Adding Logic Programming Behaviour to the World Wide Web

Seng Wai Loke

1998

Submitted in total fulfilment

of the requirements of the degree of

Doctor of Philosophy

Department of Computer Science

School of Electrical Engineering and Computer Science

The University of Melbourne
AUSTRALIA

Abstract

Imperative programming has largely dominated both aspects of *Web programming*: adding sophisticated interactive behaviours to the Web and constructing programs which interact with the Web. Most mobile code languages such as Java are based on the imperative programming paradigm. Imperative languages are widely used for building Web browsers and information gathering tools.

The focus of much programming language research has been on raising the level of abstraction. *Logic programming languages*, which view computation as deduction from a set of axioms, is at a higher level of abstraction than imperative programming languages enabling a problem or subject domain to be modelled without focusing on the computer's Von Neumann architecture. Logic programming with program structuring abstractions has shown its utility in a variety of applications including expert systems, Artificial Intelligence problem solving, and deductive databases. Implementations of logic programming such as Prolog have features not found in traditional imperative languages including ease of meta-programming, backtracking search, and dynamic database manipulation, and favourable features such as automatic memory management and modularity.

Compared to imperative programming languages, there has been little exploration of logic programming languages for Web programming. Only in the last few years has work begun on the relationship between logic programming and the Web. This thesis investigates *LogicWeb*, a model of the Web as a collection of inter-related logic programs. LogicWeb adds logic programming based interactive behaviours to the Web, and enables the manipulation of Web information without focusing on details of networking and data extraction.

With LogicWeb, a Web document is viewed as a *live information entity* able to respond to user queries using its own rules and can have the behaviours of its links determined by rule-based reasoning. Processing of user queries and link

behaviours can involve collecting and manipulating data from other Web documents. LogicWeb also adds to Web documents information in the form of deductive databases and knowledge-bases.

A new language (extending Prolog) is developed based on LogicWeb for coding the logic programming behaviours, and for representing and querying the Web-situated databases and knowledge-bases. This language offers a new Web programming paradigm, where computing with the Web is equated to evaluating goals in compositions of programs. The practical result is that programming with the Web in this language is aided by familiar modularity abstractions, and the programmer need not explicitly deal with low-level issues such as document retrieval, caching, and parsing. An operational semantics is given for the language.

A system realising LogicWeb is implemented by integrating a public domain Prolog system with an off-the-shelf Web browser. Security is an important issue in the LogicWeb system because logic programs downloaded from foreign hosts are executed locally. A flexible and precisely specified security model for the system is developed.

A range of examples illustrates LogicWeb-based programmable behaviours, and demonstrates the feasibility and advantages of the LogicWeb language for coding Web search tools, Web-situated databases called *lightweight deductive databases*, and extensions to the semantics of Web links.

Acknowledgements

This thesis would not have existed without the invaluable guidance, support, continual encouragement, and writing lessons (for Prolog and English) from my two supervisors - Leon Sterling and Andrew Davison. Their confidence in me helped when mine waned. Leon led me to think about citations for the main thesis applications, and provided helpful feedback and thought-provoking questions on LogicWeb. In late March 1995, at my first visit to his office, Andrew suggested that I could perhaps explore an idea he had in mind, which was to view Web pages as logic programming modules and Web links as relationships. This view later became the basic idea of LogicWeb. I am grateful for the many interesting discussions with Andrew throughout the years, the first one and a half years over cappuccino and tea, and the rest over the Internet, as the details, applications, and realisation of LogicWeb were being thought out. Leon gave instructive and prompt feedback on the thesis drafts. Andrew provided prompt feedback on many details in the thesis drafts and suggested many improvements to the presentation. I am also grateful to Lee Naish for a number of important points on a draft of the thesis. Leon, Andrew, and Lee formed my advisory committee.

I am thankful for the enjoyable conversations with (past and present) members of the Intelligent Agent Laboratory over technical and non-technical matters, particularly Andrew Cassin, Sharon Gao (who commented on drafts of several chapters of the thesis), Dinh Que Tranh, Stewart Baillie, Hoon Kim, and Yi Han. Thanks go to Leon for encouraging my foray into the exciting area of software agents, and to Andrew Cassin for systems help on numerous occasions.

I am glad to have shared many postgraduate student experiences with Vincent Tam (who soon will not be a student!) over the many relaxing lunch and tea breaks.

Thanks go to Michael Bieber and anonymous referees (of papers on Log-

icWeb, particularly [127, 128, 126, 130, 129]) for the encouraging feedback on LogicWeb. Roland Yap performed one of the first installations of an early version of the LogicWeb system outside Australia, and helped with debugging. I am also fortunate to have had interesting email exchanges with two experts in logic program composition operators - Antonio Brogi and Michele Bugliesi.

I am grateful for the financial support throughout my candidature from a Melbourne Research Scholarship sponsored by the University of Melbourne and the University of Melbourne Alumni, and for help from the efficient and friendly administrative and technical support staff of the Department of Computer Science.

Many thanks go to my friends in CCM and HOG for their continual support and encouragement throughout the thesis. Let's achieve our vision!

My gratitude goes to my parents who have always encouraged my education, and persevered with me throughout this work.

Last but not least, I would like to thank God for making this thesis possible.

Dedication

To my parents, and the memory of my grandmother who left us in 1995.

"Representation is the essence of programming."

- Frederick P. Brooks, Jr.,

THE MYTHICAL MAN-MONTH

Contents

1 Introduction **1**

1.1 Integrating Logic Programming Technology with the Web 6

1.2 Contributions 7

1.3 Overview of Thesis 9

2 Background **11**

2.1 Technical Overview of the World Wide Web 11

2.1.1 Client-server Architecture 11

2.1.2 The Web as a Hypertext System 12

2.1.2.1 Uniform Resource Locators (URLs) 13

2.1.2.2 Hypertext Mark-up Language (HTML) 15

2.1.2.3 Hypertext Transfer Protocol 18

2.2 Logic Programming 23

2.2.1 Syntax and Terminology 23

2.2.2 Meta-programming in Logic Programming 24

2.2.3 Compositional Logic Programming 26

2.2.3.1 Meta-level Program Composition Operators . . . 29

2.2.3.2 Syntax 31

2.2.3.3 Operational Semantics 32

2.2.3.4 Implementation 35

3 LogicWeb **37**

3.1 A Logic Programming Model of the Web 37

3.2 LogicWeb Programs 40

3.3 The LogicWeb Language 45

3.3.1 Querying and Manipulating LogicWeb Programs 46

3.3.1.1 Context Switching 46

3.3.1.2 Composing LogicWeb Programs 51

3.3.1.3 Utilising the Current Context 54

3.3.2 EBNF Syntax 56

3.3.3 An Operational Semantics for LogicWeb Programs 57

3.3.3.1 Pure Prolog 61

3.3.3.2 Clauses from LW-compositions 62

3.3.3.3 Context Switching 64

3.3.3.4 An Example Top-down Derivation 66

3.3.4 Relationship of Operational Semantics to Declarative Semantics 67

3.4 Building Applications Using LogicWeb Programs 68

3.5 Summary 69

4 Implementing LogicWeb **71**

4.1 The LogicWeb System Architecture 72

4.2 Implementation Overview 73

4.3 System Behaviour 78

4.4 The Prolog Engine 83

4.4.1 Mapping User Actions to Goals 83

4.4.2 The LogicWeb Program Interpreter 90

4.4.3 Translation into LogicWeb Programs 94

4.4.4 Caching LogicWeb Programs 95

4.5 Discussion 97

5 CIFI **101**

5.1 Looking for Citations on the World Wide Web 101

5.2 Design and Implementation of CIFI 105

5.2.1 Alternative Strategies 105

5.2.2 Search Algorithm 106

5.2.3 Obtaining Starting Points and Link Selection 109

5.2.4 Extracting the Citation 111

5.2.5 Integrating Other Information Sources 112

5.2.6 Implementation 112

5.3 Limitations of CIFI 113

5.4 Related Work 115

5.4.1 Agents for Paper Search 115

5.4.2 Web Search Tools 116

5.4.2.1 Internet Fish 116

5.4.2.2 General Heuristics Involving Web Links 117

5.4.2.3 An Abductive Framework for Web Searching . . 117

5.4.2.4 Browsing Agents 117

5.5 Discussion 118

6 Lightweight Deductive Databases **125**

6.1 A Simple Lightweight Deductive Database 128

6.2 Combining and Extending Lightweight Deductive Databases . . . 129

6.2.1 Virtual Relations and Relational Joins 129

6.2.2 Forming Virtual Databases Using LW-composition Operators 131

6.3 Knowledge-based Querying of Citation Databases on the Web . . 134

6.3.1 Organising Citation Information on the Web 134

6.3.2 Searching for Citations 135

6.3.3 Representing Knowledge 137

6.3.4 An Implementation of Citation Finding 138
6.3.5 Summary 139
6.4 Generating Guided Tours 140
6.4.1 Structure of a Guided Tour and a Guided Tour Application 141
6.4.2 Tour Generation 146
6.4.2.1 Tour Generation Using Static Information 146
6.4.2.2 Domain Knowledge as Static Information 147
6.4.2.3 Tour Generation Using Dynamic Information . . 148
6.4.3 More Complex Tour Generation 150
6.4.3.1 Tours by Appending Lists 150
6.4.3.2 Tours Containing Tours 151
6.4.3.3 Tours Constructed Using User Inputs 152
6.4.4 Implementation 156
6.4.5 Other Work on Generating Guided Tours 157
6.4.6 Summary 158
6.5 Server-side Databases 158
6.6 Related Work 161
6.6.1 Database on Web Pages 161
6.6.2 Standardised Knowledge-bases on the Web 162
6.6.3 Marked-up Text on the Web 162
6.6.4 Comparison with Deductive Database Systems 164
6.6.5 Knowledge-based Access to Information 164
6.7 Summary 166

7 Extending the Semantics of Web Links 167
7.1 The Two-level Web Model 168
7.2 Utilising Structured Information for Linking 169
7.2.1 IS-A Hierarchy Links 169
7.2.2 Page Name Links 174

7.2.3 Linking Based on Logical Relationships Between Pages . . 175
7.2.3.1 Structural Relationships 175
7.2.3.2 Temporal Relationships 179
7.2.4 Links Based on Page Information 181
7.2.5 Dynamically Constructing Pages 182
7.3 Handling Nondeterminism in Web Links 183
7.3.1 Redirection Pages 184
7.3.2 Broken Links 185
7.4 History-based Linking 187
7.5 Using Multiple Link Behaviours 188
7.5.1 LogicWeb Operators 189
7.5.2 Link Transducers 190
7.6 System Link Actions 192
7.7 Related Work 194
7.7.1 Hypertext Tools 194
7.7.2 Web-based Tools 194
7.7.2.1 Structured Maps 194
7.7.2.2 CGI 195
7.7.2.3 JavaScript 195
7.7.2.4 Link Management Systems 196
7.8 Discussion 196

8 Security in the LogicWeb System 199

8.1 What are the Security Issues in the LogicWeb System? 199
8.2 Overview of Security Model 201
8.3 Digital Signatures for LogicWeb Programs 204
8.4 Specifying Security Policies 206
8.5 Combining Security Policies 209
8.6 Enforcing Security Policies 212

8.7 Implementation . . . 219
8.7.1 A New Interpreter . . . 219
8.7.2 Installing Programs . . . 220
8.7.3 Invoking the New Interpreter . . . 225
8.8 Control of Resource Usage . . . 225
8.8.1 Resource Control Using Policy Programs . . . 225
8.8.2 Resource Control Using Meta-interpreters . . . 226
8.8.2.1 Loop Checking . . . 226
8.8.2.2 Two Resource Limits . . . 228
8.9 Comparison with Security Models in Other Mobile Code Systems 230
8.9.1 Security Models in Two Interpreted Languages . . . 230
8.9.1.1 Safe-Tcl . . . 230
8.9.1.2 Java Applets . . . 231
8.9.2 Security Policy Modules in Two Mobile Code Systems . . . 232
8.9.2.1 SERC's Safer Erlang (SSErl) . . . 232
8.9.2.2 Java Aglets . . . 233
8.9.3 Authentication in Two Mobile Code Systems . . . 233
8.9.3.1 Agent Tcl . . . 233
8.9.3.2 ActiveX . . . 234
8.10 Summary and Future Work . . . 234

9 Comparison With Related Work **239**
9.1 Logic Programming Technology for the Web . . . 239
9.1.1 Client-side Systems . . . 240
9.1.1.1 Prolog and Client-side Programming . . . 240
9.1.1.2 Logic-based Web Querying Languages . . . 243
9.1.2 Server-side Systems . . . 246
9.1.3 Peer-to-peer Systems . . . 250
9.2 Other Web Programming Languages . . . 252

9.2.1 SQL-based Web Querying Languages 253
9.2.2 Languages Modelling the Web's Nondeterministic Nature 256
9.3 Logic-based Hypertext Models 257

10 Conclusion 259
10.1 Language Extensions 262
10.1.1 Extending the Semantics of Downloading 262
10.1.2 Other Header Fields in HTTP Requests 263
10.1.3 Lazy Download 264
10.1.4 Concurrency 265
10.1.5 Operations On the Program Store 266
10.1.6 Application-specific LW-composition Operators 266
10.1.7 Multiple Page Models 267
10.2 Using Standardised Mark-up 268
10.3 Applications 268

A Prolog Facts Storing the Title, Body, Sections, and Links to Images and Applets on a Page 303

B Application Support Library 307
B.1 Displaying Information on the Mosaic Browser 307
B.2 Constructing HTML Components 309
B.3 Fast String Matching 310
B.4 Comparing Dates 310
B.5 Determining If a Program Exists in the System 311
B.6 Deleting Programs 311

List of Figures

2.1 The client-server architecture of the Web. 12
2.2 John Smith's homepage when rendered on a Web browser. 16
2.3 The HTML mark-up of John Smith's homepage. 17

3.1 Web pages connected by hypertext links. 38
3.2 The LogicWeb model where pages augmented with rules are programs and hypertext links are relationships between programs. . 39
3.3 The identifiers and components of three types of LogicWeb programs. 41
3.4 A page with rules describing research interests. 44
3.5 A derivation of the LogicWeb goal `(M + N)#>p(a)` in a program `P`. 66
3.5 (Continued) 67
3.6 A LogicWeb application with the programs it uses directly and indirectly. 70

4.1 Architecture of the LogicWeb system. 72
4.2 An overview of the main components and data-flows (arrows) between them. 74
4.3 The interface of a simple LogicWeb application. 76
4.4 The result of a query with keyword "marsupial". 77
4.5 The LogicWeb system and the steps followed after a user clicks on a link. 79

4.6 The LogicWeb system and the steps followed after a user enters a query. 81
4.7 The LogicWeb system and the steps followed after a user issues a "back" or "forward" command in Mosaic. 82
4.8 A simple LogicWeb application. 86

5.1 The interface to CIFI. 113
5.2 The result of a search. The required citation is the first citation in the displayed page fragment. 114

6.1 A representative diagram of the hypertext structure rooted at a departmental homepage. The arrows denote the sequence in which the various page types are reached, starting from the `dept` page. . . 136
6.2 A hierarchy of sections, groups and projects. The edges represent the `has_part/2` relationships. 137
6.3 The structure of a guided tour of lecturers' homepages in the University of Melbourne Computer Science department. The arrows represent the links of the tour; the rectangles are Web pages. . . . 142
6.4 The index node for the tour of lecturers' homepages. 143
6.5 A typical tour node. 144
6.6 The guided tour application consists of a set of LogicWeb programs (represented by the boxes). An arrow indicates a "uses" relationship. 145
6.7 The user interface to the Asian tour planner. 153
6.8 A guided tour constructed with sites in Thailand, Malaysia, Indonesia, and Taiwan, for 3, 4, 3, and 4 days respectively. 154
6.9 A server-side database and its interface. 159

7.1 The two-level Web model. A link abstraction layer separates the source and destination of links. 168

7.2 An IS-A hierarchy. The dashed arrow shows a link referring to a concept in the hierarchy. The dotted arrows show mappings from concepts to Web pages as specified by `page_about/2`. 170
7.3 The page constructed when the link to "agents" is selected. 173
7.4 Components for handling a link selection. 193

8.1 The security model with two concentric sandboxes. The inner sandbox consists of the LogicWeb program interpreter and policy programs, and the outer sandbox consists of meta-interpreters. 203
8.2 The use of a digital signature when sending a program from A to B, and the subsequent assignment of a policy program. 205
8.3 The invocation (with `P`'s policy consulted) of the LogicWeb goal `Q#>read_file(Contents)` in `P` leads to the invocation of the goal `read_file(Contents)` in `Q` (where both the policy of `P` and that of `Q` must be consulted). 210
8.4 The change in the context (represented by the rectangle) of goal evaluation starting from the goal `Q#>(R#>G)` in program `P` and ending in goal `G` in program `R`, and the policies for validating each subgoal. 211

9.1 Using Prolog CGI scripts. 247
9.2 Separating the interface and task processes. 248
9.3 A dedicated logic programming server. 249

List of Programs

4.1 The predicate which translates user inputs into goals. 88

4.2 The interpreter for pure LogicWeb programs. 91

8.1 The interpreter for pure LogicWeb programs modified to use policy programs. This program extends Program 4.2. 221

8.1 (Continued) 222

8.2 A version of `solve_t/1` for pure Prolog. 227

8.3 A meta-interpreter for pure Prolog with an argument carrying the recursion depth and a list of ancestor goals. 228

Preface

Preliminary work on LogicWeb was published in the Proceedings of the 7th ACM Conference on Hypertext in March 1996 [126]. A short introduction to LogicWeb was published in the Proceedings of the 2nd Joint AUUG and Asia Pacific World Wide Web Conference in September 1996 [125]. Chapter 5 uses material on CIFI from a paper published in the Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence in August 1996 [131], and uses introductory material of a paper on a later version of CIFI published in the Proceedings of the 2nd Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology in April 1997 [97]. Chapter 6 expands on material from a paper introducing lightweight deductive databases which was published in the Proceedings of the 1st Workshop on "Logic Programming Tools for Internet Applications" held in conjunction with the Joint International Conference and Symposium on Logic Programming in September 1996 [130], and a paper on guided tours published in the Proceedings of the 5th International Conference and Exhibition on the Practical Application of Prolog in April 1997 [128]. Chapter 7 is an extended version of a paper on a two-level Web model published in the Proceedings of the 2nd Workshop on "Logic Programming Tools for Internet Applications" held in conjunction with the 14th International Conference on Logic Programming in July 1997 [127]. An overview of LogicWeb incorporating selected materials from Chapters 3, 4, 5, 6, and 9 has been accepted for publication in the Journal of Logic Programming [129]. Much of the related work in Section 9.1 is based on the related work section in this journal paper. Using material on an early version of this journal paper, a short introduction to LogicWeb was published in the Proceedings of the 20th Electrical Engineering Conference of Thailand in November 1997 [65]. The LogicWeb homepage contains information taken from some of the papers:
`http://www.cs.mu.oz.au/~swloke/logicweb.html`.

The work in this thesis has not been published elsewhere, except as noted above.

Seng Wai Loke
April 1998, Melbourne, Australia

Chapter 1

Introduction

The World Wide Web [24], or *the Web* for short, is one of the most prominent technological innovations in the area of Internet computing. Invented in 1989, the Web is today a widely used repository of knowledge and medium for information sharing and transport over the Internet. On the Web, an individual can find information on almost anything and add content for world-wide consumption. The individual sees the Web as a hypertext system comprising a vast, interlinked repository of documents, and using a locally running program called a *Web browser*, can follow links from one document to another without needing to know details of networking.

While growing rapidly[1], the Web is evolving technologically. Considerable research and developments are under way with new applications and functionality being thought of for the Web. Areas of development include communi-

[1] According to a Web growth survey by Matthew Gray of the Massachusetts Institute of Technology found at `http://www.mit.edu/people/mkgray/net/web-growth-summary.html`, there are approximately 650 000 hosts serving Web information in January 1997, and since 1993, the size of the Web doubles each time in less than 6 months. The Netcraft Web server survey found at `http://www.netcraft.co.uk/Survey/` reported 1 920 933 hostnames for active Web information servers in February 1998. There are 200 million Web documents in September 1997 according to `http://www.computerworld.com/emmerce/depts/stats/sites.html`.

cation protocols, data formats, Web software (e.g., Web browsers), and security issues.[2] In recent years, there has been a growing interest in adding computational content to the Web. Such content implements sophisticated interactive behaviours on the Web providing more interesting and expressive documents, allowing useful tasks to be performed over the Web, and increasing the Web's usability.

The earliest mechanism for adding computational content is the form submission mechanism. By submitting information via a form on a document displayed on a Web browser, the user can invoke programs on remote hosts, thereby allowing arbitrary applications to be used from the Web. For example, numerous databases can be queried from the Web[3] including databases of Web pages (e.g., Lycos[4]) which help users locate desired information. The form submission mechanism has several drawbacks. Since only programming on the remote host is involved, it is not possible to locally extend the behaviour of hypertext link activation or to augment the forms interface with additional graphical user interface elements. Also, communication latency can be a problem, especially for highly interactive applications which demand extensive communication between the local host (where the form is) and remote hosts (where the computations occur). A further drawback is the load on the remote host caused by multiple Web browsers invoking computations.

Mobile code [56, 202] adds sophisticated behaviours to the Web in a different way, as popularised in 1995 by Java applets, which are programs written in the language Java [187], and JavaScript programs [155]. Both Java and JavaScript have C-based syntax. A definition of mobile code is code that can be transmitted across the network and executed on the other end [56]. This thesis focuses on mobile code which is downloaded from the remote host to the local host. In con-

[2] More information on these developments is found at `http://www.w3.org/`.

[3] A collection of such databases is seen at `http://www.mtm.kuleuven.ac.be/Services/search.html`.

