Overview

The reference implementation builds up of three major components:

- *fastcgi* encapsulates code responsible for socket-based communication using the FastCGI protocol. It is implemented both in Prolog and C, the primary advantage of the former is portability and ease of debugging, while the latter is superior in terms of execution time. The two exported predicates of the Prolog implementation *fcgi_receive*/2 and *fcgi_reply*/2 marshal reading data from and writing data to a previously opened socket stream. The C implementation also relies on Prolog streams but integrates character encoding and provides a slightly lower-level interface. It is complemented by a thin Prolog layer.

- The majority of Prolog Web Container functionality is implemented in the Prolog module *prosper_server*. It creates, manages and destroys worker threads and handles incoming requests by assigning idle threads to them. It performs necessary character conversions to ensure proper transmission of data, creates and assigns session

identifiers.

- Code related to Prolog Server Pages, which builds upon Prolog Web Container services, is distributed among the source files *prosper_core*, *prosper_builtin* and *prosper_expression_language*. This separation is mostly logical rather than architectural, the files form a single module space. The only exported predicate is *display_page*/2, which calls two other predicates of primary importance:

  – *import_page*/1, when required, reads the requested PSP page from disk, transforms it into intermediate representation, while ensuring that no other thread is performing the very same task.

  – *output_page*/1 asserts request- and session-specific variables into thread-local module space, evaluates the intermediate representation, outputs text result and retracts variables. Session-specific variables checked out prior to evaluation are reasserted into their global storage upon completion thereof.

  The actual processing of terms is performed by the predicates *element_to_terms*/4, *term_to_elements*/3 and *output_element*/1. *element_to_terms*/4 transforms Prolog XML into intermediate term representation, *term_to_elements*/3 attaches context information and results in target-independent representation of output, which is written to the output stream by *output_element*/1.

The reference implementation was developed under Windows XP using Apache 2.0 as a web server with mod_fastcgi enabled. The web server connected to the implementation through the TCP port 1160 via sockets.

6.1.1 Character encoding

In SWI-Prolog, atoms and strings are represented internally as sequences of either 8-bit ASCII characters or 31-bit UCS4 characters (stored as 32-bit integers).[1] Direct transmission of such characters is inefficient and not portable. Therefore, the implementation relies on encoding these characters in accordance with the UTF-8 scheme. UTF-8 encodes Unicode characters as one- to four-byte character sequences: ASCII characters are left intact, those with character codes ≥ 128 are encoded as multi-byte sequences up to `U+10FFFF`. This enables transmission of data through protocols that support only octects without regard to byte order. [25]

The reference implementation requires all external data to be encoded in UTF-8. This includes Prolog Server Pages as stored in the filesystem, HTTP request form data and

[1] If possible, the compact representation is chosen.

any text output generated. However, transformation between UTF-8 and Prolog internal representation is transparent.

In certain contexts, such as the query string at the end of an URL, some ASCII characters are reserved. For instance, the ampersand (&) character is used to separate name-value pairs. In order to circumvent this limitation, potentially illegal characters are encoded as %`xx` sequences, where `xx` is a hexadecimal code in the range `00-ff`. Prior to using these values, this encoding must be resolved. A UTF-8 to Unicode conversion may then commence.

6.1.2 Global configuration file

Some options that govern global functionality are stored in the configuration file *prosper_configuration.pl*. This includes the port the web container is listening to, the level of debug information printed or the number of threads available for incoming requests. The configuration is interpreted upon startup, for any changes to take effect, the web container must be restarted.

6.2 Prolog Web Container

The reference implementation of Prolog Web Container, found in module *prosper_server*, is capable of operating in single-threaded or multi-threaded mode. While the latter is suited for use in regular environments, the former is particularly useful for debugging. Single-threaded operation is specified by the configuration fact *default_workers*(1) in *prosper_configuration*. In the case of a multi-threaded server, *default_workers*(N) is used to give the number of worker threads, where $\mathtt{N} \geq 2$.

The only exported predicate of the module is *process_requests*/0. Calling the appropriate predicates of the built-in module *socket* implementing a standard, C-like TCP/IP interface, it sets up the application to listen on the port as defined by *default_port*/1 in *prosper_configuration*. In addition, if appropriate, worker threads are created by *create_workers*/1 and attached to the message queue `prosper_workers`. Finally, the predicate enters an infinite loop implemented in *listen_accept*/1, which accepts inbound connections to the application.

6.2.1 Server and worker threads

The server thread is the main thread of the container application. Running in an infinite cycle, it accepts any inbound TCP/IP requests. In order to attain the highest possible performance, once a connection is accepted, any communication through that connection

is handled by the worker thread that is assigned the particular task. Any communication problems should arise, it is the worker thread that is to make the appropriate steps, closing the connection included. Task assignment is done by dispatching a message to the queue `prosper_workers` via *thread_send_message*/2, which is then received by any available idle worker thread. The message contains an only argument: the socket related to the accepted connection.

Worker threads are created by *create_workers*/1 and execute the goal *worker*/0. They are detached threads that communicate with the server (connection-accepting) thread with messages accepted using *thread_get_message*/2. Detached threads differ from regular threads in one aspect: they cannot be joined, that is, there exists no mechanism to retrieve whether the execution of the goal the thread was initiated with terminated in success or failure or caused an exception, moreover, variable substitutions on success are lost. However, in this particular case, it is not required, worker threads – just as the main server thread – run continuously in an active-idle cycle, any exchange of information is achieved via message queues. *thread_get_message*(Queue, Message) suspends the execution of the thread until a message is received on the specified `Queue`. If so, an attempt is made to unify it with `Message`; upon success, the thread resumes execution. This prevents constant polling to verify if any job is available.

6.2.2 Handling requests

The predicate *worker*/0 only implements receiving messages and terminating the thread upon exit, actual processing of requests is done in *process_request*/1. As they are detached threads, workers may not fail or cause exceptions. To this end, communication is wrapped in a *call_cleanup*/2 clause, which closes opened streams both on unexpected error or successful completion of a request.

process_request/1, relying on the underlying FastCGI implementation, processes data obtained via the protocol. Data is available in two flavors: environment variables and standard input. Hence, processing means, in particular, actions that relate to environment variables received as name-value pairs.

- *http_path*/2 extracts the `PATH_TRANSLATED` environment variable and converts its value to a canonical file path. Canonization resolves relative paths and maps back-slashes to slashes. Any constraints that should be imposed on the resulting path (directory subtree, extension, etc.) are implemented here, if they are not met, the predicate fails and the request – in turn – is denied.

- *http_post*/4 checks if the environment variable `METHOD` equals `POST`, in which case it extracts `POST` data from the standard input. Standard input in this context refers to

a FastCGI stream rather than an actual stream. Unlike in the case of CGI applications, the predicate need not check `CONTENT_LENGTH`, FastCGI signals end of stream input with an empty record. If data can be interpreted as HTML form data sent via `POST`, the content of standard input is cleared and the corresponding outbound argument is unified with an empty list.[2] For XML HTTP requests, the outbound argument for the body of the request is unified with a Prolog XML term representation.

- *http_get*/2 parses the request URI and unifies its outbound argument with a list of name-value pairs that occur. Parsing includes recognition of ampersand or semicolon delimiters, interpreting % encoding and performing an UTF-8 decoding.

- *http_session*/3 removes the session identifier if present among the extracted name-value pairs or generates a new if none is supplied. Further requests may use this session identifier.

All data obtained by processing environment variables is handed to *display_page*/2, a predicate that belongs to the module implementing Prolog Server Pages. The predicate returns no value but any output it generates will be wrapped in a FastCGI standard output stream and will be returned to the client that initiated the request with proper UTF-8 encoding.