[4] Lycos Web site at `http://www.lycos.com/`.

trast to the form submission mechanism, mobile code is downloaded with Web documents and executed locally within the Web browser. Sophisticated user interfaces can be programmed with mobile code allowing complex interactive behaviour in documents. Mobile code uses resources on the local host and will not be affected by unpredictable network characteristics unless they access the network. Mobile code also has advantages for software distribution: mobile code does not need to be explicitly installed and to a large extent is architecturally independent (e.g., Java applets).

The form submission mechanism and mobile code add programmable behaviours to the Web making it a more powerful interactive system. These behaviours though contained in the Web need not interact with the Internet. For example, a Java applet can implement a tic-tac-toe game whose computations do not need to access the Internet. Programs which interact with the Internet or the Web can achieve computational capabilities beyond that of stand-alone systems, downloading program components to extend their own functionality, utilising programs running on remote hosts, or retrieving and processing Web information on behalf of its users. Notable examples of these programs include information gathering tools which attempt to automatically extract specific items of Web information [97, 164, 213]. These tools tackle a hard problem since Web information is mostly in natural language form, and in general, not sufficiently structured. More recently, the Web is being used to disseminate information which is more amenable to sophisticated querying and automated extraction [30, 134, 110, 74]. Such work tends towards the vision of a machine processable Internet-scale knowledge repository [23].

Both aspects of *Web programming* mentioned above, adding programmable behaviours to the Web and constructing programs which interact with the Web, has largely been dominated by imperative programming[5]. Most mobile code

[5] As described in [206], imperative programming is characterised by a state (e.g., variables) and commands (e.g., assignments) which modify the state, imitating the Von Neumann machine

languages such as Java, Tcl [163], Visual Basic in ActiveX [80], and Python [200] are based on the imperative programming paradigm (though some of these languages also utilise object-oriented concepts). An exception is Caml [173], a functional language with imperative constructs. Imperative languages such as Perl [212], C, and Java are widely used for developing Web software such as Web browsers and information gathering tools. There are Java, Perl, and C libraries[6] which provide low-level support such as communication protocol implementations and Web document parsing tools. There are also Java libraries providing application-level support for distributed persistent data exchange, interfacing to relational database systems, and electronic commerce on the Web.[7]

The focus of much programming language research has been on raising the level of abstraction. A class of programming languages deemed to be at a higher level of abstraction than imperative languages is *declarative programming languages*. Declarative languages enable the programmer to more easily model a problem or subject domain without focusing on the computer's Von Neumann architecture, and hence, increase programmer efficiency and the readability of programs. Declarative languages are characterised by simple formal semantics, and so, are easier to reason with, and to develop tools for (e.g., interpreters). For Web programming, emerging are languages specialised for querying the Web (i.e., declaratively expressing how to navigate portions of the Web to find desired information) [10, 119, 92, 144, 115], which explore various abstractions (e.g., objects and database relations) for modelling the Web in order to simplify retrieving and processing of Web data.

---

with its modifiable registers and store (or memory).

[6]Libwww is a package containing an implementation in C of HTTP and other Internet protocols, and a rudimentary HTML parser. It is available at `http://www.w3.org/Library/`. The libwww-perl distribution is a collection of Perl modules providing an application programming interface to the Web and is available at `http://www.linpro.no/lwp/`.

[7]`http://java.sun.com/products/index.html` has a list of Java libraries for application-level support.

A category of declarative languages which has had the benefit of more than 20 years of research is *logic programming languages* which view computation as deduction from a set of axioms or a theory. Logic programming with program structuring abstractions [47, 64] (e.g., theories, modules, or objects) has shown its utility in a variety of applications including expert systems [146], symbolic (e.g., natural language) processing [59], Artificial Intelligence (AI) problem solving and knowledge representation [180], deductive databases [116], and agent-based systems [117, 204, 195].[8] For software engineering, logic programming has a number of advantages including logic programs as executable specifications, meta-programming for building abstractions, and rapid prototyping [53, 72, 186]. The program structuring abstractions support logic programming in-the-large.

Implementations of logic programming, the most popular example of which is Prolog [185] based on a subset of first-order logic, have features not found in traditional imperative languages including ease of meta-programming[9], pattern matching via unification, backtracking search, structured database representation and querying, and built-in grammars for parsing. They have also the favourable features found in Java such as robustness, automatic memory management[10], and modularity.

Compared to imperative languages, there has been little exploration of declarative languages for Web programming. Only in the last few years (as this thesis was being developed) has work begun on the relationship between logic programming and the Web [196, 66, 63, 78] (see Chapter 9 for an overview). It is the general theme of this thesis to investigate the use of logic programming technology for the Web.

[8] Examples of more than 500 applications of Prolog and related languages are found in the Prolog 1000 list at `ftp://src.doc.ic.ac.uk/packages/prolog-progs-db/prolog1000.v1.gz`.

[9] Notably, an interpreter for the language could be easily written in the language itself.

[10] Prolog code is pointer-free and there is automatic garbage collection with dynamic memory allocation and deallocation.

1.1 Integrating Logic Programming Technology with the Web

This thesis presents a model for adding logic programming based interactive behaviours and structured information to the Web, and for logic programming based retrieval and processing of Web content. The model is characterised by two key ideas:

1. Logic programming code is built into Web documents. Given the versatility of logic programming for different styles of programming such as algorithmic, AI, and database programming, such code can be used in two ways. First, this code can be used to implement rule-based interactive behaviours on the Web in the mobile code style. Second, this code can represent knowledge-bases or structured information in a similar form as deductive databases. Deductive databases extend relational databases, utilising logic programming rules for more complex data modelling [116]. A deductive database is, in essence, a logic program: base relations map to facts, and rules are used to define new relations in terms of base relations, and to process queries. Also, deductive databases structure information according to predefined conceptual schema. Such structured information is sufficiently formal, and therefore, more amenable to sophisticated automated searching, extraction, and processing, but is also high-level (or abstract) enough to remain readable. Logic programming provides a uniform means for representing both the data and the queries over such data. Also, knowledge-based query processing is facilitated by logic programming.

2. Web documents inter-connected by links are modelled as inter-related logic programs. This idea is motivated by four observations. First, the Web when viewed as a collection of Web documents connected by links consists of distinct but connected components. Hence, distinct program compo-

nents, and the relationships between them, naturally model the structure of information on the Web. Second, the Internet is a distributed system. An application may consist of program components situated in separate files which, in turn, may be distributed over several Internet hosts. Program structuring abstractions provide a framework for integrating these components in a principled way. Computational content on separate Web documents can be integrated based on such abstractions. Third, well-established meta-programming techniques in logic programming provide a paradigm for manipulating the Web, especially if Web documents contain logic programming code. These documents can be manipulated using their identifiers as first class entities in programs. Fourth, as mentioned, program structuring abstractions have been useful for AI applications and software engineering, and therefore, should be explored for the development of Web applications.

The model emphasises computations on the local host due to its favourable characteristics as seen from our earlier comparison of mobile code to the form submission mechanism.

1.2 Contributions

This thesis makes the following contributions to the relationship between logic programming and the Web:

1. A new model called *LogicWeb* is introduced for integrating logic programming technology with the Web, where the Web is viewed as a collection of logic programs.

   LogicWeb adds logic programming based interactive behaviours to Web documents. A Web document is viewed as a *live information entity* able to respond to user queries using its own code and can have the behaviours of its

links determined by rule-based reasoning. The behaviours can reason with Web pages by collecting and manipulating data from them. LogicWeb also adds to the Web structured information in the form of deductive databases and knowledge-bases.

2. A new language (extending Prolog) based on LogicWeb is developed for coding the logic programming behaviours, and representing and querying the structured information. This language offers a new Web programming paradigm, where computing with the Web is equated to evaluating goals in compositions of programs. The practical result is that programming with the Web in this language is aided by familiar modularity abstractions, and the programmer need not explicitly deal with low-level issues such as document retrieval, caching, and parsing.

3. An operational semantics is given which makes precise how programs in the LogicWeb language interact with the Web during goal evaluation, and how these interactions affect goal evaluation. Interacting with the Web during program execution corresponds to consulting an oracle function during proof searching. The semantics forms the basis for a language implementation.

4. A system which realises LogicWeb is implemented as an extension to a Web browser. Security is an important issue in the LogicWeb system because code is downloaded from remote hosts and executed locally. A security model for the system is developed which is flexible and precisely specified using proof rules.

5. A range of examples illustrates LogicWeb-based programmable behaviours, and demonstrates the feasibility and advantages of the LogicWeb language for coding Web search tools, Web-situated databases called *lightweight deductive databases*, and extensions to the semantics of Web links.

1.3 Overview of Thesis

The rest of this thesis is organised as follows. In Chapter 2, the background of the thesis is given. Chapter 3 presents the LogicWeb model, and the LogicWeb language describing its operational semantics. The architecture of the LogicWeb system and its implementation is described in Chapter 4. Chapters 5, 6, and 7 discuss LogicWeb applications constructed using the implementation given in Chapter 4. Chapter 5 describes in detail a tool called CiFi for finding citations of specific Computer Science publications on the Web. Chapter 6 introduces light-weight deductive databases for structuring Web information. The use of such databases is demonstrated for organising citation information on the Web and for generating guided tours of Web documents. These guided tours are usable from the Web. Chapter 7 explores a LogicWeb-based conceptualisation of the Web as a two-layered hypertext model where Web links can have more sophisticated semantics. A number of different link semantics is demonstrated addressing weaknesses in the current Web linking mechanism, and allowing linking based on semantic criteria. Chapter 8 describes extensions to the LogicWeb system for tackling security issues. It is shown formally that the security model for the system can prevent attacks from downloaded programs, and that crucial to the security model's development are the use of meta-interpreters[11] and the extensible operational semantics of the LogicWeb language. LogicWeb is compared with related work in Chapter 9, and Chapter 10 concludes and briefly presents avenues for future work. Appendices A and B provide an additional level of detail on the extraction of components from Web documents in the LogicWeb implementation and predicates for supporting LogicWeb applications, respectively.

[11] A meta-interpreter is an interpreter for a language written in the language itself [185].

Chapter 2

Background

2.1 Technical Overview of the World Wide Web

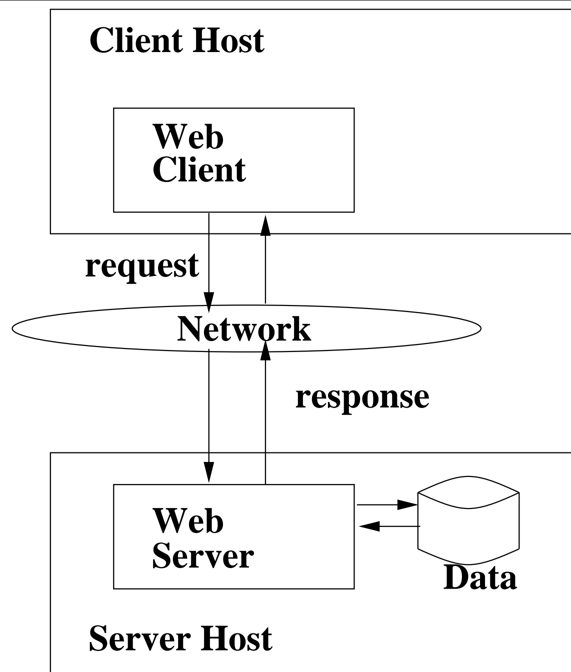
This section presents technical details concerning the Web's client-server architecture and hypertext nature. Only details relevant to the thesis are discussed. Whenever appropriate, the reader will be referred to other sources for full descriptions of the technologies.

2.1.1 Client-server Architecture

The Web, like the Internet, is based on a client-server architecture. A *server* is a program which accepts requests via the network, performs a task according to the request, and then responds to the requesters. A *client* is a program that can send a request to a server, wait for a response, and then reply to the server. The client also accepts inputs (i.e., requests) from the user and displays data (i.e., responses from the server) to the user. Servers control access to information on the Internet, accessing data on behalf of the client, manipulating the data as required, and sending the data to the client.

Servers and clients can be characterised by the protocol they use. Web servers and clients communicate over the network using the Hypertext Transfer Proto-

Figure 2.1 The client-server architecture of the Web.



col. However, usually Web browsers can also communicate with servers implementing other Internet communication protocols such as FTP, NNTP, Gopher, or WAIS. In the rest of the thesis, the term *Web browser* shall refer to a Web client which supports link traversal and fill-in forms. Popular Web browsers include Microsoft Internet Explorer, Netscape Navigator, and NCSA Mosaic.

The terms *server host* and *client host* refer to the machines where the server program and the client program run respectively. A server and a client may run on the same machine, i.e. a computer can host a server and a client at the same time. Figure 2.1 illustrates the client-server architecture.

2.1.2 The Web as a Hypertext System

The explosive growth of the Web is due largely to the use of hypertext. Using hypertext, two independently developed collections of information, if found to

be related, can be easily linked. This means that newly created information can be easily linked to the existing corpus. Standardisation of an addressing scheme and Web document formats further ease the linking process.

Following [24], the Web is taken to mean the body of data available on the Internet using all or some of the following items:

- an address system allowing a reference (called *Uniform Resource Identifier*) to be associated with each item in the Web;
- a network protocol (called *Hypertext Transfer Protocol*) for retrieving information on the Web; and
- a standard mark-up language (called *Hypertext Mark-up Language*) in which documents are written.

These items implement the Web's networked hypertext functionality.

2.1.2.1 Uniform Resource Locators (URLs)

A *Uniform Resource Locator* is a string referring to a resource on the Internet which encodes explicit instructions on how to access the resource including the network protocol to use and the resource's physical location. A URL for a resource consists of the following components:

1. the Internet protocol for accessing the resource;
2. the Internet host's (i.e., the server host's) name on which the resource is stored;
3. (optionally) a port number at which the server is receiving protocol requests (the default number is normally 80); and
4. the path identifying the resource on the server host's file system. A server is configured to map this path to the actual path on the local file system.

For example, the following is the URL of a document about Web addressing:

```
http://www.w3.org/Addressing/Addressing.html
```

The URL states that the document of path `/Addressing/Addressing.html` is retrieved using the HTTP protocol (discussed in Section 2.1.2.3) from the server running on a host with address `www.w3.org` using the default port number.

Other forms of the URL include:

- *relative URLs*. A relative URL within a Web document is expanded by combining with a portion of the containing document's URL. For instance, the relative URL

  ```
  /Addressing/Addressing.html
  ```

  within a document of URL:

  ```
  http://www.w3.org/
  ```

  expands to the *absolute* or full URL:

  ```
  http://www.w3.org/Addressing/Addressing.html
  ```

  Just as UNIX filenames are expanded by the current directory path, relative URLs are expanded by its containing document's URL.

- *URLs encoding queries*. A URL can encode a query to be sent to a server-side program. In such a URL, a "?" separates the address of the program from the query posed to it. For instance, the following URL:

  ```
  http://www.altavista.yellowpages.com.au/cgi-bin/query?
  mss=simple&pg=q&q=logicweb
  ```

  sends to a Web document index search program called AltaVista[1] the query consisting of the attribute and value pairs `(mss,simple)`, `(pg,q)`, and `(q,logicweb)`.

[1] See `http://www.altavista.yellowpages.com.au/`.

URLs are part of Uniform Resource Identifiers (URIs) [21], the set of all names or addresses that are strings referring to resources. There is another kind of URI which functions as a persistent name for a resource which is called a *Uniform Resource Name* (URN). URNs were proposed as a resource naming scheme to avoid problems with using URLs for linking. When a URL is used to identify or link to a resource, the resource's identity is associated with its physical location. This means that it is difficult to move the resource around without having to update all documents that linked to it. Also, if the resource is replicated onto multiple machines to avoid long network delays or overloaded servers, links made using URLs can only point to a specific copy of the resource. URNs identify resources independently of their physical locations. URNs are still being developed.[2] URNs will only be considered in Chapter 7.

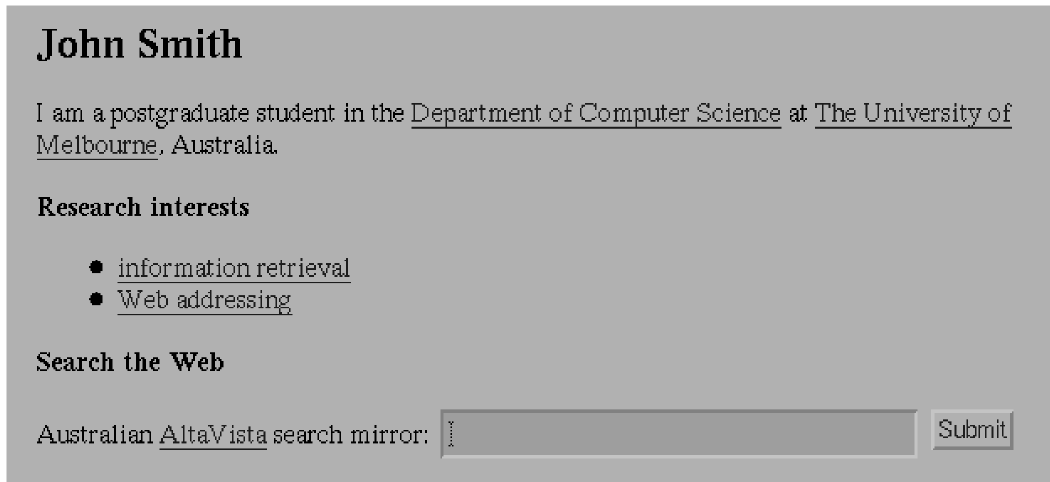
2.1.2.2 Hypertext Mark-up Language (HTML)

HTML is the standard language in which Web documents are written. It is a mark-up language consisting of *tags* for demarcating document components. The tags allow structuring of the contents of a document into paragraphs, levels of headings, bullet lists, and user interface objects such as menus, text input fields, and buttons. Tags also permit mobile code to be attached. Mobile code is either linked to from the document but rendered as part of the document (e.g., Java applets), or is textually part of the document (e.g., JavaScript). The term *Web page*, or *page*, is used to mean a HTML document with a URL referring to it.

A page consists of a "head" and a "body" denoted by the tags "`<HEAD>...</HEAD>`" and "`<BODY>...</BODY>`" respectively. For example, Figure 2.2 shows a rendered page whose HTML source is shown in Figure 2.3.

[2]See `http://www.acl.lanl.gov/URN/`.

Figure 2.2 John Smith's homepage when rendered on a Web browser.



In Figure 2.3, the "<UL>...</UL>" tags indicate a bullet list. A "<A HREF=*(URL)*>*(Label)*</A>" component is an anchor (i.e., an origin of a hypertext link), consisting of the URL of a link destination defined by the `HREF` attribute and a label. The "<FORM...</FORM>" tags define a form into which a user types a query. In the form tag, `METHOD=GET` states that the query will be sent using the GET method of the Hypertext Transfer Protocol. The GET method is discussed further in Section 2.1.2.3. The `ACTION` value indicates AltaVista's URL. The "<INPUT...>" tags define what is called the *query attributes* of the form which in Figure 2.3 are `mss`, `pg`, and `q`. User inputs to the form are assigned to the query attributes. For instance, if the user types in "logicweb" in the form, `q` is assigned the value "logicweb". No user inputs are associated with query attributes of `TYPE` set to `hidden` since such attributes are not presented on the Web browser (i.e., hidden from the user). The `VALUE` attribute in each "<INPUT...>" tag defines a default value for the query attribute (e.g., `simple` is a default value for `mss`). How the inputs to the form are used to generate a query to AltaVista is discussed in Section 2.1.2.3.

Figure 2.3 The HTML mark-up of John Smith's homepage.

```
<HTML>
<HEAD><TITLE> John Smith </TITLE></HEAD>

<BODY>
<H1>John Smith</H1>

I am a postgraduate student in the
<A HREF="http://www.cs.mu.oz.au/">Department of Computer Science</A>
at
<A HREF="http://www.unimelb.edu.au/">The University of Melbourne</A>,
Australia.

<H3> Research interests </H3>

<UL>
<LI> <A HREF="ir-if/">information retrieval</A></LI>
<LI> <A HREF="http://www.w3.org/Addressing/Addressing.html">
     Web addressing</A></LI>
</UL>

<H3> Search the Web </H3>
<CENTER>
<FORM
METHOD=GET
ACTION="http://www.altavista.yellowpages.com.au/cgi-bin/query">
Australian <A HREF="http://www.altavista.yellowpages.com.au/">
AltaVista</A>
search mirror:
<INPUT TYPE=hidden NAME=mss VALUE=simple>
<INPUT TYPE=hidden NAME=pg VALUE=q>
<INPUT NAME=q size=30 maxlength=200 VALUE="">
<INPUT TYPE=submit VALUE=Submit>
</FORM>
</CENTER>
</BODY>
</HTML>
```

HTML is ongoing work and is continually expanding with additional tags. For instance, the recent recommendation for HTML is HTML 4.0 [171] which adds to an older standard (HTML 2.0) features such as tables, applets, text flow

around images, superscripts and subscripts, and support for scripting languages. In the rest of this thesis, HTML tags are introduced as required.

2.1.2.3 Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) [25] is a low-overhead protocol based on the URL addressing scheme which can transfer information quickly to support hypertext browsing. Various forms of information can be transferred including text, hypertext, images, and other media types defined for the Multipurpose Internet Mail Extension (MIME) [25].

HTTP is stateless in that the Internet connection is held only for the duration of one operation (e.g., a request for a file and its response) and that HTTP returns results based only on the current request and not on previous ones. Since the location of the resource is encoded in the URL, HTTP does not need to support full navigation features like the FTP and Gopher protocols, and so does not keep navigation history. Thus, HTTP is a simple request and respond protocol. Once the client establishes a connection with the server, it sends a request to the server. The server responds with the requested information (if available). The information received by the client in response to HTTP requests are called *HTTP response objects*.

All client requests consist of two parts. The first part consists of

- a request method which specifies the operation to perform;
- the URL of the resource (e.g., a file) on which to perform the operation; and
- the protocol version (e.g., HTTP 1.0).

The second part consists of (optional) headers providing information to the server such as what types of data the client can handle, the client's identity, the URL of the document from which the request originated (the referrer), and body content.

An early draft of the HTTP protocol specification contains a list of 13 HTTP request methods.[3] Currently, the three most commonly implemented methods are GET which is used to retrieve a resource, HEAD which requests header (or meta-) information about a resource, and POST which allows the client to send information of arbitrary size to the server [25]. The other methods (particularly those which update information on the server) are at this time rarely supported by Web servers, and hence, only the three common methods are considered for LogicWeb. As more methods are defined and become widely used in the future, they can be considered for LogicWeb.

The server's response consists of two parts:

1. a status line containing the server's protocol version, a code indicating the status of the request, and descriptive information about the status; and

2. a message containing resource meta-information, and possibly body content. Typically, the meta-information includes the date the resource was requested, the name and version of the server containing the resource, MIME version, the time the resource was last modified, the content type (e.g., `text/html`), and the content length. If the body content is of type `text/html`, then it is a HTML document.

A GET Request Example. The ordinary behaviour of links as seen in Web browsers (e.g., NCSA Mosaic) is to retrieve and display a page. When a user clicks on a hypertext link on a Web browser, the browser translates the URL specified in the link into a GET request using the server host and path information encoded in the URL, and then sends that request to the server. For example, if the user clicks on the link on John Smith's page marked-up by

```
<A HREF="http://www.w3.org/Addressing/Addressing.html">
Web addressing</A>
```

[3] See `http://www.w3.org/Protocols/HTTP/HTTP2.html`.

the following GET request is sent after connecting to the host `www.w3.org`:

```
GET /Addressing/Addressing.html HTTP/1.0
       /* a blank line here */
```

The server responses with:

```
HTTP/1.0 200 Document follows
Server: CERN/3.0A
Date: Thu, 31 Jul 1997 00:59:05 GMT
Content-Type: text/html
Content-Length: 12998
Last-Modified: Wed, 04 Jun 1997 20:17:41 GMT
       /* a blank line here */
<HTML>
       /* the rest of the requested document is here */
```

The status code of 200 indicates a successful request. On failure, other status codes (described in [25]) are returned indicating the reason for failure such as forbidden access to the requested resource, the server is busy or encountered some error, or the resource requested can not be found. If a resource has been moved, the server (if configured to do so) sends to the client the URL of its new location using the field:

```
Location: /* new URL*/
```

A client can be built which understands this location field and automatically utilises the new URL. Web browsers normally handle such redirections on behalf of the user. It is possible that querying the server with the new URL results in another URL being returned, i.e. several redirections can occur before the resource is finally obtained. The final URL used to retrieve the resource is the resource's *actual URL*.

A HEAD Request Example. A HEAD request such as the following:

```
HEAD /Addressing/Addressing.html HTTP/1.0
       /* a blank line here */
```

retrieves only the meta-information from the server, i.e. everything up to and including the last modified time in the above example.

Query Submission Examples. A GET request encoding a query to a server-side program is submitted in the same way. For instance, after connecting to the host `www.altavista.yellowpages.com.au`, the following request is issued:

```
GET /cgi-bin/query?mss=simple&pg=q&q=logicweb HTTP/1.0
```

Such a GET request is generated when a user follows a link with a URL encoding the query (e.g., the example shown in Section 2.1.2.1), or when a user fills in a form on a Web browser (e.g., the form shown in Figure 2.3) and submits it to the server-side program of path `/cgi-bin/query`.

In the GET method, query information is submitted as part of the URL. This limits the amount of information that can be sent since the length of a URL is limited (to 255 characters normally). The POST method allows the user to submit information of arbitrary length in a request. For instance, a POST request which sends 10 bytes of data takes the following form:

```
POST /cgi-bin/post-query HTTP/1.0
Content-type: application/x-www-form-urlencoded
Content-length: 10
       /* a blank line here */
int=hello!
```

The above content type is reserved for submitted form inputs. The content length indicates the size of the body to be submitted with the request. `/cgi-bin/post-query` is the path of the program processing the form inputs. Forms on Web browsers can specify whether to submit form inputs using a GET or POST request using the `METHOD` attribute, for example, as seen in Figure 2.3. This form submission mechanism is called Common Gateway Interface (CGI) and the server-side program often called a *CGI script*.

A HTTP request issued by a Web client to a Web server *succeeds* if the client receives the requested information. HTTP does not guarantee the success of HTTP requests. HTTP requests may *fail* which means that the client does not receive

the requested information, and instead receives an error message giving the reason for the failure. If the server is down, then no reply will be received from the server, and the request must be made to time-out.

From the client's perspective, the Web has a nondeterministic nature in that the client generally cannot predict the result of a HTTP request. The performance and reliability of the Web (e.g., as experienced by users of Web browsers) are affected by a number of factors which are often unpredictable and out of the user's control. Such factors include network traffic and bandwidth, availability of servers, changes to information accessible by servers, and server host characteristics (e.g., CPU speed and memory). For instance, the client will not receive a requested file if it has been deleted, or the server is busy. A request which failed at one point in time may succeed at another.

At any point in time, i.e. assuming a "snapshot" of the Web, a URL uniquely refers to at most one page. However, the same URL used at different times can retrieve different information. The dynamic nature of Web information makes it generally impossible for the client to predict what information a successful HTTP request will return. An example of this is a URL encoding a query. The result of the query can vary with the time the request is issued. Another example is when the page on the server host is modified between two HTTP requests.

The Web's nondeterministic nature as seen by the client is central in discussing the semantics of the LogicWeb language in the next chapter.

The Hypertext Transfer Protocol (HTTP) has been in use on the Web since 1990, and is continually developing. HTTP 1.1 is the most recent proposed standard and is designed to improve several aspects of HTTP 1.0 including the performance of Web clients and servers, and the reduction of Internet traffic due to HTTP data transfers. In order for these improvements to take effect, servers and not just clients must implement HTTP 1.1. Implementation of servers using HTTP 1.1 are underway but this thesis considers only HTTP 1.0 since it is currently supported by most servers.

2.2 Logic Programming

This section briefly presents the syntax and terminology of logic programming, introduces meta-programming in logic, and reviews work on compositional logic programming relevant to LogicWeb. For more thorough introductions to logic programming and Prolog, the reader is referred to the *Art of Prolog* [185] and the *Foundations of Logic Programming* [124].

2.2.1 Syntax and Terminology

A *logic program* is a finite set of *Horn clauses* or *rules* of the form:

$$\mathcal{A} \texttt{ :- } \mathcal{B}_1, \ldots, \mathcal{B}_n$$

where $n \geq 0$. Conjunction is denoted by the symbol ",". If a clause is read from left to right, "`:-`" is often read as "if", but if read from right to left, "`:-`" can be read as "implies". The goal $\mathcal{A}$ is the *head* of the clause and the conjunction $\mathcal{B}_1, \ldots, \mathcal{B}_n$ is the *body* of the clause. Each $\mathcal{B}_i$ is called a *subgoal*. $\mathcal{A}$ and each $\mathcal{B}_i$ are atomic formulae where an atomic formula is as defined in the standard way in [124]. When n is 0, the clauses are called *facts*. Facts can be written in the form A `:- true`, where the constant `true` represents the empty goal. All the variables occurring in a clause are implicitly universally quantified. For example, the clause

```
h(X,Y) :- q(X,Y)
```

stands for

```
∀ X,Y [h(X,Y) :- q(X,Y)]
```

A *predicate* is a collection of clauses each of whose head has the same *functor* and *arity* (e.g., `h(X,Y)` has functor `h` and arity 2). *Constants* are function symbols of arity 0.

A *pure Prolog* program, as defined in [185], is a logic program where an order is defined for clauses in the program and for goals in the body of the clause. The computation model of Prolog exploits this ordering.

A goal G_1 *subsumes* another goal G_2 if there exists a substitution θ such that $G_2 = G_1\theta$.

Strings within double quotes are equivalent to Prolog lists of ASCII codes.

2.2.2 Meta-programming in Logic Programming

Meta-programming involves programming with data which are themselves programs [20, 172]. Programs used as data are said to be at the *object*-level, and the programs manipulating programs are at the *meta*-level. Meta-programming has been used extensively in logic programming such as for debuggers, compilers, and program transformers [20]. Many applications intrinsically involve meta-programming such as those which formalise proof procedures (e.g., constructing meta-interpreters) and knowledge assimilators. Moreover, meta-programming techniques have been shown to be powerful enough to capture many devices for knowledge representation in AI [33, 34].

A simple application of meta-programming in logic is the *vanilla meta-interpreter* which captures the computation model of logic programs [185]:

```
solve(true).
solve((A, B)) :- solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

The above interpreter states how to prove goals with respect to an object-level program represented using the predicate `clause/2`, where the first argument is a clause's head and the second, the clause's body. Logic programs have a declarative and procedural reading.

Declaratively, the `solve/1` fact states that the empty goal is always true. The first `solve/1` rule states that the conjunction `(A,B)` is true if both `A` and `B` are

true. The second rule states that a goal `A` is true if there is a clause `A :- B` in the object program such that `B` is true.

The program can also be given a procedural reading. The `solve/1` fact states that the empty goal is solved. The first rule states that to solve a conjunction `(A,B)`, solve `A` and then `B`. This reflects the Prolog computation model where subgoals in the body are evaluated left to right. The second rule states that to solve a goal, obtain a clause from the object program whose head unifies with the goal, and then solve the body of the clause. In Prolog, the required clause is obtained by a sequential search over the clauses of the program.

Various extensions and modifications to the vanilla meta-interpreter have been studied. For instance, the vanilla meta-interpreter can be modified to compute information for detecting infinite loops (which can happen due to the use of recursion [29]) and proof tracing (e.g., printing out information indicating the progress of a proof), or to implement reasoning mechanisms such as forward and backward chaining [180]. Meta-interpreters can also be combined. Enhancements and combinations of meta-interpreters have been used for building expert systems [184]. Some of these techniques are used in the implementation of LogicWeb and are discussed further in Chapters 4 and 8.

`solve/1` can be generalised with an argument specifying the program in which goal evaluation is to be carried out. This generalisation is done with the `demo/2` predicate first proposed by Bowen and Kowalski in [34]. A key motivation of their work was to provide a language where the programmer can explicitly refer to different logic programs and discuss provability with respect to alternative programs.

The vanilla meta-interpreter with `demo/2` takes the form:

```
demo(P, true).
demo(P, (A, B)) :- demo(P, A), demo(P, B).
demo(P, A) :- P::(A :- B), demo(P, B).
```

The goal `demo(P,G)` states that the program named `P` can be used to demonstrate the conclusion `G`. The clauses of the program `P` are associated with the pro-

gram's name using the operator "`::`" and are stored in facts of the form `P::(A :- B)`. The program name argument allows the program in which goals are evaluated to be changed dynamically in a proof. For instance, by prefixing a goal `G` with the name of the program `P` using "`:`", a new kind of goal is introduced, and is written as `P:G`. The following rule is added to `demo/2` to process such goals:

```
demo(P, Q:G) :- demo(Q, G).
```

The definition of `demo/2` can be extended to model operators for compositional logic programming as will be shown later.

2.2.3 Compositional Logic Programming

Early implementations of logic programming languages did not adequately support software engineering and AI applications. A *structuring mechanism* [118], that is a construct, mechanism, or framework for organising and composing the parts (subprograms) of a program, is needed which would allow an application to be constructed by combining together separate components. The procedure as a structuring mechanism in logic programming is too fine-grained in that logic programs appear flat and unstructured. As a result, numerous logic programming languages with structuring mechanisms have been developed over the past decade. From the software engineering perspective, structuring mechanisms provide modularity and various abstractions for building software systems and encouraging code reuse. Such structuring mechanisms have also been proven useful in AI applications to represent, structure, and manipulate knowledge-bases.

Approaches for structuring logic programs have been surveyed extensively by Kwok [118], Bugliesi *et al* [47], and Davison [64]. These approaches include incorporating into logic programming the notions of modules and blocks as usually found in traditional imperative languages and object-oriented features such as inheritance as seen in [140, 214, 46, 37, 1], combining programs via meta-level statements as seen in systems like MultiLog [109], Epsilon [58], and MetaPro-

log [13] which have been used for manipulating knowledge-bases, and language extensions using modal logic [93, 199]. Many Prolog implementations such as SICStus [106], BinProlog [191], SWI-Prolog [209], and ESP [52] provide a practical module system with a predicate import and export mechanism to support information hiding and encapsulation. These module systems and many object-oriented extensions to logic programming have been developed from a pragmatic point of view and have not been given a formal semantics.

As noted in [47], two main lines of research into structuring logic programs with a solid theoretical foundation include *algebraic composition operators* [160, 138, 35] and *implication goals*, an idea first introduced by Miller [148, 149].

Algebraic composition operators treat programs as elements in an algebra, and operators for composing programs as operators over that algebra. This approach has a number of benefits including:

- facilitating the reuse of the same program within different compositions of programs;
- extensibility, where new composition operators can be added by introducing operators into the algebra, or defined using existing operators; and
- a formal theoretical basis for the composition operators and for building new operators from existing ones.

In contrast to the practical module system in Prolog implementations, the algebraic approach provides, in a single framework, a number of different ways for combining programs, each with its own precise meaning and formal semantics. The algebraic approach takes a *meta-linguistic* view where the logic programs are sets of Horn clauses, and the meaning of the composition operators are specified in terms of operations on the component clauses.

One particular algebraic program composition framework which was proposed by Brogi in his doctoral thesis [35] stands out because of its elegant semantics and expressive power. The framework builds on familiar techniques for

logic programming semantics, and integrates functionalities for program specification, databases, knowledge representation, and problem solving. The framework consists of a set of basic operators for composing separate logic programs. The operators are basic in the sense of their simple semantics and their extensive modelling capabilities. The generality and wide applicability of the operators are shown in their use for modelling hypothetical and abductive reasoning, and for modelling extensions to logic programming such as blocks, inheritance in object-oriented programming, and contextual logic programming [151]. Moreover, the framework provides a formal approach for modularising logic programs, where predicate import and export mechanisms and information hiding notions are modelled.

The algebraic approach on its own has a drawback. The program system formed by an algebraic composition of programs is used for evaluating the top-level goal. All subgoals are evaluated in the same program system. There is no way to dynamically prove subgoals in different configurations.

A more flexible method of composition uses the idea of implication goals. Miller's implication goal [149] is of the form $D \supset G$ (D a program) and has the following operational meaning: $D \supset G$ is provable from a program P if the goal G is provable from $P \cup D$. Implication goals provide an object-level connection to meta-level algebraic compositions. Additional clauses can be dynamically imported and combined with existing ones as part of goal evaluation. This approach contrasts with the algebraic composition approach in that implication goals occur as subgoals in the body of clauses extending the Horn clause (i.e., a *linguistic* approach) whereas the algebraic approach leaves the Horn clause as it is. Variants along this line of work which extend the Horn clause with different forms of subgoals include logic programming language extensions such as contextual logic programming [151], N-Prolog hypothetical implications [88], linear implication in BinProlog [191], Structured Prolog [94], the framework developed by Brogi *et al* in [36], and use of modal operators in subgoals of the form $[M]G$,

where M is a program, $[M]$ is a modal operator, and G is a goal [16].

Shown in [40] is a logic programming language which combines the use of *program expressions* (constructed using Brogi's algebraic composition operators) with subgoals written in the style of implication goals. Such subgoals allow goals in the body of a rule to be evaluated with respect to different program expressions. Below, four operators, the syntax, an operational semantics, and a meta-interpreter for this language are presented. In the next chapter, this language is extended to form the LogicWeb language.

2.2.3.1 Meta-level Program Composition Operators

Four meta-level operators in Brogi's algebraic program composition framework are: *union* ("$\cup$"), *intersection* ("$\cap$"), *restriction* ("$\prec$"), and *encapsulation* ("$\circledcirc$") [35].

The union of two programs is the set-theoretic union of their clauses. For example, consider two programs with identifiers `M` and `N`, where `M` consists of the clauses:

```
p(X) :- q(X).
r :- b.
```

and `N`, the clauses:

```
r :- w.
q(a) :- true.
```

The union of `M` and `N`, written as `M` $\cup$ `N`, is the program:

```
p(X) :- q(X).
r :- b.
r :- w.
q(a) :- true.
```

The intersection of two programs is the program consisting of clauses which result from combining clauses from both programs whose heads unify. More

precisely, if the function $mgu(H_1, H_2)$ returns the most general unifier of atoms H_1 and H_2, the intersection of P and Q is the set

$$\{A \texttt{:-} G \mid (H_1 \texttt{:-} G_1 \in P) \ and \ (H_2 \texttt{:-} G_2 \in Q) \\ and \ \theta = mgu(H_1, H_2) \ and \ A = H_1\theta \ and \ G = (G_1, G_2)\theta\}$$

The intersection of programs `M` and `N`, written as `M` $\cap$ `N`, corresponds to the following program with only one clause:

```
r :- b, w.
```

The restriction of two programs, written as $P \prec Q$, is the program consisting of clauses of P which are not defined in Q. Formally, if $pred(A)$ is the predicate symbol (i.e., functor/arity) of an atom A, and $preds(P)$ is the set of predicate symbols defined by program P, the restriction of P by Q is the set

$$\{A \texttt{:-} G \mid (A \texttt{:-} G \in P) \ and \ pred(A) \notin preds(Q)\}$$

The restriction of `M` by `N` is the program:

```
p(X) :- q(X).
```

since `r/0` is defined in `N`.

The encapsulation of a program P, written as $\circledcirc P$, consists of clauses formed from atoms provable (using the rules given in Section 2.2.3.3) from P alone:

$$\{A \texttt{ :- true} \mid A \ is \ an \ atom \ and \ A \ is \ provable \ from \ P\}$$

In theory, this set may not be finitely computable. But in practice, not all the elements of this set need to be computed. For instance, given a goal A and a program P, an attempt is made to prove A from P, and if A is proved from P, then a clause A `:- true` is obtained from $\circledcirc P$. Proving A from P need not necessarily involve computing all atoms provable from P.

Encapsulation hides the program's clauses in compositions but exports its atomic consequences. For instance, the encapsulation of `N` is the set:

$$\{\texttt{q(a) :- true}\}$$

and the encapsulation of the union `M ∪ N` is the following set of clauses:

$$\{\texttt{p(a) :- true}, \texttt{q(a) :- true}\}$$

This thesis will demonstrate the use of the above four operators in LogicWeb. This choice is motivated by their proven utility in the numerous applications from [35] mentioned earlier, and for database manipulation mentioned in [8, 41]. Moreover, the operators deal with a notion of programs which corresponds closely to the notion of programs used in LogicWeb described in the next chapter. However, there are other composition operators proposed in the literature which are more specialised in their application and can not be expressed in terms of these four operators. Such operators include *tuple inheritance* which models a specific form of inheritance between logic programs [152], a form of restriction [7], and *import* which models a form of predicate import into a program [35]. In theory, an arbitrary number of specialised operators can be defined, and it is impractical to include all. The above four operators suffice to demonstrate the key applications described in this thesis. The use of other operators in LogicWeb is discussed further in Chapter 10.

The extensive survey of modularity in logic programming [47] used the operators union, encapsulation, and *overriding union* as the basic building blocks for compositions. Overriding union has been used to model object-oriented inheritance between programs. The overriding union of two programs P and Q can be expressed using union and restriction as $P \cup (Q \prec P)$.

2.2.3.2 Syntax

The meta-language for program expressions is the following:

$$\mathcal{E} ::= Program \mid \mathcal{E} \cup \mathcal{E} \mid \mathcal{E} \cap \mathcal{E} \mid \mathcal{E} \prec Program \mid \circledcirc(\mathcal{E})$$

$Program$ is the name of a logic program.

The Horn clause is extended to utilise meta-level operators, where the body consists of a finite sequence of elements each either an atomic formula or a meta-level formula of the form $B\ in\ E$ where B is an atomic formula and E, a program expression.

2.2.3.3 Operational Semantics

The operational semantics given here is found in [40]. A Plotkin style approach is adopted where a set of inference rules defines a derivation relation. For any goal formula G and program expression E, $E \vdash_\theta G$ denotes the fact that there exists a *top-down derivation* of G in E with computed answer substitution θ. The inference rules defining the derivation relation are in the form:

$$\frac{premises}{conclusion}$$

which is read as $conclusion$ holds whenever $premises$ hold. $\vdash_\theta$ is defined to be the smallest relation satisfying the inference rules below. In the rules, P denotes the name of a single program, E and F denote program expressions of the form $\mathcal{E}$, and ϵ denotes the empty (identity) substitution.

The language is an extension of standard pure Prolog. The rules which define derivation in standard Prolog are first given.

True.

$$\frac{}{E \vdash_\epsilon \texttt{true}} \tag{2.1}$$

`true` represents the empty goal, and is always derivable in any program expression E returning the empty substitution ϵ.

Conjunction.

$$\frac{E \vdash_\theta G_1 \quad \wedge \quad E \vdash_\gamma G_2\theta}{E \vdash_{\theta\gamma} G_1, G_2} \tag{2.2}$$

To derive a non-empty conjunction, derive each conjunct in turn. The proof of the second conjunct proceeds with the answer substitution θ returned by the proof of the first. The computed answer substitution for the conjunction is the composition of the answer substitutions obtained from the proof of the two conjuncts $\theta\gamma$. This rule specifies a left-to-right ordering in the evaluation of conjunctions.

Atomic formula.

$$\frac{E \vdash_{\theta} (H \text{ :- } G) \quad \wedge \quad \gamma = mgu(A, H\theta) \quad \wedge \quad E \vdash_{\delta} G\theta\gamma}{E \vdash_{\theta\gamma\delta} A} \tag{2.3}$$

To prove an atomic formula, obtain a clause (using substitution θ) whose head unifies with the formula with the most general unifier (mgu) γ, and then prove the body of the clause with answer substitution δ returning the composed answer substitution $\theta\gamma\delta$.

Obtaining program clauses.

$$\frac{P \; is \; a \; program \quad \wedge \quad (A \text{ :- } G) \in P}{P \vdash_{\epsilon} (A \text{ :- } G)} \tag{2.4}$$

This rule states that a clause is obtained from a program if the clause is a member of the program. The computed answer substitution is ϵ. This rule is generalised to represent how clauses are obtained from a composition of programs. The relation $E \vdash (A \text{:-} G)$, where E is a program expression, is defined by adding new rules for each type of composition. These rules define the set of clauses (virtually) found in a composition, and are defined based on the syntax of program expressions.

Union.

$$\frac{E \vdash_{\theta} (A \text{ :- } G)}{E \cup F \vdash_{\theta} (A \text{ :- } G)} \tag{2.5}$$

$$\frac{F \vdash_{\theta} (A \text{ :- } G)}{E \cup F \vdash_{\theta} (A \text{ :- } G)} \tag{2.6}$$

A clause is chosen from a union $E \cup F$ by choosing a clause from either E or F.

Intersection.

$$\frac{E \vdash_{\theta_1} (H_1 \texttt{ :- } G_1) \quad \wedge \quad F \vdash_{\theta_2} (H_2 \texttt{ :- } G_2) \quad \wedge \quad \gamma = mgu(H_1\theta_1, H_2\theta_2)}{E \cap F \vdash_{\theta_1\theta_2\gamma} (H_1 \texttt{ :- } G_1, G_2)} \tag{2.7}$$

A clause $H \texttt{:-} G$ is obtained from the intersection $E \cap F$ if there exists a clause $H_1 \texttt{:-} G_1$ obtained from E using θ_1 and a clause $H_2 \texttt{:-} G_2$ obtained from F using θ_2 such that $H_1\theta_1$ unifies with $H_2\theta_2$ via γ, $H = H_1\theta_1\gamma$, and $G = (G_1\theta_1, G_2\theta_2)\gamma$. Note that θ_1 and θ_2 are mutually exclusive since they are computed from different program expressions, i.e. variables in $H_1 \texttt{ :- } G_1$ are renamed apart from those in $H_2 \texttt{ :- } G_2$. This means, for example, that $G_1\theta_1\theta_2 = G_1\theta_1$ and $G_2\theta_1\theta_2 = G_2\theta_2$.

Encapsulation.

$$\frac{E \vdash_{\theta} A}{\circledcirc E \vdash_{\theta} A \texttt{ :- true}} \tag{2.8}$$

A clause $A \texttt{:-} true$ belongs to $\circledcirc E$ if A is provable in E.

Restriction.

$$\frac{E \vdash_{\theta} (A \texttt{ :- } G) \quad \wedge \quad pred(A) \notin preds(P)}{E \prec P \vdash_{\theta} (A \texttt{ :- } G)} \tag{2.9}$$

A clause is obtained from a program or composition denoted by E restricted by another (single) program P by choosing the clause from E and checking that it has not been defined in P. The requirement that E is restricted by a single program rather than a composition of programs allows the programmer control over the actual clauses which will be involved in the operation [40].

The final rule below defines the "in" operation which integrates the meta-language of program expressions into the object language. The rule states that to solve a goal of the form $A \ in \ F$, solve A in the program expression F:

$$\frac{F \vdash_{\theta} A}{E \vdash_{\theta} A \ in \ F} \tag{2.10}$$

The following is an example of how the rules are used to prove the goal `p(a)` in the union of `M` and `N`:

$$\dfrac{\dfrac{\texttt{M} \vdash_\epsilon \texttt{p(X) :- q(X)}}{\texttt{M} \cup \texttt{N} \vdash_\epsilon \texttt{p(X) :- q(X)}}(2.5) \quad \dfrac{\dfrac{\texttt{N} \vdash_\epsilon \texttt{q(a) :- true}}{\texttt{M} \cup \texttt{N} \vdash_\epsilon \texttt{q(a) :- true}}(2.6) \quad \dfrac{}{\texttt{M} \cup \texttt{N} \vdash_\epsilon \texttt{true}}(2.1)}{\texttt{M} \cup \texttt{N} \vdash_\epsilon \texttt{q(a)}}(2.3)}{\texttt{M} \cup \texttt{N} \vdash_{\{\texttt{X/a}\}} \texttt{p(a)}}(2.3)$$

First, rule (2.3) is used resulting in two branches. In the left branch, rule (2.5) is used to retrieve a clause from the union. The head of the clause unifies with the goal `p(a)` with the substitution of `X` by `a`, denoted by {`X/a`}. In the right branch, the subgoal `q(a)` resulting from applying {`X/a`} to `q(X)` is proven with the clause from `N` using the rules (2.3), (2.6) and (2.1).

2.2.3.4 Implementation

The above rules translate easily into an implementation which extends the vanilla meta-interpreter [35, 38, 42].

The following rules defining `::/2` for union, intersection, encapsulation, and restriction are added to the `demo/2` version of the vanilla meta-interpreter :

```
(E ∪ F)::(A :- B) :- E::(A :- B).
(E ∪ F)::(A :- B) :- F::(A :- B).

(E ∩ F)::(A :- (B, C)) :- E::(A :- B), F::(A :- C).

⊙E::(A :- true) :- demo(E, A).

(E ≺ P)::(A :- B) :- E::(A :- B), undefined(A, P).
```

The first argument of `::/2` is a program expression representing a composition of programs rather than a single program. The first two rules define union stating that a clause belongs to the union of two programs (or compositions) `E ∪ F` if it belongs to either `E` or `F`.

The third clause for intersection utilises the unification mechanism to ensure that the clause heads from both `E` and `F` are the same. This clause should not be

seen as breaking down the clause `A :- (B, C)` into two clauses `A :- B` and `A :- C` and checking for their existence in `E` and `F` respectively, but as forming the clause `A :- (B, C)` from the two clauses. When `::/2` is used to retrieve a clause from `E ∩ F`, this clause is invoked with only `E`, `F`, and `A` instantiated. On succeeding, the result returned is the body `(B,C)` which comprises the bodies of two clauses.

The fourth clause states that clauses from an encapsulation of a program `⊚E` are always of the form `A :- true` where `A` can be demonstrated from `E`.

In the fifth clause, `undefined(A, P)` checks that the predicate symbol of `A` is not defined in `P`, and if so, the clause `A :- B` from `E` is returned.

Rule (2.10) can be implemented by adding the following clause:

```
demo(E, A in F) :- demo(F, A).
```

Chapter 3

LogicWeb

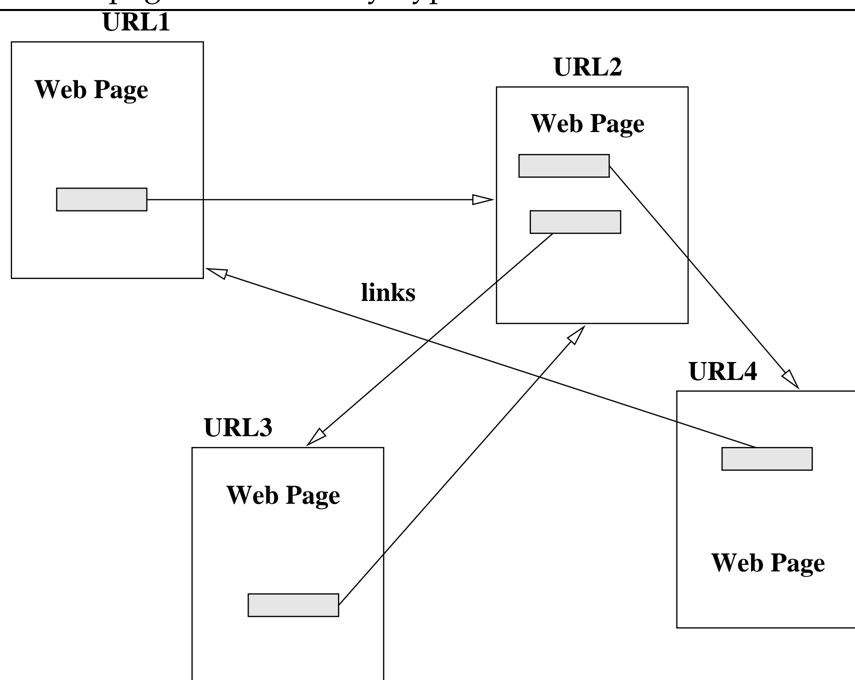
As mentioned in the introduction, LogicWeb is based on two key ideas: modelling Web pages inter-connected by links as distinct inter-related programs and building logic programming rules into Web pages. This chapter details both these aspects of LogicWeb and the LogicWeb language. In particular, compositional logic programming ideas introduced in the previous chapter are applied for manipulating Web pages with rules.

We begin in Section 3.1 by introducing *LogicWeb programs*, the central notion in LogicWeb. The components of LogicWeb programs are detailed in Section 3.2. Section 3.3 describes the language in which LogicWeb programs are written and manipulated. The EBNF syntax and an operational semantics for the language are given. The operational semantics shows how LogicWeb programs are executed and provides a basis for language implementation. Section 3.4 outlines how LogicWeb programs are used for building applications on the Web.

3.1 A Logic Programming Model of the Web

LogicWeb models the Web as a collection of inter-related logic programs. In LogicWeb, pages can contain rules, and the hypertext Web structure is captured as

Figure 3.1 Web pages connected by hypertext links.



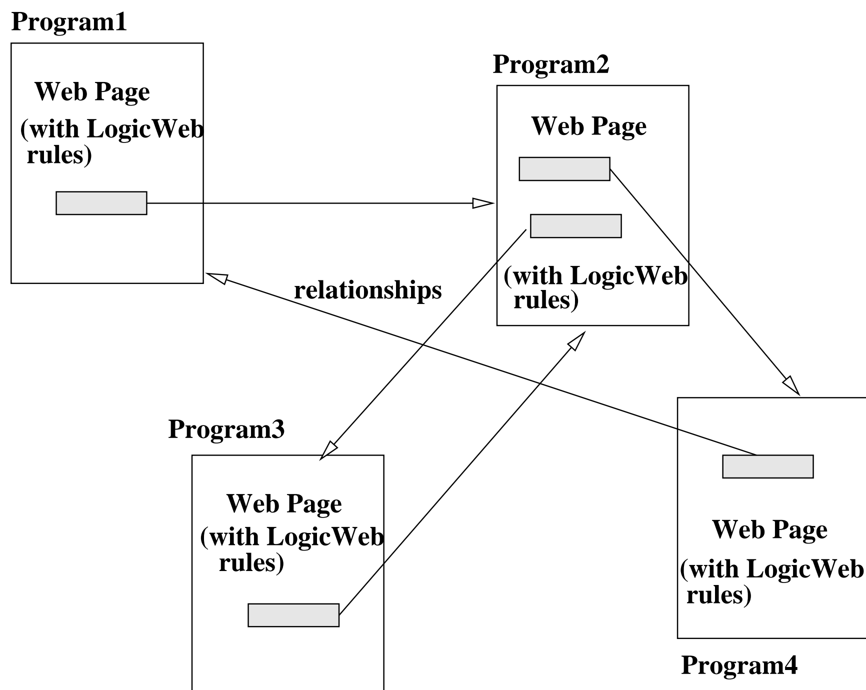
follows: each page is represented by a logic program called a *LogicWeb program* and each hypertext link is a relationship between LogicWeb programs. Figure 3.1 shows the hypertext structure of the Web consisting of a collection of pages connected by hypertext links. As noted in Chapter 2, each page is referred to by a URL. Figure 3.2 depicts the LogicWeb model showing pages augmented with rules and hypertext links as relationships between programs. The LogicWeb program corresponding to a page is identified using the URL of the page, and is made up of facts storing the page's HTML components and the rules (if any) embedded within the page. LogicWeb programs are described in detail in the next section.

This model is not simply a pleasing abstraction, but has pragmatic uses. The modelling of pages as programs and links as relationships between programs allows reasoning with the Web (with embedded rules) using well-established meta-programming techniques in logic programming. The LogicWeb programs

are at the object level, and the relationships between them and rules manipulating them are at the meta-level.

As emphasised in the introduction, rules can be viewed as code or data. Rules embedded within a page can be used to encode dynamic behaviour turning a page into a live information entity which uses its rules to respond to user queries. The rules within a LogicWeb program can reason with other LogicWeb programs, i.e. the behaviour of a LogicWeb program may depend on other programs. Alternatively, rules within a collection of pages can form a repository of knowledge which is published on the Web and sufficiently structured to be processed automatically.

Figure 3.2 The LogicWeb model where pages augmented with rules are programs and hypertext links are relationships between programs.



3.2 LogicWeb Programs

In this section, we look into the details of LogicWeb programs. LogicWeb programs are constructed from meta-information and page contents in HTTP response objects. As mentioned in Chapter 2, the three types of HTTP requests used in this thesis are HEAD, GET, and POST, and hence, three kinds of HTTP response objects are considered. When successful, the HEAD request returns only the meta-information about a page, whereas GET and POST requests return both the meta-information and page contents. A HTTP response object is mapped to a logic program. The logic program derived from a GET or POST response object consists of facts storing the page's meta-information, HTML text, and embedded rules. The logic program translated from a HEAD response object comprises mainly facts storing meta-information. The components of the program (including the names of the facts) for each type of HTTP response object are shown in Figure 3.3.

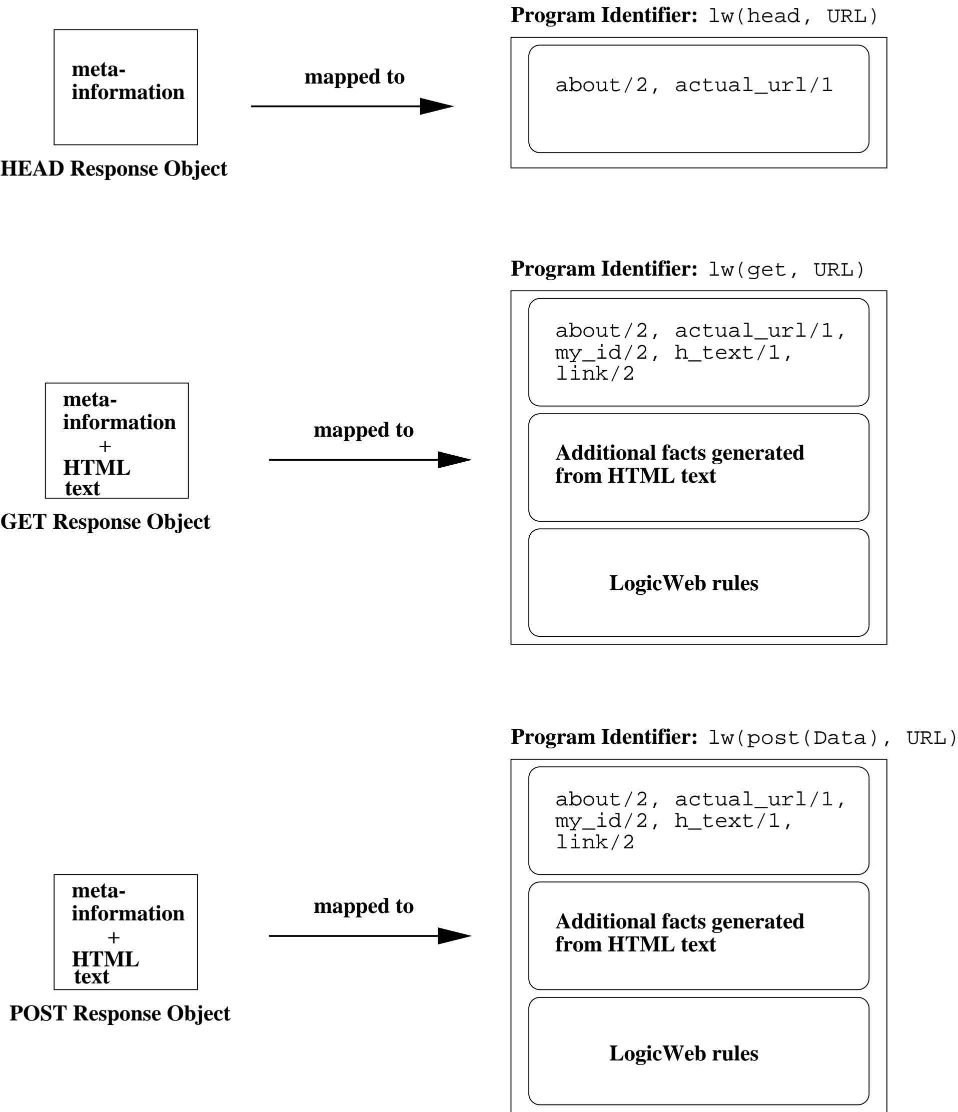
A HEAD response object corresponds to a LogicWeb program consisting of two types of facts storing the meta-information and the actual URL of a page. The two types of facts are:

- `about(`*`FieldName`*`, `*`Value`*`).` The `about/2` facts store the meta-information about a page supplied by the server as described in Section 2.1.2.3. For instance, the `about/2` facts for a page are:

  ```
  about(mime_version, "1.0").
  about(server, "CERN/3.0").
  about(date, "Thursday, 26-Dec-96 14:47:05 GMT").
  about(content_type, "text/html").
  about(content_length, "2322").
  about(last_modified, "Sunday, 08-Dec-96 20:48:29 GMT").
  ```

  In the case of redirections, only the meta-information of the page is stored, and not header fields indicating the redirection (e.g., the `Location` field described in Section 2.1.2.3).

Figure 3.3 The identifiers and components of three types of LogicWeb programs.



- `actual_url(URL)`. This holds the URL of the page whose meta-information is retrieved from the server. If several redirections occur, then `actual_url/1` stores the actual URL of the page.

A program corresponding to a HEAD response object has the identifier `lw(head, URL)`. Programs corresponding to GET and POST response objects have identifiers of the form `lw(get, URL)` and `lw(post(Data), URL)` respectively. A GET or POST response object corresponds to a LogicWeb program consisting of:

1. five types of facts generated from the meta-information returned by the page's server, and the HTML text of the page, and
2. rules included within the page via a special tag.

The five types of facts are `about/2` and `actual_url/1` described above, where in the case of POST, `actual_url/1` holds the actual URL of a CGI script, and the following three types of facts:

- `my_id(Type, URL)`. The argument *Type* stores the type of the program which is either the term `get` or `post(Data)`. In the case of `get`, *URL* is the URL of the page used in a GET request, and for `post/1`, *Data* is information posted to the CGI script at *URL*. `my_id/2` stores the URL used in the original Web request. This URL can differ from the URL stored in `actual_url/1` if a redirection occurs. For example, the original URL in a request is

  ```
  http://www.cs.mu.oz.au/~swloke
  ```

  but the actual URL (returned in the `Location` field described in Section 2.1.2.3) is

  ```
  http://www.cs.mu.oz.au/~swloke/index.html
  ```

- `h_text(HTMLText)`. This contains the complete standard HTML text of the page (including the HTML tags) as a string.

- `link(Label, URL)`. This stores a page's link information. For instance, the marked-up anchor:

  ```
  ...<A HREF="http://www.cs.mu.oz.au/">Melbourne U</A>...
  ```

  becomes:

  ```
  link("Melbourne U", "http://www.cs.mu.oz.au/").
  ```

  Relative URLs in anchors are expanded into absolute URLs which are stored in `link/2`. `link/2` is redundant in that it stores information already contained in `h_text/1`. However, `link/2` represents explicitly as relationships the hypertext links between programs. The advantage from the programmer's point of view is that link information is easily accessed and searched over as facts without having to re-parse the page each time link information is required.

These five types of facts suffice to model the basic page and link structure of the Web. If other components of the HTML text are needed in applications, they can be similarly extracted and stored in facts to simplify their use. In the next chapter, additional components which should be extracted to support applications are prescribed.

We now consider how rules are embedded within pages. Rules are included within a page using the tags "`<LW_CODE>`" and "`</LW_CODE>`". Typically, the code appears inside a verbatim container "`<PRE>...</PRE>`" or a comment container "`<!--...-->`" so that it is uninterpreted by the browser. Figure 3.4 depicts a page containing rules which describe research interests.

`interested_in/1` states that the author is interested in several topics and in what his friends are interested in. The LogicWeb operator "`#>`" evaluates the

Figure 3.4 A page with rules describing research interests.