6.3 Prolog Server Pages

The implementation of Prolog Server Pages is centralized around two distinct phases: preprocessing and evaluation of PSP pages. This is reflected in the two major goals of *display_page*/2: *import_page*/1 and *output_page*/2. The former, if it is unavailable in memory, consults the supplied PSP document, converts its XML representation into intermediate term format and stores that document in a cache for later retrieval. The latter fetches the document from the cache, asserts any context-specific environment information (including session variables) into the special module *psp*, performs its evaluation, outputs the resulting target-independent representation and retracts asserted clauses.

6.3.1 Concurrent importation

The goal of page importation is making a Prolog Server Pages document available in cache. While a relatively simple task for single-threaded applications, multi-threading introduces difficulties to overcome: multiple executing threads should not be importing and

[2] This is very similar to the Java behavior for `POST` requests.

preprocessing the very same document. To this end, *import_page*(File) after ascertaining that `File` has not yet been asserted, guarded by a mutex, attempts importation of the page (*import_test_and_set*/1). If it does not yet exist, the thread places a dynamic predicate *page_importing*(File, Thread) in the program database. The mutex is released immediately and importation begins. On the other hand, if *page_importing*(File, Thread) already exists, a message is sent to the importing `Thread` requesting notification, prior to releasing the mutex. The message is deposited in the private message queue of the importing thread and the sender suspends execution. As soon as importation completes with success or failure, the message queue is checked and all suspended threads are notified. Upon success, previously suspended threads can begin evaluation in their individual context. Upon failure, no thread should perform a repeated importation and all return with failure. Notification is protected by the aforementioned mutex to avoid other threads hooking the importer's message queue while it dispatches completion notifications but *page_importing*/2 is no longer in the database.

6.3.2 Page preprocessing

Importation consists of two distinct phases: loading an XML document into Prolog (to yield XML term representation) and preprocessing it (intermediate representation). To attain loading the document, any suitable parser may suffice.[3] Preprocessing is implemented in the predicate *element_to_terms*/4. It recognizes tags with the prefix `psp` and transforms them into corresponding intermediate term representations. Registered extension prefixes are also recognized. In order to avoid ambiguities, prefix names are fixed, even though proper `xmlns` namespace specification is required at the beginning of a document. Preprocessing-time hooks, if they exist, are called, this is done by *ensure_hook*/4. All built-in functional elements have preprocessing-time hooks. Error messages that may result from syntactic non-conformance (missing or contradictory attribute specification, missing nested elements, etc.) are embedded into the result as `failure` elements by *error_term*/2,3. Such warnings and errors can be sup

6.3.3 Page evaluation

Performed by *output_page*/2, page evaluation itself may be broken into three processes: assertion of global context-specific information (*http_get*/2, *http_post*/2, *session*/2, etc.), actual evaluation and retraction of context-specific information. *terms_to_elements*(Terms,

[3]Currently, the SWI library module *sgml* performs this task. One of its primary disadvantages is that it attempts to interpret ill-formed XML documents, which may be desirable for HTML content but clearly leads to prolonged debugging in the case of Prolog Server Pages. Instead, the place of error should be precisely reported.

Variables, Results) performs the main task.

- `Terms` is a list of intermediate term representations, which may be one of
  - *element*/3 for static elements,
  - *builtin*/3 for built-in elements of the `psp` namespace,
  - *extension*/3 for user-defined extensions as defined in the configuration file,
  - *cache*/2 for content which is stored in target-independent rather than intermediate term representation,
  - *capture_simple*/2 for content that is captured in a variable as a string,
  - *capture_element_list*/2 for content that is captured preserving its structure,
  - *insert_content*/1 for insertion of captured content,
  - *insert_var*/1,2 for insertion of atomic or nth argument of compound terms,
  - *insert_list*/2 for insertion of an nth argument of a list,
  - *insert_goal*/1 for execution of a goal whose output is inserted,
  - *insert_function*/1 for execution of a single-predicate goal, whose last, outbound argument is inserted,
  - *insert_expr*/1 for insertion of the result of an expression.