```
<HTML>
<HEAD>
<TITLE>Seng Wai Loke's Home Page</TITLE>
</HEAD>

<BODY>
<H1>Seng Wai Loke's Home Page</H1>
I'm from the <A HREF="http://www.cs.mu.oz.au/">
Department of Computer Science</A> at the
<A HREF="http://www.unimelb.edu.au/">
University of Melbourne</A>.
<!--
<LW_CODE>
interests(["Logic Programming", "AI", "Web", "Agents"]).

friend_home_page("http://www.cs.mu.oz.au/~friend1/").
friend_home_page("http://www.cs.mu.oz.au/~friend2/").

interested_in(X) :-
  interests(Is), member(X, Is).
interested_in(X) :-
  friend_home_page(URL),
  lw(get, URL)#>interested_in(X).
</LW_CODE>
-->
</BODY>
</HTML>
```

goal `interested_in/1` against a program into which each friend's homepage is translated. This operator is similar to the "*in*" operator described in Chapter 2, and allows the contents of a page to be accessed from rules in another page. This operator is discussed further in the next section.

The mapping of a page to a LogicWeb program is illustrated below. Assuming that the page shown in Figure 3.4 was retrieved using the GET method and the URL `http://www.cs.mu.oz.au/~swloke/` without redirection, the LogicWeb program corresponding to the page comprises the following:

1. `about/2` facts for the page's meta-information (not shown in Figure 3.4).

2. `actual_url("http://www.cs.mu.oz.au/~swloke/").`

3. `my_id(get,"http://www.cs.mu.oz.au/~swloke/").`

4. `h_text("<HTML>...</HTML>").` This fact stores all the HTML text including tags except for the rules within the “<LW_CODE>” and “</LW_CODE>” tags.

5. Two `link/2` facts:

```
link("Department of Computer Science",
     "http://www.cs.mu.oz.au/").
link("University of Melbourne",
     "http://www.unimelb.edu.au/").
```

6. The rules within the “<LW_CODE>” and “</LW_CODE>” tags.

3.3 The LogicWeb Language

The language in which LogicWeb programs are written and manipulated is an elementary Edinburgh-style Prolog with additional program operators. A crucial idea is that the identifiers of LogicWeb programs are treated as first-class entities. This makes it easier for programs to access and manipulate each other, and allows arbitrary relationships to be defined between programs. LogicWeb programs can be queried in a similar way as ordinary logic programs using the operator “#>” as already seen in the previous section, and manipulated using meta-level operators similar to those given in Section 2.2.3.1.

This section first discusses informally the operational meaning of “#>” and the meta-level operators for composing LogicWeb programs, illustrating their use with examples. Then, the formal syntax and an operational semantics for the LogicWeb language are given. Finally, the relationship between the operational semantics and a fixpoint semantics is considered.

3.3.1 Querying and Manipulating LogicWeb Programs

3.3.1.1 Context Switching

We first consider the operational meaning of the operator "`#>`" which is called *context switching*. A *LogicWeb goal* is a goal formed using context switching. The following LogicWeb goal applies a goal to a program specified by its identifier:

```
lw(get, URL)#>Goal
```

If the program is not present on the client-side, then its page will be downloaded and transformed into a LogicWeb program before the query is evaluated. However, if the program is already present, then the goal is executed immediately. Thus, the "`#>`" operator permits the programmer to think of Web computation as goals applied to programs, with no need for explicit Web page retrieval or parsing.

The current context of a LogicWeb goal, i.e. the program or composition of programs in which the goal is evaluated, is ignored. For example, consider the following goal:

```
?- lw(get, "URL0")#>((lw(get, "URL1")#>interested_in(X)), goal).
```

The program `lw(get, "URL0")` is the context for the conjunction of the LogicWeb goal `lw(get, "URL1")#>interested_in(X)` and `goal`. However, the evaluation of `interested_in(X)` only uses the rules in `lw(get, "URL1")`, ignoring those in `lw(get, "URL0")`, whereas `goal` is evaluated in `lw(get, "URL0")`.

When a LogicWeb goal is evaluated, it is possible that its corresponding program can not be created. This may happen if the HTTP request for the required page fails due to server or network overload, which means that no page is delivered to the LogicWeb system. A failure of this kind is represented by the failure of the corresponding LogicWeb goal, which makes the problem observable at the application level. This means that LogicWeb goal failure is either due to download failure, or ordinary goal evaluation failure (when the downloads succeed).

Since a LogicWeb goal `E#>G` interacts with the Web as part of its evaluation, another design choice is to specify the extent of that interaction when the same goal is invoked multiple times.

When a goal is called the first time, there are two possible cases:

1. *The requested HTTP response object is downloaded and the corresponding program is created*. If the same goal is invoked at a later time, there are two options:

   - reissue the HTTP request and create a new program, or
   - keep using the existing program.

   The first option can lead to inconsistent query results. If the corresponding page has been changed on its server between the first and second HTTP requests, then the new program may be different from the original. Consequently, the goal which may have succeeded with the original program may now fail or succeed with different bindings. In contrast, the second option guarantees consistent results since the program is never changed.

2. *The requested HTTP response object could not be downloaded*. If the same goal is later invoked, either the:

   - HTTP request can be reissued, perhaps returning the actual page this time, or
   - goal failure is forced, in order to be consistent with the first invocation.

For case (1), the second option is used (i.e., repeated downloads are avoided). For case (2), the first option is implemented (i.e., allow a failed HTTP request to be retried).

In case (1), reuse is chosen primarily to retain consistency between goal evaluations. Moreover, most Web pages change quite infrequently and can safely be assumed to be constant over the duration of a query evaluation (though there are

exceptions). Another advantage is that LogicWeb programs are cached on the local host, reducing the number of network accesses, and so increasing efficiency. This consistency also allows information to be inferred about goal failures.

The failure of a `E#>G` goal is ambiguous since it may be caused by the client's inability to retrieve the program, or because `G` is not provable in `E`. This ambiguity abstracts away details about failures and is not a problem if the programmer only wants to know if a LogicWeb goal succeeds or fails. However, the programmer may want to resolve this ambiguity. This ambiguity is resolved by subsequently calling `E#>true`, which will only fail if `E` is not present. However, if `E#>true` succeeds, then this may mean that `G` was not provable *or* that `E` was just downloaded successfully in order to evaluate the `true` goal. The situation can be resolved by invoking `E#>G` again, which can now only fail if `G` is not provable in `E`. Another way to resolve the situation is to determine if a program exists by invoking a built-in predicate `program_exists/1` in the implementation of the LogicWeb language (see Appendix B).

Such testing to disambiguate goal failure can be coded up by the programmer. For example, the following rules use the second way to resolve the ambiguity:

```
try_goal(LWProgramID, Goal, success) :-
  LWProgramID#>Goal.          % download and evaluation succeeded
try_goal(LWProgramID, Goal, ReasonForFailure) :-
  (program_exists(LWProgramID) ->
    ReasonForFailure = evaluation_failure       % program exists
  ;
    ReasonForFailure = download_failure % program does not exist
  ).
```

`try_goal/3` will always succeed. If the LogicWeb goal fails, a test is made to determine if the program has been downloaded. If the program exists (and hence, has been downloaded), then it must have been that `Goal` was not provable in the downloaded program. If the program does not exist, then the attempt to download the program must have failed. This test incurs additional computation costs

(involving a search through all existing programs) and adds complexity, but can be packaged up in the system as a variant of "`#>`".

There are alternatives to reuse, which still reduce network access overheads. For example:

- *A program is replaced only if it has been modified*. This reduces network access costs by issuing a HEAD request to determine if a page has been modified instead of a more expensive GET message. However, the result of LogicWeb goal evaluations may change after a new program has been installed.

- *A program is replaced periodically*. A separate agent can periodically update the cache independently of when the programs are used. However, it is hard to determine the desirable frequency of such updates. HTTP permits servers to return the expiry date of a page as meta-information [25], which would be stored in `about/2`. However, servers do not necessarily have to return such information (and many do not).

- *Programs are replaced automatically when the evaluation of a query completes*. This makes LogicWeb goal evaluations consistent for the duration of a query. A disadvantage is that a user may want to try a different query with the same set of programs. In the current LogicWeb implementation described in Chapter 4, the user can clear or update locally stored programs at any time through a utility program (itself a LogicWeb application).

In case (2), repeated download attempts allow failed HTTP requests to be retried. This will permit useful work to be accomplished if a retry succeeds, though at the cost of some inconsistency between goal evaluations (or non-logical behaviour).

The use of context switching is illustrated below with two simple search utilities; a more complicated search tool will be developed in Chapter 5.

The first example finds a similar page given a starting URL. The query:

```
?- similar_pg("http://www.cs.mu.oz.au/~ad", P).
```

will try to bind `P` to a URL which is similar to the given address. `similar_pg/2` is defined as:

```
similar_pg(CurrURL, SimilarURL) :-
  lw(get, CurrURL)#>interested_in(Topic),
  lw(get, CurrURL)#>link(Topic, SimilarURL).
```

The program obtains an `interested_in/1` topic from the given page and uses it to select a link leaving that page. The evaluation of the query will probably involve backtracking as it is unlikely that every topic of interest has an associated link. Note that the program `lw(get, CurrURL)` is downloaded only once though it is utilised repeatedly in backtracking and in both LogicWeb goals.

A drawback of this code is that it assumes that the page contains `interested_in/1` and `link/2` facts. It will have the latter, since they are generated automatically (unless there are no links leaving the page). It is less certain that there will be an `interested_in/1` predicate. This can be remedied by including an extra clause in `similar_pg/2` which analyses the page using the `h_text/1` string. Inspecting overly large pages is avoided by first issuing a HEAD request to determine the content length of a page, and checking that the content length is below a specified threshold.

The second example uses the `h_text/1` approach to find a page below a certain size relevant to a given subject and starting page. The query:

```
?- relevant_pg("Logic Programming", 10 000,
                              "http://www.cs.mu.oz.au/~ad", P).
```

will bind `P` to a URL which is related to logic programming, where the page is less than 10 000 bytes in length, and is linked to the starting page. `relevant_pg/4` is defined as:

```
relevant_pg(Subject, MaxSize, StartURL, URL) :-
  lw(get, StartURL)#>link(_, URL),
  lw(head, URL)#>about(content_length, L),
  name(AL, L),    % a Prolog built-in to convert string to atom
  AL < MaxSize,
  lw(get, URL)#>h_text(Source),
  contains(Source, Subject).
```

`relevant_pg/4` selects a link in the starting page without concerning itself about the link's label. If the size of the page linked to is less than `MaxSize` bytes, then the text of that page is retrieved and passed to the LogicWeb built-in predicate `contains/2` to see if it contains the subject string. `relevant_pg/4` only relies on the predicates automatically generated from HTML text and meta-information, and so should be more robust than `similar_pg/2`. This example shows the ease with which meta-information and the HTML page text can be accessed.

3.3.1.2 Composing LogicWeb Programs

LogicWeb programs can be composed enabling their behaviour or content to be combined. The operators for composing LogicWeb programs are based on the operators described in Section 2.2.3.1 and are collectively called *LW-composition operators*. The operators are *LW-union* ("+"), *LW-intersection* ("*"), *LW-restriction* ("/"), *LW-encapsulation* ("@"), and *LW-reduce* ("<>"). These operators are used to form expressions called *LogicWeb program expressions*, or simply program expressions, when context differentiates these expressions from the program expressions of Chapter 2.

LogicWeb programs are composed after they have been retrieved from the Web, and this behaviour means that the semantics of the LW-composition operators can be viewed as a variant of those described in Section 2.2.3.1, extended to address issues related to page downloading and the mapping of pages to programs. For example, consider the following LogicWeb goal utilising LW-union:

```
?- (lw(get, "URL0") + lw(get, "URL1") + lw(get, "URL2"))#>p(X).
```

This goal expresses the programmer's intention to download three programs, and then evaluate `p(X)` in the set-theoretic union of their clauses. However, if any of the programs can not be created (e.g., because a page can not be retrieved over the network), then the goal fails (though the goal may also fail in the usual manner with all the programs present).

It is convenient to use Prolog's list manipulation capabilities to work with collections of programs. LW-reduce is similar to the *reduce* in functional programming and applies a binary LW-composition operator between the elements of a list starting from the leftmost element. The following equation illustrates its meaning:

```
(+)<>[lw(get, "URL0"), lw(get, "URL1"), lw(get, "URL2")] =
               lw(get, "URL0") + lw(get, "URL1") + lw(get, "URL2")
```

LW-reduce can be used with LW-intersection in a similar way, for example, by replacing "+" by "*" in the above equation.

LW-reduce is used with LW-restriction in a slightly different way. LW-reduce applies the operator "/" to a pair whose first member is a program expression, and the second a list of program identifiers. The following equation illustrates this use:

```
(/)<>(lw(get, "URL0") + lw(get, "URL1"),
     [lw(get, "URL2"), lw(get, "URL3")]) =
               (( lw(get, "URL0") + lw(get, "URL1") )
                      / lw(get, "URL2")) / lw(get, "URL3")
```

The failure of a LogicWeb goal when a program named in a program expression can not be obtained means that we either obtain all possible solutions (e.g., on backtracking) when all the required programs are downloaded, or no solutions when any program is absent.

In some situations, this behaviour seems too restrictive. For example, we may only want one solution, and do not care which one. In the above LogicWeb goal which evaluates `p(X)` against the LW-union, suppose that `p(X)` can succeed using `lw(get, "URL0")` alone. Allowing `p(X)` to succeed in the LW-union when

only `lw(get, "URL0")` can be created permits at least one solution to be returned. However, this compromises the declarative semantics since `p(X)` is evaluated in only one program when it should be evaluated in three.

One way to obtain as many solutions as possible is to fetch all the required programs before using them in a program expression. For instance:

```
?- setof(P,
         (member(P, [lw(get, "URL0"), lw(get, "URL1"),
                     lw(get, "URL2")]),
          P#>true),
         Ps),
   setof(X, ((+)<>Ps)#>p(X), Xs).
```

The first `setof/3` goal attempts to download `URL0`, `URL1`, and `URL2`, and returns the identifiers of the successfully downloaded ones in `Ps`. `Ps` is then employed by LW-reduction in the second `setof/3` goal. LogicWeb goals of the form `P#>true` can be generally employed to download programs without performing any other work.

LW-composition operators encourage software engineering principles. For example, the LogicWeb program components of an application can be distributed over several Internet hosts, and integrated when needed. This approach is utilised in the Web-based databases discussed in Chapter 6.

The LW-composition operators allow collections of program identifiers to be treated as first-class entities, which further extends the generality of programs. For example, `links/2` below obtains a list of links in a program expression `C`, but this variable can be bound to any composition of programs by using suitable operators.

```
links(C, Ls) :-
  setof(link(A, B), C#>link(A, B), Ls).
```

The following query retrieves links from the union of three pages:

```
?- links(lw(get, "http://www.cs.mu.oz.au/~swloke/") +
         lw(get, "http://www.cs.mu.oz.au/~ad/") +
         lw(get, "http://www.cs.mu.oz.au/~leon/"), Links).
```

3.3.1.3 Utilising the Current Context

The operators discussed so far ignore the current context when proving goals. However, LogicWeb includes a *context* operator, denoted by "`(#)`", which can be used to represent the current context in a program expression. For instance, in the goal:

```
?- lw(get, "URL0")#>(((#) + lw(get, "URL1"))#>interested_in(X)).
```

"`(#)`" is instantiated to `lw(get, "URL0")` when the goal is evaluated. "`(#)`" can be used in place of a program identifier in any expression, which provides very useful expressive power. For example, it can be employed to model Miller's implication goal introduced in Chapter 2, and contextual logic programming [151].

Recall that Miller's implication goal of the form $D \supset G$ when evaluated in P (the current context of the goal) causes the evaluation of G in $P \cup D$. An implication goal can be simulated using the context operator by a goal of the form `((#) +` D`)#>`G, which denotes `(`P `+` D`)#>`G where P is the current context. The context operator can be used to generalise the implication goal to $D \supset_{\oplus} G$ which proves G in $P \oplus D$, where $\oplus$ denotes a composition operator. $D \supset_{\oplus} G$ would be represented by `((#)` $\oplus$ D`)#>`G. The expressive power of implication goals for implementing memoing, abstract datatypes, and modular logic programming is detailed in [149].

One of the key ideas in contextual logic programming is the *context extension* operator (denoted by "`>>`"). A goal of the form Q `>>` G is evaluated in a program P by evaluating G using the predicates in Q and those in P which are not defined in Q. The goal Q `>>` G can be rewritten as the LogicWeb goal `(`Q `+ (@(#) /` Q`))#>`G.

As mentioned in [151], the advantages of using the current context in programming include modularity for software construction, greater generality, and reusability of predicates. For example, "`(#)`" allows predicates to be defined whose definitions are context-dependent.

Consider an application for translating dates in numerical form (e.g., “25/12/97”) into English (e.g., “25 December 1997”), located at `http://trans.date.english/`. The top-level predicate is `translate_date/2`, which uses `date_components/4` to break a numerical date into its three numerical components, uses `translate_month/2` and `translate_year/2` to convert the month and year into another form, and uses `flatten/2` to merge the new components into a date:

```
my_id(get, "http://trans.date.english/").

% predicate to perform translations
translate_date(NumericalDate, EnglishWordDate) :-
  date_components(NumericalDate, Day, NMonth, NYear),
  translate_month(NMonth, Month),
  translate_year(NYear, Year),
  flatten([Day, " ", Month, " ", Year], EnglishWordDate).

% predicate to break a numerical date into its components
% e.g., 25/12/97 to "25", 12, and 97
date_components(NumericalDate, Day, NMonth, NYear) :-
  ..../* definition of date_components/4 */...

% predicate to translate a numerical month into English
translate_month(NMonth, EnglishMonth) :-
  ..../* definition of translate_month/2 */...

% predicate to expand a year
translate_year(NYear, Year) :-
  ..../* definition of translate_year/2 */...
```

This program can be modified to translate dates into French instead of English (e.g., “25 Decembre 1997”) by using a different definition for `translate_month/2`, but reusing all the other predicates. Assuming that the definition of `translate_month/2` which converts numerical months into French is stored in `http://trans.date.french/`, the LogicWeb program for translating dates into French is given below:

```
my_id(get, "http://trans.date.french/").

% predicate to translate numerical months into French
translate_month(NMonth, FrenchMonth) :-
  ..../* definition of translate_month/2 */...