  The first three – *element*/3, *builtin*/3 and *extension*/3 – may nest other intermediate terms in their last, `Content` argument.

- `Variables` is a list of local variables defined in the current context. During evaluation, *term_to_elements*(3), which is called by *terms_to_elements*(3) for each single intermediate term, passes this list to any nested element. These elements may prepend the list by variables that they define within their content. For instance, `for-each` prepends the variable `iterator`. It is possible to isolate page fragments by passing an empty list but is not recommended for the sake of consistence. Variants of *capture*/2 are treated specially: their validity extends only to those siblings that follow the declaration. `Variables` is an empty list for the root element, that is, implicit (globally scoped) variables are not propagated through this prepending mechanism even though they may be viewed as if it were so.

- `Results` is a list target-independent terms the evaluation produces.

A second pass on EL expressions whose evaluation ended with potential success is also attempted at this phase. However, this time all such expressions must reduce to a constant. Failure may indicate that the expression

- either refers to a global variable that does not exist
- or to a local variable that is undefined in the context.

A failure causes the expression to evaluate to `null`, which is propagated into the output.

6.4 Performance evaluation

While primarily designed to increase designer and programmer performance, Prolog Server Pages architecture is comparable to other Prolog-based technologies in terms of speed. In a simple loopback scenario, different configurations were polled by HTTP requests with GET parameters. All configurations parsed the query string and returned a simple computed result: Prolog Server Pages produced the `prosper_info.psp` page, which prints asserted get, post and session data and all available environment variables, other configurations evaluated the passed expression by means of *is*/2 and printed the result. CGI and FastCGI-based applications connected to Apache/2.0.54, Java-based solutions used Apache-Coyote/1.1. Benchmarking was performed by ApacheBench 2.0.41 with a concurrency level of 3 and 1000 complete requests. (Table 6.1) [1]

configuration	**requests/sec**
SWI-Prolog as a CGI application	35
PrologBeans in conjunction with JSP	35
Prolog Server Pages with FastCGI implemented in Prolog	45
Prolog Server Pages with FastCGI implemented in C	200
static html content	580

Table 6.1: Throughput for different Prolog configurations as measured in requests per second (rounded to nearest multiple of 5)

6.5 Sample application

The full capabilities of Prolog Server Pages are demonstrated by means of a sample web application. The application core (business logic) performs parsing and analysis of postal addresses and – connecting to an external data source – produces lists of addresses that are potentially identical to those given as input. Extending core functionality, the web interface enables its user to supervise the process of address identification: he may enter an address, see the results of parsing, modify address entries as needed, verify if unique

identification based on the data supplied can be carried out. If not, he may manually select a list entry from among addresses thought to be potentially identical by the system to the one in question. Alternatively, he may supply additional entries to the postal address to narrow the results. Multiple output formats (`xhtml` or `xml`) cater for multiple usage scenarios (regular browsers or client-side scripting-enabled browsers).

The sample application makes use of all features offered by the Prolog Server Pages Extensible Architecture. Its core module is completely independent from the web interface and may be run separately from the web application. Web pages displayed to the user exploit the basic functionality of standard elements, including conditional processing, iteration, inclusion of external fragments and dynamic element generation. Variables and expression language is heavily relied on to formulate context-specific conditions. Information gained via user interaction is propagated through requests by means of sessions until the desired results are found. A special element to produce address list entries is added to demonstrate the use of the extension mechanism.

Chapter 7

Evaluation

By designing Prolog Server Pages Architecture, the author has provided a comprehensive support for creating web interfaces for Prolog applications. Integrating the benefits of different technologies (PiLLoW, FastCGI, server pages), a two-layered architecture has been put forward. The author hopes that is will facilitate the easier development of middle-sized applications intended for the web.

Prolog Web Container, the lower-level layer, hides protocol-specific functionality from the programmer. Extending the support offered by existing libraries, it provides a more natural access to HTTP context by automating parsing of environment variables and asserting data into confined Prolog module space. The container is long-lived, which allows for persistent data through sessions and globally shared application variables. By relying on the FastCGI protocol for communication with the web server, it makes Prolog web solutions flexible and independent.

Prolog Server Pages, the higher-level layer, exploits the opportunities in separating model and view. By doing so, not only can existing Prolog applications be extended with web interfaces easily but also design and development of code is separated. A standard set of server page tags, commonly found in many other technologies is provided with an extension mechanism to supplement built-in functionality with user-defined behavior and an expression language facilitates easier insertion of simple computed values. Prolog Business Logic files, attached to Prolog Server Pages encapsulate application logic and isolate it from interface design while making use of the extensive module system of Prolog.

The reference implementation has shown that the architecture described is sound and compares to other available Prolog technologies in terms of ease of development and speed. The sample application has demonstrated that middle-sized solutions can indeed be developed using Prolog. While it is intended to complement Prolog applications with web interfaces rather than replace technologies based on imperative languages, Prolog Server Pages might also be seen as an alternative when developing new applications.

7.1 Future work

In creating a full-fledged web application, database access is of primary importance. Prolog itself provides an internal mechanism for storing data by means of facts in the form *relation*(data1, data2, data3, ...). Facts provide a seamless integration of data into a program by allowing the toolset of the language to operate on them. While the power of Prolog to handle large amounts of data is often underestimated, strong persistence requirements may make it necessary to make use of an external relational database. In addition, relational databases often provide a more efficient data retrieval, a more flexible lookup and a faster and significantly easier management of frequently changing data. On the other hand, Structured Query Language (SQL), a standard declarative way of accessing and manipulating RDBMS data, is inadequately integrated into Prolog. To remedy the situation and bridge the gap between RDBMS and Prolog, it is feasible to utilize a Prolog-to-SQL compiler that transforms Prolog terms into SQL queries, which are – in turn – used to retrieve data against the database. This has several advantages:

- Only those records are fetched that match a search criterion. This dramatically reduces Prolog memory overhead, which requires all data to be simultaneously present.
- Prolog constructs (conjunction, disjunction, negation, etc.) can be used to formulate a term that defines required data. Dynamic creation of such terms is possible without the danger of undesired SQL injection.
- Different SQL dialects can be seen uniformly.

One notable implementation of this schema is the ProDBI interface featuring the Prolog-to-SQL compiler by Draxler. [15] Draxler's compiler distinguishes two forms of representation: relation and view level. Relation level is the logical correspondent of database tables (relations), while views incorporate higher-level abstraction: restrictions, joins, aggregate functions. Views allow usage of a wide range of Prolog syntax, notably conjunction, disjunction, negation and comparisons, which are then translated to their equivalent SQL queries, thereby avoiding fetching database records that would be discarded by later processing. An illustration is given by the example below:

```
query(
   medical(A,B,C,D,E),
   (  doctor(A, B, C, D, E),
      \+ floor('1st floor', B),
      \+ (A = 'd001'),
      E > avg(ChargePerMin, A1 ^ A2 ^ A3 ^ A4 ^
```

```
         (doctor(A1, A2, A3, A4, ChargePerMin)))
   )
).

SELECT rel1.DoctorId, rel1.FloorId, rel1.DoctorName,
       rel1.PhoneNo, rel1.ChargePerMin
FROM doctor rel1
WHERE
   rel1.FId = ? AND
   NOT EXISTS (
      SELECT *
      FROM Floor rel2
      WHERE rel2.FloorName = '1st floor' AND
            rel2.FloorId = rel1.FloorId
   ) AND
   rel1.Did <> 'd001' AND
   rel1.ChargePerMin > (
      SELECT AVG(rel3.ChargePerMin)
      FROM Doctor rel3
   )
;
```