% translate from lingua to French
translate_date_french(NumericalDate, FrenchWordDate) :-
  ((lw(get, "http://trans.date.english/") / (#)) + (#))#>
                  translate_date(NumericalDate, FrenchWordDate).
```

The program expression in the `translate_date_french/2` rule states that all the predicates from `lw(get, "http://trans.date.english/")` except `translate_month/2` are added to the current context (the program `lw(get, "http://trans.date.french/")`). The use of LW-restriction excludes `translate_month/2` from `lw(get, "http://trans.date.english/")` since it is already defined in the current context. The LW-union operation supplies `translate_month/2` from the current context.

3.3.2 EBNF Syntax

This subsection gives the EBNF syntax of the pure LogicWeb language, which is roughly given by the equation:

$$pure\ LogicWeb\ language\ =\ pure\ Prolog\ +\ LogicWeb\ operators$$

A *pure* LogicWeb program is a finite set of clauses of the form:

$$[\forall x](\mathcal{A}\ \texttt{:-}\ \mathcal{G})$$

where $\mathcal{G}$ is defined recursively as:

$$\mathcal{G} ::= \mathcal{A} \mid \mathcal{E}\ \texttt{\#>}\ \mathcal{G} \mid (\mathcal{G}, \mathcal{G})$$

The LogicWeb language replaces the "$A\ in\ F$" goal in the compositional logic programming language presented in Chapter 2 with the LogicWeb goal.

$\mathcal{E}$ defines LogicWeb program expressions precisely:

$$\mathcal{E} ::= \mathcal{P} \mid \mathcal{E}\texttt{+}\mathcal{E} \mid \mathcal{E}\texttt{*}\mathcal{E} \mid \mathcal{E}\texttt{/}\mathcal{P} \mid \texttt{@}E \mid \texttt{(/)<>(}\mathcal{E}\texttt{,}\mathcal{L}_{(\mathcal{P})}\texttt{)} \mid \texttt{(}\oplus\texttt{)<>}\mathcal{L}_{(\mathcal{E})}$$

$$\mathcal{P} ::= \texttt{lw(head,}URL\texttt{)} \mid \texttt{lw(get,}URL\texttt{)} \mid \texttt{lw(post(}\mathcal{L}_{(\mathcal{F})}\texttt{),}URL\texttt{)} \mid \texttt{(\#)}$$

$$\mathcal{F} ::= \texttt{field(}Name\texttt{,}Value\texttt{)}$$

$$\mathcal{L}_{(\mathcal{I})} ::= \texttt{[]} \mid \texttt{[}\mathcal{I}\texttt{|}\mathcal{L}_{(\mathcal{I})}\texttt{]}$$

$$\oplus ::= \texttt{+} \mid \texttt{*}$$

$\mathcal{L}_{(\mathcal{I})}$ defines a Prolog list of items each of which is described by a nonterminal $\mathcal{I}$. URL is a URL, and $\mathcal{F}$ is a query attribute submitted to a CGI script. $Name$ is the name of a query attribute, and $Value$ is the value submitted to the server for the corresponding attribute. The context operator "(#)" can appear anywhere a program identifier can but "(#)" must be instantiated to a program identifier when used in the right argument of LW-restriction (which must always be a single program identifier as noted in Section 2.2.3.3). This requirement can be enforced by the programmer, or a run-time check can be built into an interpreter for the language.

3.3.3 An Operational Semantics for LogicWeb Programs

This subsection presents an operational semantics for the pure LogicWeb language which makes precise how LogicWeb programs interact with the Web during goal evaluation, and how these interactions affect goal evaluation.

Interactions with the Web are modelled by calls to an *oracle function*[1]. A Turing machine augmented with an oracle was used to formalise queries on the Web in [145] by using the oracle to model Web data accesses. Given a node identifier (i.e., a URL), the oracle maps it to the node's content if the node exists, or to a

[1] The idea of an oracle function has its roots in *oracle Turing machines* [89]. Such a machine is similar to a Turing machine but has an oracle function which it "consults" at certain states of its computation. The oracle function corresponds to a hypothetical subroutine computing the solution to a subproblem.

special symbol if not. A similar oracle function is defined below to model the accessing of LogicWeb programs.

DEFINITION 1 (ORACLE FUNCTION)
The oracle function

$$download : LW\,ProgramIDs \rightarrow LW\,Programs \cup \{\perp\}$$

takes a LogicWeb program identifier P (of the form defined by $\mathcal{P}$), and returns the program D_P if it is successfully created. Failure to obtain a program is represented by returning the symbol $\perp$.

$$download(P) = \begin{cases} D_P & \text{if the program denoted by identifier } P \text{ is} \\ & \text{successfully created,} \\ \perp & \text{otherwise.} \end{cases}$$

$download$ attempts to download a HTTP response object and translate it into a LogicWeb program in the way specified in Section 3.2. Due to the nondeterministic nature of the Web (as seen by a client) as explained in Section 2.1.2.3, the result of $download$ is unpredictable: a HTTP request can either succeed or fail, and two calls to $download$ with the same arguments, but at different times, can return different results. $download$ represents the idea that LogicWeb programs can only be accessed via their identifiers. How $download$ interacts with the Web depends on whether its program identifier invokes a HEAD, GET, or POST method, but the result is always a program or $\perp$.

With an ordinary (i.e., without Web interactions) logic programming language (e.g., the language in Chapter 2), a proof search on succeeding returns an answer substitution set. With the LogicWeb language, a goal is evaluated with respect to downloaded LogicWeb programs, and derivation cannot proceed if any of these programs are unavailable. As a result of (possible) downloads (and creation of LogicWeb programs) occurring during a goal derivation, the derivation does not only compute an answer substitution (as in the derivation relation of Chapter 2),

but as a side-effect, also computes a new set of downloaded LogicWeb programs extending an existing program set. The set of downloaded programs is extended by calling the function $add_programs$, which is invoked by LogicWeb goals as will be shown later (in rule (3.10)). $add_programs$ takes the set of existing LogicWeb programs and maps it to a new set using $download$.

DEFINITION 2 (ADDITION OF LOGICWEB PROGRAMS)
Let $\wp$ denote powerset. The function

$$add_programs : \wp(LW\,Programs) \times \wp(LW\,ProgramIDs) \rightarrow \wp(LW\,Programs)$$

takes a set S of programs and a set I of program identifiers and returns a new set $add_programs(S, I)$ consisting of S augmented with newly created programs mentioned in I but previously not in S:

$$add_programs(S, I) = S \cup \{D_P \mid P \in (I \setminus ids(S)),\ D_P = download(P),\ D_P \neq \bot\}$$

where ids is a function that takes a set of programs and returns the identifiers of the programs in the set, i.e. $ids(S)$ is the set of identifiers of the programs in S.

The above definition of $add_programs$ captures the following points about the semantics of a LogicWeb goal discussed in Section 3.3.1:

1. If a program required by a LogicWeb goal evaluation does not exist locally (i.e., is not in S), then an attempt is made to download and create the program.

2. Existing programs are not replaced. $add_programs$ is a monotonically increasing function. Programs mentioned in I which are already in S are not downloaded again. This implies that if all the programs mentioned in I are already in S then no HTTP requests are made, thereby avoiding potential HTTP request failures.

3. A LogicWeb goal which previously failed may succeed when called at a later time. The reason is that $download$ represents download failure as $\bot$

which means that the associated program identifier is not recorded. Thus, $download$ can be invoked with that program identifier again, which may result in a program being added to the existing set. This in turn may permit the LogicWeb goal to succeed.

It is assumed that there is no limit to local storage space for downloaded LogicWeb programs, though such a limit exists in the implementation of the LogicWeb system described in Chapter 4.

A derivation relation involving the set of downloaded LogicWeb programs specifies the computation model involving interactions with the Web.

DEFINITION 3 (DERIVATION RELATION)
For any goal formula G and program expression E, we denote by $S, E \vdash_{\theta}^{S'} G$ the fact that there exists a top-down derivation *of G in E starting with the set S of existing LogicWeb programs and ending with computed answer substitution θ and created program set S'. A top-down derivation or proof of G in E starting with S and ending with S' and θ is a finite tree such that:*

1. *the root node (bottom node) is labelled by the string "$S, E \vdash_{\theta}^{S'} G$";*
2. *the internal nodes are derived according to the set of inference rules given below; and*
3. *all the leaves of the tree are either empty or labelled by a string not containing the symbol "$\vdash$" (e.g., the label "$(A \text{ :- } G) \in P$").*

The difference between S and S', $S' \setminus S$, is the set of LogicWeb programs created during the derivation of G. $\vdash_{\theta}^{S}$ is defined to be the smallest relation satisfying the inference rules below. If $S, E \vdash_{\theta}^{S'} G$, then the goal G $succeeded$ when evaluated in E using S. Otherwise, the goal G $failed$ when evaluated in E using S.

Failure in finding a matching clause (i.e., one whose head unifies with the goal) or download failure can result in failure in finding (using the proof rules given below) a top-down derivation for the goal.

Given a top-down derivation $S, E \vdash_{\theta}^{S'} G$, if the derivation does not involve interactions with the Web or if no programs are downloaded, then S' is the same as S (i.e., $S' = S$). If programs are downloaded, then S is extended to S' ($\supseteq S$). The difference $S' \setminus S$ represents the effect of Web interactions during the derivation.

A context is defined for each node in a top-down derivation whose label is of the form "$S, E \vdash_{\theta}^{S'} G$".

DEFINITION 4 (CONTEXT OF A GOAL)
Given the node label "$S, E \vdash_{\theta}^{S'} G$", E is the $context$ (of G).

In the rules, P denotes single program identifiers of the form $\mathcal{P}$, and E and F denote program expressions of the form $\mathcal{E}$ as defined in Section 3.3.2. $L_{(\mathcal{I})}$ denotes a list of the form $\mathcal{L}_{\mathcal{I}}$. ϵ denotes the empty (identity) substitution.

3.3.3.1 Pure Prolog

The following rules define derivation in pure Prolog taking into account the creation of LogicWeb programs:

True.

$$\frac{}{S, E \vdash_{\epsilon}^{S} \texttt{true}} \tag{3.1}$$

`true` is always derivable in any program expression E without any change to the set of created programs.

Conjunction.

$$\frac{S, E \vdash_{\theta}^{S'} G_1 \quad \wedge \quad S', E \vdash_{\gamma}^{S''} G_2\theta}{S, E \vdash_{\theta\gamma}^{S''} G_1, G_2} \tag{3.2}$$

To derive a non-empty conjunction, derive each conjunct in turn. The proof of the second conjunct proceeds with the answer substitution θ and the set of programs

S' computed by the proof of the first. S', which may or may not be the same as S, is the result of Web interactions from G_1's derivation. The result of these interactions are propagated to G_2 by starting the derivation of $G_2\theta$ with S'. S'' is the result of Web interactions from $G_2\theta$'s derivation, and since $G_2\theta$ started with S', S'' is the result of Web interactions during the derivation of the conjunction.

Atomic formula.

$$\frac{S, E \vdash_{\theta}^{S'} (H \text{ :- } G) \quad \wedge \quad \gamma = mgu(A, H\theta) \quad \wedge \quad S', E \vdash_{\delta}^{S''} G\theta\gamma}{S, E \vdash_{\theta\gamma\delta}^{S''} A} \tag{3.3}$$

Obtaining clauses from E can involve the creation of new programs due to LW-encapsulation (see rule (3.9)), and so, S is changed to S'. The proof of the body starts with the computed program set S' and returns the new set S'' and the answer substitution δ.

Obtaining clauses from a single program.

$$\frac{(A \text{ :- } G) \in P}{S, P \vdash_{\epsilon}^{S} (A \text{ :- } G)} \tag{3.4}$$

The answer substitution is ϵ, and there is no change to S.

3.3.3.2 Clauses from LW-compositions

The rules below determine how clauses are chosen from LW-compositions, and are defined based on the syntax of the program expressions. Except for LW-reduce, the rules are similar to those described in Chapter 2, with the key difference being the creation of LogicWeb programs.

LW-union.

$$\frac{S, E \vdash_{\theta}^{S'} (A \text{ :- } G)}{S, E{+}F \vdash_{\theta}^{S'} (A \text{ :- } G)} \tag{3.5}$$

$$\frac{S, F \vdash_{\theta}^{S'} (A \text{ :- } G)}{S, E{+}F \vdash_{\theta}^{S'} (A \text{ :- } G)} \tag{3.6}$$

A clause is chosen from a LW-union $E+F$ by choosing a clause from either E or F.

LW-intersection.

$$\frac{S, E \vdash_{\theta_1}^{S'} (H_1 \texttt{ :- } G_1) \quad \wedge \quad S', F \vdash_{\theta_2}^{S''} (H_2 \texttt{ :- } G_2) \quad \wedge \quad \gamma = mgu(H_1\theta_1, H_2\theta_2)}{S, E * F \vdash_{\theta_1\theta_2\gamma}^{S''} (H_1 \texttt{ :- } G_1, G_2)} \tag{3.7}$$

This rule utilises a left-to-right ordering in choosing clauses from the LW-intersection. A clause is first chosen from E returning S', and then, S' is used when selecting a clause from F ending up with S''.

LW-restriction.

$$\frac{S, E \vdash_{\theta}^{S'} (A \texttt{ :- } G) \quad \wedge \quad pred(A) \notin preds(P)}{S, E\,/\,P \vdash_{\theta}^{S'} (A \texttt{ :- } G)} \tag{3.8}$$

This rule is a straightforward extension of the rule for restriction in Chapter 2.

LW-encapsulation.

$$\frac{S, E \vdash_{\theta}^{S'} A}{S, @E \vdash_{\theta}^{S'} A \texttt{ :- true}} \tag{3.9}$$

LogicWeb programs can be created in the proof of A, i.e. a new program set S' is computed starting with S.

LW-reduce. The operational meaning of LW-reduce is given in terms of the above binary operators since it is a simplified notation for applying a binary operator between the program expressions in a list. LW-reduce is defined as:

$$(\oplus)\texttt{<>}[\,] = empty\ program\ with\ no\ clause \qquad \text{(eq. 3.1)}$$

$$(\oplus)\texttt{<>}[\,E\,] = E \qquad \text{(eq. 3.2)}$$

$$(\oplus)\texttt{<>}[\,E1\,,E2\,|\,Es\,] = (\oplus)\texttt{<>}[\,E1 \oplus E2\,|\,Es\,] \qquad \text{(eq. 3.3)}$$

where $\oplus$ is "+" or "*", and

$$\texttt{(/)<>(}E\texttt{,[])} = E \quad \text{(eq. 3.4)}$$

$$\texttt{(/)<>(}E\texttt{,[}P|Ps\texttt{])} = \texttt{(/)<>[(}E\ \texttt{/}\ P\texttt{)}|Ps\texttt{]} \quad \text{(eq. 3.5)}$$

The expression $\texttt{(/)<>}L$, where L is a non-empty list, is not allowed by the EBNF definition in Section 3.3.2 but is used here to specify the meaning of $\texttt{(/)<>(}E\texttt{,}L\texttt{)}$. $\texttt{(/)<>}L$ is defined by replacing $\oplus$ with "/" in (eq. 3.2) and (eq. 3.3). Note that the above definitions define left-to-right reductions. For associative operators $\oplus$, a right-to-left reduction can be similarly defined, but with no difference in the reduction result.

3.3.3.3 Context Switching

The rule for context switching will need to download the programs referred to in a program expression. To refer to the program identifiers within a program expression, the following function is used:

$$expids : ProgramExpressions \rightarrow \wp(LWProgramIDs)$$

$expids$ is defined recursively based on the syntax of program expressions:

$$expids(P) = \{P\}$$

$$expids(\texttt{(\#)}) = \{\}$$

$$expids(E_1 \texttt{ + } E_2) = expids(E_1) \cup expids(E_2)$$

$$expids(E_1 \texttt{ * } E_2) = expids(E_1) \cup expids(E_2)$$

$$expids(E \texttt{ / } P) = expids(E) \cup expids(P)$$

$$expids(\texttt{@}E) = expids(E)$$

$$expids(\texttt{(/)<>(}E\texttt{,}L_{(\mathcal{P})}\texttt{)}) = expids(E) \cup expids(L_{(\mathcal{P})})$$

$$expids(\texttt{(}\oplus\texttt{)<>}L_{(\mathcal{E})}) = expids(L_{(\mathcal{E})})$$

$$expids(L_{(\mathcal{E})}) = \bigcup_{E \in L_{(\mathcal{E})}} expids(E)$$

In the above, $\in$ is used to represent list membership, and a $L_{(\mathcal{P})}$ is a $L_{(\mathcal{E})}$ from the EBNF definition in Section 3.3.2.

The following function is needed to implement the context operator:

$$insertCC : ProgramExpressions \times ProgramExpressions \rightarrow ProgramExpressions$$

which substitutes every occurrence of the operator "`(#)`" in a program expression (the first argument) with the current context (the second argument):

$$\begin{aligned}
insertCC(\texttt{(\#)}, C) &= C \\
insertCC(P, C) &= P \\
insertCC(E_1 \texttt{ + } E_2, C) &= insertCC(E_1, C) \texttt{ + } insertCC(E_2, C) \\
insertCC(E_1 \texttt{ * } E_2, C) &= insertCC(E_1, C) \texttt{ * } insertCC(E_2, C) \\
insertCC(E \texttt{ / } P, C) &= insertCC(E, C) \texttt{ / } insertCC(P, C) \\
insertCC(\texttt{@}E, C) &= \texttt{@}insertCC(E, C) \\
insertCC(\texttt{(/)<>(}E\texttt{,}L_{(\mathcal{P})}\texttt{)}, C) &= \texttt{(/)<>(}insertCC(E, C)\texttt{,}insertCC(L_{(\mathcal{P})}, C)\texttt{)} \\
insertCC(\texttt{(}\oplus\texttt{)<>}L_{(\mathcal{E})}, C) &= \texttt{(}\oplus\texttt{)<>}insertCC(L_{(\mathcal{E})}, C) \\
insertCC(L_{(\mathcal{E})}, C) &= [insertCC(E, C) \mid E \in L_{(\mathcal{E})}]
\end{aligned}$$

Operator "#>" (context switching). The rule defining context switching is the following:

$$\frac{I \subseteq ids(S') \quad \wedge \quad S', F' \vdash_{\theta}^{S''} G}{S, E \vdash_{\theta}^{S''} F \texttt{ \#> } G} \tag{3.10}$$

where $F' = insertCC(F, E)$, $I = expids(F)$, and $S' = add_programs(S, I)$.

The rule states that the goal F `#>` G is provable in E starting with the program set S if the goal G is provable in F' starting with the updated program set S' which contains all the programs mentioned in I (and hence, in F).

The condition $I \subseteq ids(S')$ captures the semantics of goal evaluation in LW-compositions discussed in Section 3.3.1.2. It requires that all the programs mentioned in F be downloaded before goal evaluation can continue. Whenever a

program mentioned in F is not available, the LogicWeb goal fails since the condition is not satisfied.

3.3.3.4 An Example Top-down Derivation

Consider two LogicWeb programs with identifiers `M` and `N`. `M` contains the clauses:

```
p(X) :- q(X).
r :- b.
```

`N` contains:

```
r :- w.
q(a) :- true.
```

Figure 3.5 shows an example of how the rules are used to prove the following goal posed to a program `P`, which is assumed to be already downloaded (and therefore contained in the starting program set).