While the representation is powerful is creating equivalent joins through WHERE clauses by means of identical uninstantiated variables, incorporating SQL-level restrictions by means of atoms, numbers and comparison operators, it lacks, in particular, the ability

- to allow natural, easy-to-read definition
- of representing arbitrarily complex joins
- of unlimited nesting of individual SELECT queries
- of advanced string manipulation
- to rename arguments in the result

Therefore, in order to supplement Prolog Server Pages with a data access layer, a new schema, utilizing the term expansion facilities provided by most Prolog implementations, should be devised. Building on available low-level RDBMS interfaces, it should provide a higher level of abstraction yet preserve the full expressive power of SQL.

Bibliography

[1] *Apache benchmarking tool*. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] Module mod_fastcgi. http://www.fastcgi.com.

[3] *PrologBeans*. http://www.sics.se/sicstus/docs/latest/html/sicstus/PrologBeans.html.

[4] SICS*tus Prolog manual*. http://www.sics.se/sicstus/docs/latest/html/sicstus/.

[5] The CGI specification. http://hoohoo.ncsa.uiuc.edu/cgi/interface.html.

[6] *Velocity*. http://jakarta.apache.org/velocity/.

[7] Eric Armstrong et al. The J2EE 1.4 Tutorial (For Sun Java System Application Server Platform Edition 8.1 2005Q2 UR2). Sun Microsystems, June 7, 2005.

[8] Tim Bray et al. Extensible Markup Language (XML) 1.0 (Second Edition), October 6, 2000. http://www.w3.org/TR/2000/REC-xml-20001006.html.

[9] Mark R. Brown. FastCGI specification. Technical report, Open Market, Inc., 29 April 1996. Document Version: 1.0.

[10] Daniel Cabeza and Manuel Hermenegildo. The PiLLoW Web Programming Library. Technical report, The CLIP Group, School of Computer Science, Technical University of Madrid, January 5, 2001.

[11] David Carmona. Programming the Thread Pool in the .NET Framework. *Microsoft Developers' Network*, June 2002.

[12] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4), April 1989.

[13] James Clark. XSL Transformations (XSLT) Version 1.0, November 16, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[14] Danny Coward and Yutaka Yoshida. Java Servlet specification (Version 2.4). Sun Microsystems, November 24, 2003.

[15] Christoph Draxler. A Powerful Prolog to SQL compiler. Technical report, CIS Centre for Information and Language Processing, Ludwig-Maximilians-Universität München, August 16, 1993.

[16] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. The Intenet Society, June 1999.

[17] Jesse James Garrett. Ajax: A new approach to web applications, February 18, 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[18] Gabor Hojtsy. PHP *manual*, 31 March 2005.

[19] Myunghwa Kang et al. WEBIO Library for Executing Application Programs on the Internet. Technical report, Graduate School of Information and Telecommunications, Sangmyung University Seoul, 1999. IEEE TENCON.

[20] Monte Ohrt and Andrei Zmievski. *Smarty – the compiling* PHP *template engine*, March 31, 2005.

[21] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.0 Specification, April 24, 1998. http://www.w3.org/TR/1998/REC-html40-19980424.

[22] Mark Roth and Eduardo Pelegrí-Llopart. JavaServer Pages specification (Version 2.0). Sun Microsystems, November 24, 2003.

[23] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2003.

[24] Jan Wielemaker. SWI-*Prolog manual*.

[25] F. Yergeau. RFC 3629 - UTF-8, a transformation format of ISO 10646. Technical report, The Internet Society, Network Working Group, November 2003.

Appendix A

Expression language functions

A.1 Comparison operators

A =:= B	equal to
A eq B	equal to (xml-compliant syntax)
A = \ = B	not equal to
A neq B	not equal to (xml-compliant syntax)
A < B	less than
A lt B	less than (xml-compliant syntax)
A =< B	equal to or less than
A lte B	less than or equal to (xml-compliant syntax)
A >= B	greater than or equal to
A gte B	greater than or equal to (xml-compliant syntax)
A > B	greater than
A gt B	greater than (xml-compliant syntax)

A.2 String comparison operators

A == B	identical to
A iden B	identical to (xml-compliant syntax)
A \= B	not identical to
A niden B	not identical to (xml-compliant syntax)

A.3 Arithmetic operators

Special constants

e	the base of natural logarithm
pi	one half times the perimeter of a unit circle

Elementary arithmetic

A + B	addition
A - B	subtraction
A * B	multiplication
A / B	division
A // B	integer division
-A	unary minus
A rem B	remainder
A mod B	modulo: A - A // B * B
sqrt(A)	square root of A

Exponentiation and logarithm

exp(A)	e ** A
A ^ B	exponentiation: A raised to the power B
A ** B	exponentiation
log(A)	natural logarithm
log10(A)	base 10 logarithm

Trigonometric

sin(A)	sine
cos(A)	cosine
tan(A)	tangent
asin(A)	inverse sine
acos(A)	inverse cosine
atan(A)	inverse tangent
atan(A, B)	arctan(B/A)

Bitwise operators

msb(A)	most significant bit: equivalent to floor(log(A)/log(2)) but more efficient
\ A	bitwise negation
bw_not A	bitwise not
A /\ B	bitwise and
A bw_and B	bitwise and
A \/ B	bitwise or
A bw_or B	bitwise or
A xor B	bitwise exclusive or
A << B	bitwise left shift
A ls B	bitwise left shift
A >> B	bitwise right shift
A rs B	bitwise right shift

Floating-point numbers

float_integer_part(A)	integer part
float_fractional_part(A)	fractional part
ceiling(A)	smallest integer larger than or equal to
floor(A)	largest integer smaller than or equal to
round(A)	round to the nearest integer

Miscellaneous

sign(A)	sign of A
abs(A)	absolute value of A
min(A, B)	the least of A and B
max(A, B)	the greatest of A and B

A.4 String functions

prefix(A, B)	true if the prefix of A is B, fail otherwise
suffix(A, B)	true if the suffix of A is B, fail otherwise
contains(A, B)	true if A contains B, fail otherwise
concat(A, B, C, ...)	the string concatenation of A, B, C, etc.
trim(A)	A with leading and trailing whitespace removed
replace(A, B, C)	all occurrences of B replaced with C in A

Appendix B

Deployment instructions

During the development of the Prolog Server Pages Architecture the author used SWI-Prolog 5.5.33 running on Windows XP. Requests were served by Apache 2.0.54 running in conjunction with mod_fastcgi 2.4.2. Data requests were processed by Microsoft SQL Server 2000 8.0.760. All processes ran on the same machine.

In order to run Prolog Server Pages Extensible Architecture with the default configuration – identical to that used in the development of the framework – one must possess an operational installation of FastCGI-enabled Apache 2.x series. The web sites `http://httpd.apache.org` and `http://www.fastcgi.com` give details on how to perform the installation. In particular, the Apache configuration file needs to contain the following global directives:

```
LoadModule fastcgi_module modules/mod_fastcgi.dll
FastCgiExternalServer <path> -host localhost:1160
```

As explained in the FastCGI documentation, `path` is virtual: all requests that are resolved by Apache to point to this directory are handled by the external application listening to port 1160 rather than the web server itself. Nevertheless, in the case of Prolog Server Pages, the directory must correspond to an actual entry in the filesystem as this path is used by the achitecture to locate server pages files and logic modules with relative paths.

To test the sample application, an ODBC-enabled database server must also be running on the same machine as Prolog Server Pages, accessed with the name `mqis`. The attach and detach scripts provided with the sample application can be used to deploy the database the application requires.

Prolog Server Pages Extensible Architecture must be started prior to accessing any Prosper-processed pages by consulting the module *prosper_server* and executing the predicate *go*/0.