```
?- (M + N)#>p(a).
```

First, rule (3.10) is used to switch from program `P` to the LW-union. Then, assuming that `M` and `N` are successfully created, rule (3.3) is used resulting in two branches. In the left branch, rules (3.5) and (3.4) are used to retrieve a clause from the LW-union. The head of the clause unifies with the goal `p(a)` with the substitution of `X` by `a`, denoted by $\{\texttt{X/a}\}$. In the right branch, the subgoal `q(a)`, resulting from applying $\{\texttt{X/a}\}$ to `q(X)`, is proven with the clause from `N` using the rules (3.3), (3.6), (3.4), and (3.1). The result of the proof is the computed answer substitution $\{\texttt{X/a}\}$ and the new set of programs $\{\texttt{P, M, N}\}$.

Figure 3.5 A derivation of the LogicWeb goal `(M + N)#>p(a)` in a program `P`.

$$\dfrac{\dfrac{\dfrac{\texttt{p(X) :- q(X)} \in \texttt{M}}{\{\texttt{P,M,N}\},\texttt{M} \vdash_{\epsilon}^{\{\texttt{P,M,N}\}} \texttt{p(X) :- q(X)}}(3.4)}{\{\texttt{P,M,N}\},(\texttt{M + N}) \vdash_{\epsilon}^{\{\texttt{P,M,N}\}} \texttt{p(X) :- q(X)}}(3.5) \quad \boxed{\text{(see next part of Figure 3.5)}}}{\dfrac{\{\texttt{P,M,N}\},(\texttt{M + N}) \vdash_{\{\texttt{X/a}\}}^{\{\texttt{P,M,N}\}} \texttt{p(a)}}{\{\texttt{P}\},\texttt{P} \vdash_{\{\texttt{X/a}\}}^{\{\texttt{P,M,N}\}} \texttt{(M + N) \#> p(a)}}(3.10)}(3.3)$$

Figure 3.5 (Continued)

$$\dfrac{\dfrac{\dfrac{\texttt{q(a) :- true} \in \texttt{N}}{\{\texttt{P,M,N}\},\texttt{N} \vdash_{\epsilon}^{\{\texttt{P,M,N}\}} \texttt{q(a) :- true}}(3.4)}{\{\texttt{P,M,N}\},(\texttt{M + N}) \vdash_{\epsilon}^{\{\texttt{P,M,N}\}} \texttt{q(a) :- true}}(3.6) \quad \dfrac{}{\{\texttt{P,M,N}\},(\texttt{M + N}) \vdash_{\epsilon}^{\{\texttt{P,M,N}\}} \texttt{true}}(3.1)}{\{\texttt{P,M,N}\},(\texttt{M + N}) \vdash_{\epsilon}^{\{\texttt{P,M,N}\}} \texttt{q(a)}}(3.3)$$

3.3.4 Relationship of Operational Semantics to Declarative Semantics

As shown in [40], the language given in Chapter 2 has a fixpoint semantics which is sound and complete with respect to its operational semantics. This result can be applied to a class of LogicWeb programs called *restricted LogicWeb programs*. A restricted LogicWeb program is a pure LogicWeb program which does not use the context operator, and only uses ground LogicWeb program identifiers in LogicWeb goals. Such a program can be syntactically translated into a program in the language of Chapter 2 by rewriting expressions which use LW-reduce, replacing each occurrence of "`#>`" by "in", and replacing the LW-composition operators with their counterparts from Section 2.2.3.1. Thus, a fixpoint semantics can be defined for a restricted LogicWeb program which corresponds to the fixpoint semantics of that program's translation. Also, if a goal evaluation using the operational semantics of the previous section succeeds, then a corresponding goal derivation in the program's translation can be constructed using the rules from Section 2.2.3.3. Taken together, these two points imply the soundness of restricted LogicWeb programs.

On the other hand, logical completeness[2] using the above operational seman-

[2]Completeness in theory (i.e., assuming time and memory resources are unlimited) is meant here, since completeness in practice is never achievable (even with no Web interactions) due to limited resources (e.g., time or memory) or sometimes due to unfairness in Prolog's depth first

tics is not generally attainable with LogicWeb programs. The reason being that a LogicWeb program has an open or interactive nature which contrasts with a closed set of logical axioms whose declarative meaning is dependent only on the form of its formulae. The success or failure of a goal in a LogicWeb program depends not only on its axioms, but also on the result of the oracle function, which is unpredictable. Goal evaluation can succeed only if the oracle function behaves favourably towards the computation. Hence, it is possible that a goal is true with respect to the declarative semantics of a set of programs (some residing locally on the client host and others on remote hosts), but is not provable (on the client) using the above operational semantics because of the oracle function's results. Since it is generally impossible to adequately specify (e.g., using axioms) the result of a HTTP request at a given time, the semantics of LogicWeb programs cannot be fully declarative.[3]

However, using the completeness result in [40], and given a history of favourable Web interactions, completeness can be stated for restricted LogicWeb programs: if a goal is true with respect to the fixpoint semantics of a set of restricted LogicWeb programs, then the goal is provable when the set of restricted LogicWeb programs required for evaluating the goal can be successfully downloaded.

3.4 Building Applications Using LogicWeb Programs

Logic programming applications can be constructed on the Web based on LogicWeb. Such an application, which is called a *LogicWeb application*, comprises a set of LogicWeb programs of which one is singled out (by the programmer) as the *main program*. Each application has an *identity* which is the identity of the

searching behaviour.

[3] To add to this difficulty, since a download takes time, and the state of the Web can change during the download, we would have to "freeze" the Web for the duration of the download.

main program as given by its program identifier, and a *user interface* with which the user interacts. Users interact with a LogicWeb application (i.e., with the main program of the application) through the Web's interface mechanisms of hypertext links and HTML forms. Rules in the main program define the forms interface and the mapping of user actions to the invocation of particular predicates in the main program. A page without any LogicWeb rules can be viewed as a rudimentary form of LogicWeb application with the ordinary behaviour of Web links (described in Section 2.1.2.3).

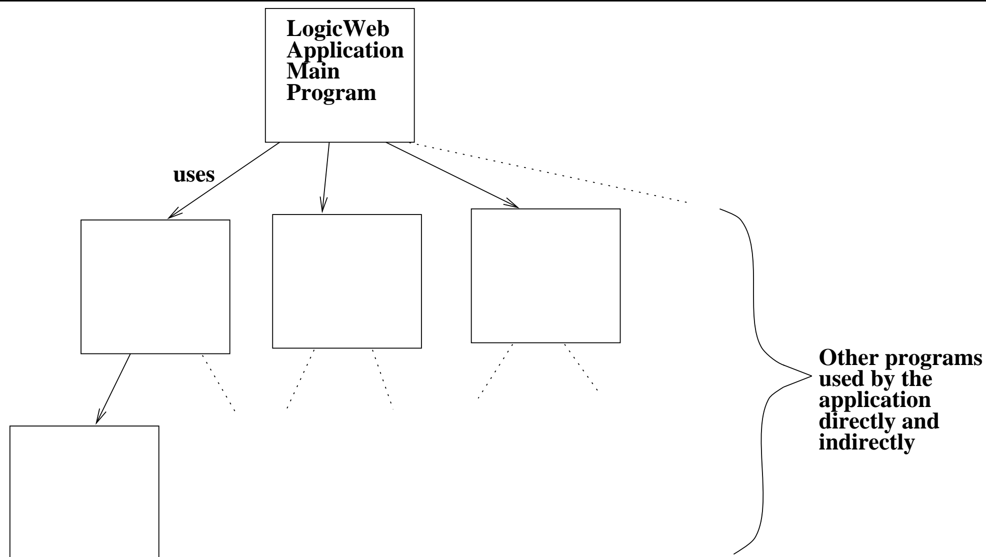
A LogicWeb application is downloaded by downloading its main program. The main program, in turn, may download and *use* other programs (which may or may not be members of the application) directly or indirectly. A program M *directly uses* P if M invokes a goal in P, and *indirectly uses* P if M invokes a goal in Q which directly or indirectly uses P. The set of programs used directly or indirectly by an application is dynamically determined and depends on the particular goals being evaluated. These programs are downloaded and added to the monotonically increasing set of created LogicWeb programs. Figure 3.6 depicts a LogicWeb application consisting of only one program (i.e., the main program) and the programs used directly and indirectly by it.

The next chapter describes in detail how a LogicWeb application is constructed. Chapters 5, 6, and 7 present examples of LogicWeb applications.

3.5 Summary

This chapter has presented LogicWeb which represents the Web augmented with rules as a query-able collection of inter-related logic programs. The translation of pages with rules to LogicWeb programs has been detailed. LogicWeb programs are used as the building blocks of applications on the Web. A user queries LogicWeb programs through the Web's interface mechanisms (how this is done is shown in the next chapter). This means that LogicWeb adds to the Web a new

Figure 3.6 A LogicWeb application with the programs it uses directly and indirectly.



form of programmable behaviour which is rule-based. Queries to LogicWeb programs are processed by rules which reason with the Web's hypertext structure and manipulate other LogicWeb programs using compositional logic programming ideas. A LogicWeb goal performs two tasks: download required LogicWeb programs and (after programs are downloaded) continue the proof search. If either tasks fails, then the LogicWeb goal fails. Coding techniques have been presented to differentiate goal failures due to download failures from evaluation failures.

The execution behaviour of LogicWeb programs have been captured precisely in an operational semantics using Plotkin-style rules and an oracle function. The operational semantics forms the basis for a language implementation. Although an abstract concept, the oracle function has a direct implementation as a routine which sends HTTP requests to servers and receives their responses. An implementation of LogicWeb is presented in the next chapter.

Chapter 4

Implementing LogicWeb

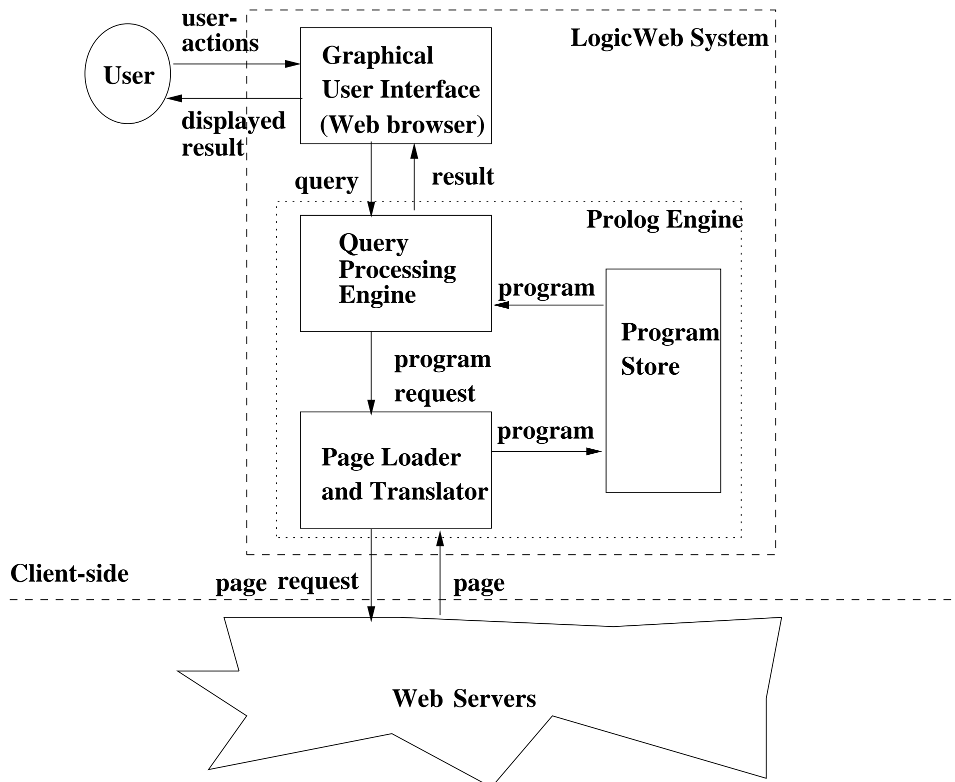
This chapter is concerned with the implementation of LogicWeb. As mentioned in the previous chapter, a user accesses and queries LogicWeb programs through the Web's interface mechanisms. The mechanisms are link selection and HTML fill-in forms on the page. A system implementing LogicWeb must support the invocation of LogicWeb rules via these two mechanisms. The implementation of the LogicWeb language must take care of the details of Web page retrieval, parsing, and caching, allowing the programmer to focus on the meanings of program compositions and the application problem.

Section 4.1 presents an architecture for such a system which is called the *LogicWeb system*. The system has been built by integrating an off-the-shelf Web browser with a public domain Prolog system. Section 4.2 presents an overview of the implementation using a simple mini picture database search application for demonstration purposes. Section 4.3 describes the behaviour of the system in response to user actions. Responses to user actions are determined by the system's Prolog component (called the *Prolog engine*) with the help of rules from relevant LogicWeb programs. Section 4.4 describes the Prolog engine in detail.

4.1 The LogicWeb System Architecture

The architecture of the LogicWeb system (its components and data-flows between them) is shown in Figure 4.1.

Figure 4.1 Architecture of the LogicWeb system.



The main components of the system are:

- a graphical user interface (GUI);
- an engine for processing user queries;
- a page loader and (page to program) translator; and
- a store of LogicWeb programs.

The Prolog engine is formed by the three components: the query processing engine, the program store, and the page loader and translator.

Using this system, a LogicWeb program which is downloaded and displayed on the GUI (or a Web browser) can be queried. The system converts user actions into queries to the program, and computes the results of the queries with respect to the program in the query processing engine. Not all user actions on the browser are processed by the Prolog engine. The particular user actions which involve the Prolog engine are specified in Figure 4.2 and detailed in Sections 4.3 and 4.4. Processing of the query may result in other pages being downloaded and translated into LogicWeb programs. Created LogicWeb programs are stored in the program store and are utilised in the evaluation of goals. When query processing ends, the system formats the results and shows it to the user.

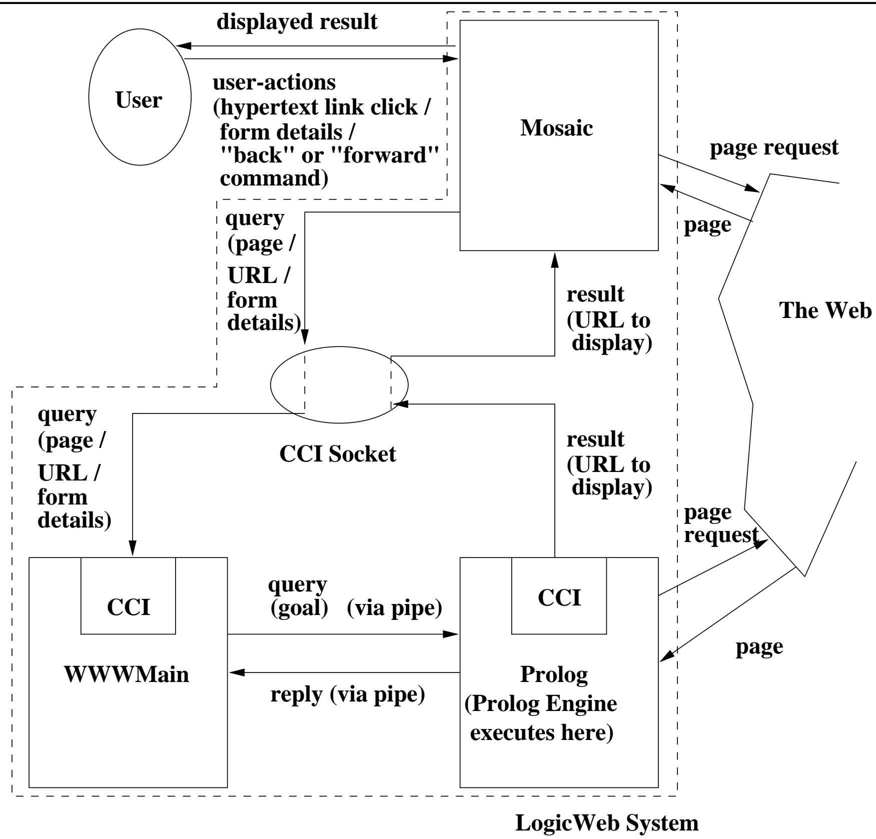
Recall from Section 3.4 that a LogicWeb application consists of a set of LogicWeb programs of which one is distinguished as the main program. A LogicWeb application is used by downloading and querying its main program in the way described above. A query evaluated in the main program causes other components of the application to be retrieved.

4.2 Implementation Overview

The LogicWeb system has been implemented using the Common Client Interface (CCI) (version 1.1) library[1] of NCSA Mosaic (version 2.6b1) [154]. The CCI allows the capabilities of Mosaic to be utilised by external applications without modifying Mosaic internally. The CCI library is a set of C routines implementing a protocol for external applications to communicate with Mosaic. Using the CCI protocol, an external application may request Mosaic to perform various functions such as informing the external application when anchors (or hypertext links) are

[1] CCI specification found at
`http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-spec.html`.

Figure 4.2 An overview of the main components and data-flows (arrows) between them.



selected and sending data for specified MIME-types to the external application instead of rendering it. Figure 4.2 presents an overview of the implementation which consists of the following components:

- **Mosaic.** The NCSA Mosaic browser allows the user to browse Web pages and serves as the GUI for the LogicWeb system. As mentioned in Section 3.4, the main program of a LogicWeb application defines the visible interface to the application. For example, Figure 4.3 shows a form which is the interface to a mini picture database search application. The browser is also used to display pages generated by applications. To display a page on the browser, the page is first saved into a temporary file. Then, the file's URL is passed to the browser which utilises it to fetch and display the page. The temporary file is used since the CCI library does not have the capability to send the page's contents directly to Mosaic. Figure 4.4 presents an example of a page created as the result of a search with the keyword "marsupial" on the mini picture database.

- **Prolog.** This is a running Prolog process where the Prolog engine, which handles user queries to LogicWeb programs, is executed. The query processing engine consists of the LogicWeb program interpreter and predicates to translate user actions into the evaluation of specific goals in LogicWeb programs. Downloaded LogicWeb programs are stored inside the Prolog system (as facts in the SWI-Prolog database). The collection of stored programs constitute the program store. The page loader and translator consists of predicates which download and parse HTML documents to extract the clauses making up LogicWeb programs.

  All three components are implemented in SWI-Prolog 2.1.14 [209] (about 2300 lines of Prolog) extended with utilities for parsing and communicating with the Web and Mosaic. Utilities which have speed critical features (such as string matching) and which communicate with the Web and Mo-

Figure 4.3 The interface of a simple LogicWeb application.

saic are coded as C functions with Prolog interfaces (about 1200 lines of C). The predicate which displays information on Mosaic uses the CCI library.

- **WWWMain.** This C program converts CCI outputs from Mosaic, such as the notification of a hypertext link selection or information entered into forms, into a suitable format for the Prolog engine. WWWMain starts up Prolog and Mosaic informing it of what CCI outputs to send, and sets up the communication links. WWWMain is about 400 lines of C.

The LogicWeb system consists of the above three communicating processes.

Figure 4.4 The result of a query with keyword "marsupial".

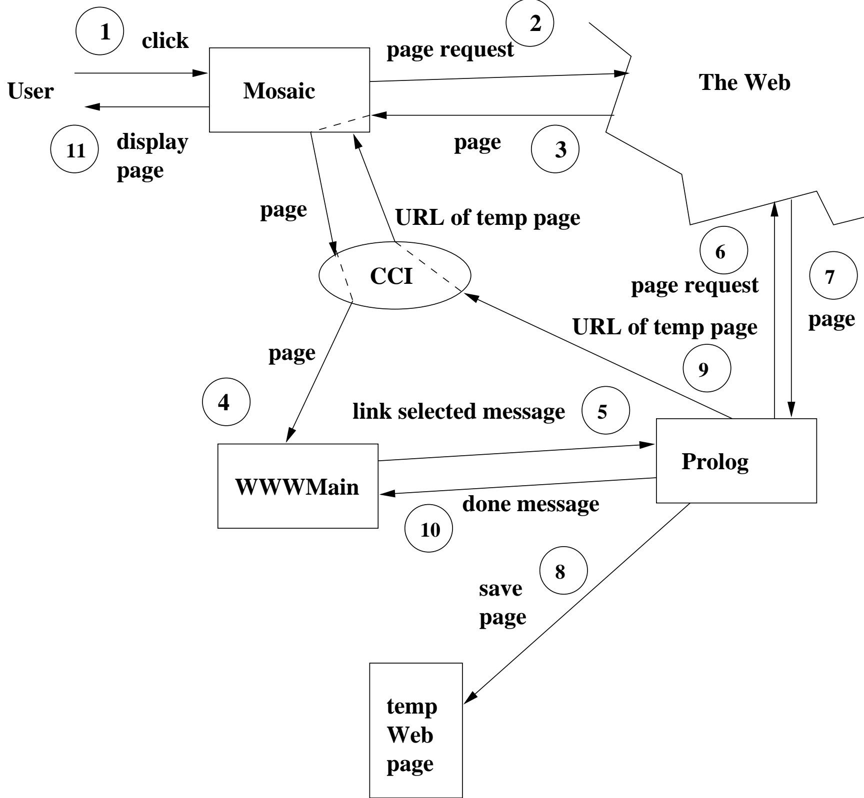
Both WWWMain and Prolog communicate with Mosaic via CCI (using socket connections), while the communication between WWWMain and Prolog utilises Unix pipes. WWWMain and Prolog share a socket connection to Mosaic. Note from Figure 4.2 (but not shown in Figure 4.1) that Mosaic may download pages directly from the Web bypassing the Prolog engine. The data-flows of the system are discussed further in the next section.

4.3 System Behaviour

User inputs are channelled through Mosaic to the Prolog engine which computes the desired responses. Then, the responses are returned from the Prolog engine to Mosaic. Figure 4.5 shows the sequence of steps taken when a user clicks on a hypertext link and the system displays the page linked to.

When the user clicks on a link (step 1), Mosaic gets the page from the Web (steps 2 and 3). The page's data is not displayed but passed through the CCI to WWWMain (step 4). On receiving the page's data, WWWMain notices that a link has been selected on Mosaic, and sends a "link selected" message containing the selected URL via a pipe to the Prolog engine (step 5). The data from Mosaic is insufficient for generating a LogicWeb program since it does not include the page's meta-information. At this point, the Prolog engine could issue a HEAD request with the page's URL to retrieve the meta-information. However, this meta-information would not correspond to the page's data obtained earlier by Mosaic. For instance, the time of this HEAD request and the last modified date (if the page happened to be modified) would be different from those for the page retrieved by Mosaic. Also, the page received from Mosaic is not always the requested page, but may only be a page indicating that the requested page has moved. Hence, the Prolog engine re-issues the page request (a GET request) in order to obtain the meta-information together with the page's contents (steps 6 and 7). Once the page's meta-information and contents are obtained,

Figure 4.5 The LogicWeb system and the steps followed after a user clicks on a link.



they are converted into a LogicWeb program which is stored inside the Prolog system (i.e., added to the program store) for use by the LogicWeb program interpreter. No downloading is carried out by the Prolog engine if the program corresponding to the page (identified by its URL) already exists in the program store. The engine may execute a goal with respect to the new (or existing) program or simply display the page with Mosaic. To display the page, its HTML contents are augmented with a form for entering queries and stored in a temporary file (step 8), and the URL of the temporary file is sent to Mosaic via the CCI (step 9). A "done" message is then transmitted to WWWMain (step 10) signalling that the Prolog engine is ready for further work. Mosaic uses the URL it receives via the CCI to load and display the temporary page (step 11).

Figure 4.6 illustrates the second kind of user interaction with the system: the processing of a query entered via a form acting as the interface to a LogicWeb application.

The query is input via a form (step 1) (such as the query in Figure 4.3), and the goal is extracted by a CGI script (step 2). The goal is passed to the CCI (step 3) and onto WWWMain (step 4) and finally to the Prolog engine (step 5). If the goal uses a program that has already been downloaded (such as the current page), then the meta-interpreter immediately evaluates the goal and stores the answer in a temporary Web page (step 8). The URL of this page is sent to Mosaic via the CCI (step 9) and the page is displayed by Mosaic (step 11) (such as the result in Figure 4.4). At the same time, the Prolog engine sends a "done" message to WWWMain to signal its readiness for further work (step 10).

A slightly more complicated sequence occurs if the LogicWeb goal requires a program that is not presently on the client-side. In that case, the corresponding page is obtained from the Web (steps 6 and 7), and the program is extracted before the goal is evaluated.

The third form of user input which is channelled through Mosaic to the Prolog engine is a notification whenever the user issues a "back" or "forward" com-

Figure 4.6 The LogicWeb system and the steps followed after a user enters a query.

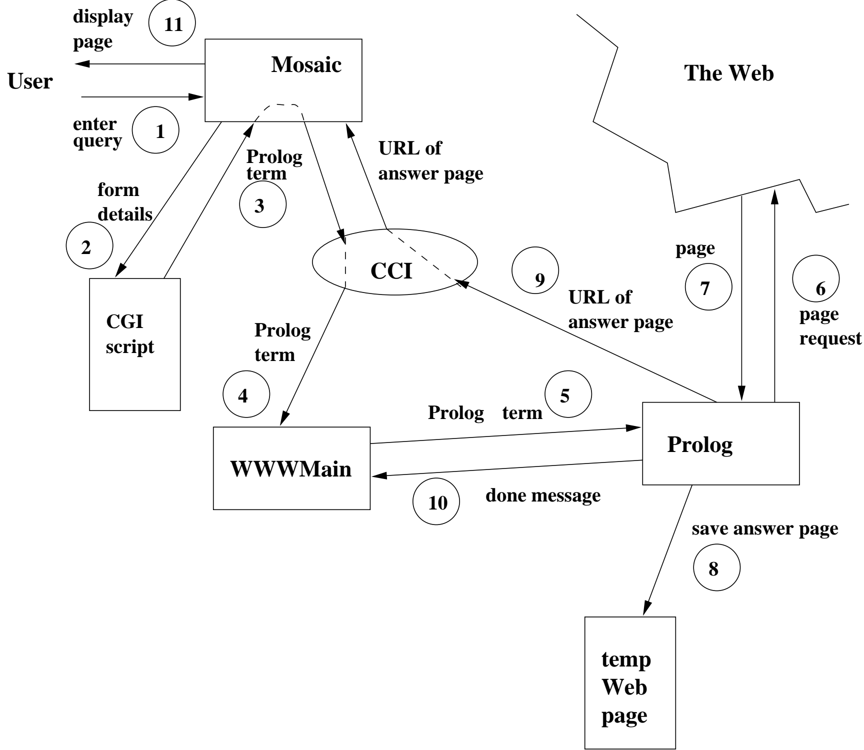
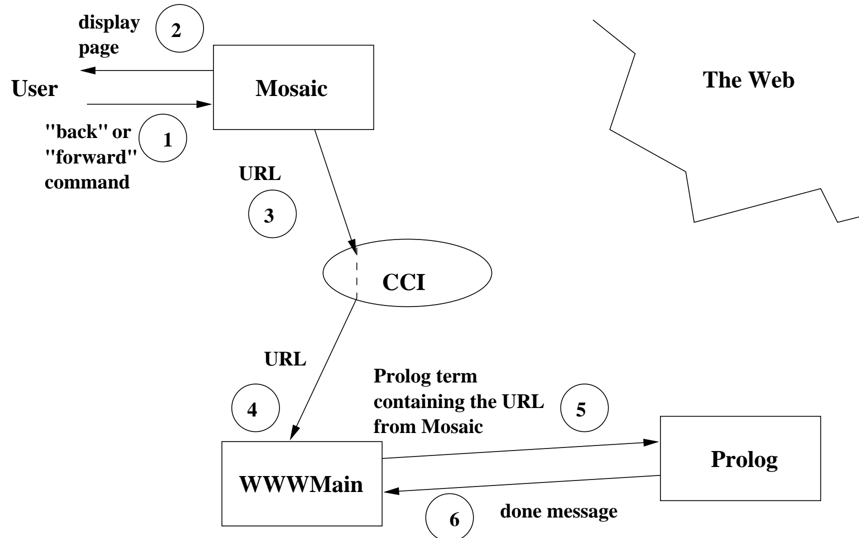


Figure 4.7 The LogicWeb system and the steps followed after a user issues a "back" or "forward" command in Mosaic.



mand in Mosaic. These two commands allow navigation within a list (stored in Mosaic) of pages representing the history of displayed pages. The "back" command causes Mosaic to display the page which was previously displayed, and the "forward" command displays the page after the current page in the history list. The sequence of steps for this form of input is simpler than the previous two and is shown in Figure 4.7. When the user invokes a "back" or "forward" command in Mosaic (step 1), Mosaic displays the previous or next page (in its history list) (step 2) and passes the URL of the page to be displayed through the CCI (step 3) to WWWMain (step 4). WWWMain forwards the URL as a goal to the Prolog engine (step 5) which, on receiving the goal, processes it (as shown in the next section) and signals WWWMain that it is ready for further work (step 6).

4.4 The Prolog Engine

As mentioned in the previous section, the Prolog engine determines the responses to user inputs received via WWWMain. In this section, we take a detailed look into how the user inputs are processed by LogicWeb applications and discuss the creation and storage of LogicWeb programs. Prolog code is given defining important parts of the engine. This code is a simplification of that in the implementation leaving out details of error checking, error message display, optimisations, and communication with WWWMain, and function as specifications.

4.4.1 Mapping User Actions to Goals

The top-most level of the Prolog engine is a loop where each cycle accepts inputs from WWWMain, invokes specific goals using the inputs, and signals to WWWMain that computation is done and the engine is ready for more input. The following recursive clause specifies this loop:

```
top_level :-
  accept_inputs(UserInput),
  ( execute_goal(UserInput)
  ; true
  ),
  signal_done,
  top_level.
```

`accept_inputs/1` reads the user's inputs from WWWMain via a pipe, and blocks while waiting for input. Note from the definition of `top_level/0` that the system continues execution if `execute_goal/1` fails (by the disjunction with `true`). `signal_done/0` sends the done message to WWWMain as shown in Figures 4.5, 4.6, and 4.7.

As depicted in Figures 4.5, 4.6, and 4.7, WWWMain forwards CCI outputs from Mosaic to the Prolog engine as Prolog terms. The input from WWWMain is one of the following:

1. the user's inputs to the HTML form acting as the interface of a LogicWeb application;

2. a notification to the Prolog engine that the user has selected a link or has issued an "open URL" command in Mosaic; or

3. a notification to the Prolog engine that the user has invoked the "back" or "forward" navigation command in Mosaic.

Corresponding to these three kinds of inputs, `UserInput` is instantiated to one of the following terms:

1. `form_input(`*`AttributeValueList`*`, `*`QueryHandlerURL`*`)`: *`AttributeValueList`* is a list of terms of the form `p(`*`Attribute`*`, `*`Value`*`)`, each corresponding to the name and value of a query attribute defined in the HTML form (as described in Section 2.1.2.2). *`QueryHandlerURL`* is used to identify the LogicWeb program containing the predicate `do_query/2` which will be invoked with the *`AttributeValueList`* as one of its arguments. *`QueryHandlerURL`* is specified as a hidden query attribute in the form.

2. `link_selection(`*`URL`*`)`: *`URL`* is the URL of a link selected by the user on Mosaic, or the URL opened via an "open URL" command.

3. `navigation(`*`NewURL`*`)`: *`NewURL`* is the URL of the page to be displayed in response to a "back" or "forward" command.

As pointed out in Section 2.1.2.2, HTML forms are used for accepting inputs to CGI scripts on the Web. But HTML forms are also used as visible interfaces to LogicWeb applications. Forms used as application interfaces are generated by the system using a *reserved predicate* called `interface/1` which is included within a page. Reserved predicates are defined by the page's author but are recognised by the system as special purpose predicates. Reserved predicates

serve as "hooks" into the system which an application uses. Below, the use of `interface/1` is first discussed. Then, two other reserved predicates are described. The system supports only these three reserved predicates.

`interface/1` contains a list of terms defining a HTML form. The possible forms of these terms are given in Appendix B (Section B.1). When a page is downloaded, the system uses the contents of `interface/1` on the page to generate a form with a specific system value for its `ACTION` attribute. This system assigned `ACTION` value distinguishes forms used as application interfaces. This is required because inputs to a LogicWeb application interface are processed differently from inputs to ordinary forms. Form inputs to a LogicWeb application are intercepted by the LogicWeb system's CGI script (as shown in Figure 4.6) and used to invoke a goal in the application's main program, whereas inputs to ordinary forms are used for invoking scripts on remote server hosts. Form inputs to a LogicWeb application are sent by WWWMain to the Prolog engine in a `form_input/1` term whereas the inputs to an ordinary form (using the GET method) are passed in as a `link_selection/1` term. For instance, the user may fill in a form on Mosaic as a query to the AltaVista search engine. WWWMain sends the URL encoding the query to the Prolog engine in a `link_selection/1` term. Due to a limitation of the CCI, form inputs submitted via the browser using the POST method are not redirected to WWWMain, and hence, are not handled by the Prolog engine. However, LogicWeb rules can retrieve programs with the POST method using an identifier of the form `lw(post(`$Data$`),` URL`)`.

Figure 4.8 is a HTML page which shows how `interface/1` is used in the mini picture database application introduced earlier. `interface/1` describes the form for interacting with the application. When the page is downloaded, it is translated by the system into a LogicWeb program in the way described in Chapter 3. Then, the contents of `interface/1` is used to generate a form. This form and the non-code portion of the page is then rendered on the browser as shown in Figure 4.3.

Figure 4.8 A simple LogicWeb application.

```
<HTML>
<HEAD>
<TITLE>Mini Animal Picture Database</TITLE>
</HEAD>

<BODY>
<H1>Welcome to my searchable mini animal picture database!</H1>
<!--
<LW_CODE>
% a form interface to the database
interface([textnl("<P>Enter keyword to identify required picture:"),
           input("text", "keyword", "", "", "20", "20")
          ]).

% a collection of images
pic("koala.jpg").
pic("reindeer.jpg").
pic("snake.jpg").
pic("eagle.jpg").

% generalisations
isa("eagle", "bird").
isa("reindeer", "deer").
isa("koala", "marsupial").
isa("python", "snake").

% process user queries
do_query([p("keyword", Keyword)], _) :-
  pic(Name),
  ( contains(Name, Keyword)
  ; isa(Instance, Keyword), contains(Name, Instance)
  ),
  display_picture(Name).

display_picture(Name) :-
  build_head("Result Picture", Head),
  flatten(["<IMG SRC=""",Name,""">"], Data),
  build_body("Here is the picture", Data, Body),
  build_whole(Head, Body, ResultPage),
  string_to_list(Results, ResultPage),
  display_page("http://www.cs.mu.oz.au/~swloke/", [data(Results)]).
</LW_CODE>
-->
</BODY>
</HTML>
```

User's inputs to the generated form or link activations are passed by `execute_goal/1` to one of two other reserved predicates in a LogicWeb program. These two predicates are:

- `do_query/2`. This predicate receives user queries entered via the application's interface (as defined by `interface/1`) and decides how the queries are processed. Its first argument is the form inputs returned by `accept_inputs/1` and is instantiated by the system. The second argument is instantiated with a term representing the goal invoked in the body of `do_query/2`. This goal is printed for debugging purposes.

  Figure 4.8 presents an example of the use of `do_query/2` which processes the input keyword, matching it with the name of an image defined in `pic/1`, or with its generalisation as defined by `isa/2`. Successful processing of the query results in the required picture being retrieved and displayed. `contains/2` is a built-in predicate which determines if the second argument is a substring of the first (see Appendix B). `build_head/2`, `build_body/3`, and `build_whole/3` are used for constructing a HTML document and are described in Appendix B. `display_page/2` is a built-in predicate for LogicWeb programs which displays a page on the browser. Its first argument specifies the URL to be displayed and its second argument, the HTML source. Appendix B describes this predicate in detail. When the user submits the keyword "marsupial" via the form, the following goal is invoked to construct the page with the picture of a koala shown in Figure 4.4:

  ```
  ?- do_query([p("keyword", "marsupial")], _).
  ```

- `link_action/1`. This predicate determines what action to take when a link is selected. For example, the following rule fetches a page and displays it:

```
link_action(URL) :-
  lw(get, URL)#>h_text(HTMLSrc),
  display_page(URL, [data(HTMLSrc)]).
```

The system supports the three reserved predicates mentioned above, but an application need not define all of the reserved predicates. The system uses the predicates in a specific order and defines default behaviours catering for the absence of each of the predicates. This is implemented in `execute_goal/1` whose basic functionality is specified in Program 4.1.

Program 4.1 The predicate which translates user inputs into goals.

```
execute_goal(form_input(AVList, QueryHandlerURL)) :-
  demo(empty, lw(get, QueryHandlerURL)#>do_query(AVList, _Goal)).
execute_goal(link_selection(SelectedURL)) :-
  handle_link(SelectedURL).
execute_goal(navigation(NewURL)) :-
  update_current_page(NewURL).

% handle a link
handle_link(SelectedURL) :-
  current_page(CurrentPageURL),
  demo(empty, lw(get, CurrentPageURL)#>link_action(SelectedURL)).
handle_link(SelectedURL) :-
  display_page_interface(SelectedURL).
handle_link(SelectedURL) :-
  demo(empty, lw(get, SelectedURL)#>h_text(Src)),
  display_page(SelectedURL, [data(Src)]).

% update the current URL
update_current_page(NewURL) :-
  retract(current_page(_OldURL)),
  assert(current_page(NewURL)).
```

In the first clause of `execute_goal/1`, on receiving a `form_input/2` term from WWWMain, the interpreter `demo/2` is called to evaluate the goal `do_query/2` in the program `lw(get, QueryHandlerURL)`. Goals are in-

voked in the LogicWeb program interpreter (described in Section 4.4.2). The context switching operator "`#>`" is used to ensure that the program is created before the goal is evaluated. This is necessary since the program handling the query `lw(get, QueryHandlerURL)` may be different from the current page (i.e., the page currently displayed on Mosaic and accepting the user's inputs).

In the second clause of `execute_goal/1`, on receiving a `link_selection/1` term, `handle_link/1` is called. `handle_link/1` consists of three clauses which implements how the system responds to a link selection. The selected link resides on the current page. For determining the response, priority is given to the predicates on the current page over those in the page referred to by the selected link. This behaviour is implemented as follows.

The first clause of `handle_link/1` retrieves the URL of the current program (i.e., the program corresponding to the current page) and invokes `link_action/1` in the current program. `current_page/1` stores the URL of the current page, and is updated in response to user inputs which change the current page, or when a page is displayed as a result of goal evaluations. If the `link_action/1` goal fails, control passes from the current program to the new program identified by `SelectedURL`.

The next two clauses of `handle_link/1` attempt to start a new application. The second clause of `handle_link/1` displays the interface and the (non-code) HTML text of the page using `display_page_interface/1` (whose definition is not given in Program 4.1) which invokes `interface/1`. On succeeding, `display_page_interface/1` updates `current_page/1` to store the value of `SelectedURL`. If the `interface/1` goal fails, then `display_page_interface/1` fails and the system attempts to display the new page on the browser (in the third clause of `handle_link/1`). `display_page/2` on succeeding updates `current_page/1` to store the value of `SelectedURL`. In effect, if none of the reserved predicates are present or if their invocations fail, the system defaults to the ordinary behaviour of links where a link selection

simply displays the new page.

In the third clause of `execute_goal/1`, `current_page/1` is updated with the new URL contained in `navigation/1`. This ensures that, as the user navigates "back" and "forward" in Mosaic, `current_page/1` is kept up to date.

4.4.2 The LogicWeb Program Interpreter

The design of the interpreter for LogicWeb programs is based on the operational semantics developed in Section 3.3.3. Program 4.2 shows the interpreter for pure LogicWeb programs.

The first clause of `demo/2` implements context switching as described by rule (3.10). The predicate `establish_context/3` takes a program expression `F`, the current context (a program expression) `E`, and establishes a new context `F1` (also a program expression) in the following sense:

1. `establish_context/3` implements $expids$ and $add_programs$ (both defined in Chapter 3) by calling the predicate `download/2` (as seen in the second last clause of `establish_context/3`) on each LogicWeb program mentioned in `F`. `download/2` implements the oracle function (see Definition 1 in Section 3.3.3) augmented with a test to determine if a program exists. The following predicate specifies `download/2`:

```
download(Type, URL) :-
  created(Type, URL), !.              % program already exists
download(Type, URL) :-                % program does not exist
  retrieve(Type, URL, Contents),      % retrieve from the Web
  create_program(Type, URL, Contents). % create the program
```

   `create_program/3` creates a LogicWeb program from `Contents` and is explained further in the next subsection. `download/2` does not return the created program in its argument since the program is asserted into the Prolog database. `download/2` fails if a LogicWeb program can not

Program 4.2 The interpreter for pure LogicWeb programs.

```
% demo/2 with LogicWeb goal
demo(E, F#>G) :- establish_context(F, E, F1), demo(F1, G).
demo(_E, true).
demo(E, (A,B)) :- demo(E, A), demo(E, B).
demo(E, A) :- E::(A :- B), demo(E, B).

% establish a context
establish_context(E + F, C, E1 + F1) :-
  establish_context(E, C, E1), establish_context(F, C, F1).
establish_context(E * F, C, E1 * F1) :-
  establish_context(E, C, E1), establish_context(F, C, F1).
establish_context(E / P, C, E1 / P1) :-
  establish_context(E, C, E1), establish_context(P, C, P1).
establish_context(@E, C, @E1) :-
  establish_context(E, C, E1).
establish_context((/)<>(E, L), C, (/)<>(E1, L1)) :-
  establish_context(E, C, E1), establish_contextL(L, C, L1).
establish_context(Op<>L, C, Op<>L1) :-
  establish_contextL(L, C, L1).
establish_context(lw(T, U), _C, lw(T, U)):-
  download(T, U).
establish_context((#), C, C).

establish_contextL([], _C, []).
establish_contextL([E|Es], C, [E1|Es1]) :-
  establish_context(E, C, E1), establish_contextL(Es, C, Es1).

% definition of ::/2
(E + _F)::(A :- B) :- E::(A :- B).
(_E + F)::(A :- B) :- F::(A :- B).

(E * F)::(A :- (B,C)) :- E::(A :- B), F::(A :- C).

(E / P)::(A :- B) :- E::(A :- B), not defined(A, P).

(@E)::(A :- true) :- demo(E, A).

((/)<>(E, []))::(A :- B) :- E::(A :- B).
((/)<>(E, [P|Ps]))::(A :- B) :- ((/)<>[(E / P)|Ps])::(A :- B).

(_Op<>[E])::(A :- B) :- E::(A :- B).
(Op<>[E1,E2|Es])::(A :- B) :-
  C =.. [Op, E1, E2], (Op<>[C|Es])::(A :- B).

defined(A, P) :-
  functor(A, Functor, Arity), functor(H, Functor, Arity),
  P::(H :- _B).
```

be retrieved (the situation where the oracle function returns $\perp$). The failure of `download/2` causes `establish_context/3` to fail. This implements the condition in rule (3.10) that all LogicWeb programs in `F` must be created before goal evaluation proceeds. In contrast to the definition of $add_programs(S, I)$ which *simultaneously* attempts to download all programs mentioned in I but not in S (i.e., no ordering is specified for downloading programs), the implementation imposes an ordering on downloading the programs mentioned in `F` which follows Prolog's ordering in the evaluation of `establish_context/3`. With the ordering on downloads, as soon as one call to `download/2` fails, no further downloads will be attempted (since `establish_context/3` fails).

2. Each occurrence of "`(#)`" in `F` is replaced by `E` (in the last clause of `establish_context/3`). This implements $insertCC$ used in rule (3.10).

The next three clauses constitute the vanilla meta-interpreter. The clauses of LogicWeb programs are stored in `::/2` facts, but rules are defined for retrieving clauses from LW-compositions of programs. The five rules of the predicate `::/2` defining LW-union, LW-intersection, LW-restriction, and LW-encapsulation are similar to `::/2` defining union, intersection, restriction, and encapsulation in Section 2.2.3.4. The other rules determine how to retrieve clauses from LW-reductions, and are defined using the equations (eq. 3.1) to (eq. 3.5) in Section 3.3.3. For instance, (eq. 3.4) states that `(/)<>(E, [])` is equal to `E`. Hence, retrieving a clause from the expression `(/)<>(E, [])` is the same as retrieving a clause from `E`, and this is represented by the clause:

```
((/)<>(E, []))::(A :- B) :- E::(A :- B).
```

in Program 4.2.

Note that `::/2` and `establish_context/1` are defined to allow nested expressions, i.e. terms such as

```
(+)<>[(*)<>[lw(get, "URL1"), @lw(get, "URL2"), lw(get, "URL3")],
      lw(get, "URL4"),
      lw(get, "URL5")].
```

A downloaded LogicWeb program typically does not run in isolation. It may download and use other LogicWeb programs or utilise local system resources. The interpreter implements the linking to other programs and local system resources dynamically as the need arises and not during compilation. Loading remote programs on demand is an important feature for LogicWeb applications which often can only determine the required programs (or data) at run-time. Such *dynamic linking* to remote code and to local resources is typical of mobile code languages [61].

For practical programming, the interpreter has been extended to inherit language features found in SWI-Prolog including cut (!), and built-in predicates such as `setof/3`, `bagof/3`, `maplist/3`, `predsort/3`, `once/1`, if-then-else (`->/3`), `assert/1`, `retract/1`, list manipulation predicates (e.g., `append/3` and `member/2`), and `not/1`. These extensions are based on work on Prolog meta-interpreters (as noted in Chapter 2). For example, the rule

```
demo(E, A) :- built_in(A), call(A).
```

is added to the interpreter where `built_in/1` determines if `A` is a built-in predicate and if so, calls `A` using the Prolog meta-predicate `call/1`. A similar rule can be used to define the set of built-in predicates which a LogicWeb program is allowed to invoke (see Chapter 8). Higher-order predicates are used by calling `demo/2` in the body. For instance, for `not/1`, the following rule is added:

```
demo(E, not(A)) :- not(demo(E, A)).
```

Although much of the extensive library of Prolog built-in predicates is usable within LogicWeb programs, new built-in predicates are added to ease the programming of LogicWeb applications. A small set of predicates for communicating with Mosaic, constructing HTML documents, fast string matching, com-

paring dates, checking if a LogicWeb program exists in the program store, and deleting programs in the program store are listed in Appendix B.

4.4.3 Translation into LogicWeb Programs

`download/2` implements the page loader and translator described in Section 4.1. `download/2` requests the object using the specified URL, and if the object is retrieved, calls `create_program/3` to translate it into a LogicWeb program in the way depicted in Figure 3.3 (in Chapter 3). `create_program/3` parses the page storing HTML components into facts. The facts generated from the translation are asserted into the Prolog database. To simplify processing by the LogicWeb interpreter, all facts F belonging to a program P whether generated such as `about/2` and `link/2`, or appearing within the page, are translated into rules of the form:

$$P :: (F \texttt{:- true})$$

Besides the link information stored in `link/2` (mentioned in Section 3.2), the system can also extract and store as facts the title (`title/1`), body (`body/1`), sections (`section/3`), and links to images (`image/2`) and applets (`applet/5`) occurring on the page. Appendix A describes these facts in full. Only programs obtained via GET and POST requests can contain these five types of facts, since programs obtained via HEAD requests do not have HTML text. These facts are generated "on demand" when a call to them is made rather than by default thereby reducing the amount of unused information stored. However, this incurs a slight overhead: each time the facts need to be used, a check is made to see if they have been created. In contrast, `link/2` facts are generated by default since link information is commonly used in applications.

These facts support applications which examine those components, such as applet or image searching over the Web. These components are used in the examples described in the next three chapters. The body and sections of a page are

coarse-grained fragments which can be further broken up. Images and applets are deemed to be significant for applications. For instance, the popular page index AltaVista can be searched based on the occurrence of keywords in the pages' mark-up for titles, anchors, applets, and images[2]. However, which components are extracted from the text ultimately depends on the particular application. For example, the text in the page's body or sections can be further parsed for structures such as ordered lists, tables, or forms. Instead of supporting all levels of parsing, it is left to the specific application to determine what further to extract.

As mentioned in Chapter 2, HTML is evolving. Later HTML specifications standardise tags which allow multimedia components to be incorporated and links to be labelled with relationship descriptions. The extraction of these components can be supported when they are more widely used.

4.4.4 Caching LogicWeb Programs

A number of practical issues arise with the caching of LogicWeb programs.

Limited storage space. A policy is needed to determine which programs to remove whenever space runs out. The programs may be ranked according to how often they are used, or when they were last referenced. Otherwise, the system can simply disallow creation of new programs when some limit is attained.

Side-effects on backtracking. On backtracking over a LogicWeb goal (e.g., `E#>G`), a question is whether created programs should be removed. For instance, the inference rules given in Section 3.3.3 do not keep track of programs created in failed branches during a proof search. The set of created programs in the rules allows goals in a proof to reuse created programs, but when a goal fails and the proof search proceeds to a different branch the set of created programs

[2]See AltaVista help page at `http://www.altavista.yellowpages.com.au/cgi-bin/query?mss=helpadv&pg=ah`.

is "dumped". In contrast, the LogicWeb program interpreter does not remove created programs on backtracking.

Persistency across applications. One more issue is whether caching should be persistent across applications or local to each application. A related issue is whether the program store should be saved and restored in another session. If caching persists across applications, network access is reduced since different applications can use previously loaded programs. But this may cause applications to interfere with each other and an application might want to use the latest programs.

In the implementation, LogicWeb programs are cached to allow reuse, thereby keeping network access to a minimum and consistency in successful goal evaluations. Such caching corresponds to the semantics of LogicWeb goals discussed in Chapter 3. While an application is in use, the LogicWeb programs used directly or indirectly by it are not updated. Programs on the client-side are not kept consistent with their original copies residing on the server-side. However, the user can clear the cache at any time without exiting the system using a LogicWeb application called the *LogicWeb program manager*. This application uses the built-in predicate `delete_programs/0` (see Appendix B) to clear all the programs in the program store except the program manager itself. After the program store is cleared, applications can be reloaded, allowing queries to be evaluated with the most recent programs.

The cache stores all programs created during a proof search including those in failed branches since they could be used for other goals. Keeping programs created from failed branches is an implementation addendum to reduce network access and is not specified in the operational semantics of Chapter 3. This behaviour means that the set of programs created during a proof search may be larger than that necessary for the proof (where the right goals are always evaluated).

To deal with limited storage space, the amount of storage an application can use is restricted. Once the limit is used, no more programs can be downloaded. In the current implementation, this limit is large enough to support an application downloading several megabytes of information. Enforcement of resource limits is discussed further in Chapter 8.

4.5 Discussion

This chapter has shown how LogicWeb can be realised, describing an implementation in Prolog of the LogicWeb system which extends the NCSA Mosaic browser. Prolog is a natural choice as the implementation language since LogicWeb programs are Prolog-based and due to the extensive existing work on Prolog meta-programming. In fact, the use of Prolog permitted the rapid prototyping and specification of the key components of the system.

This implementation continues a recent trend which is to write Web browsers and the mobile code they support in the same language. This trend began with the HotJava browser [153] which is implemented in Java and supports Java applets. Another example is the MMM browser [173] which is implemented in Objective Caml (an object-oriented dialect of the functional language ML) using an interface to Tcl/Tk, and supports mobile code in Objective Caml. A third example is SurfIt! [17] which supports Tcl applets (or Tclets). The advantage of using the same language for the browser and mobile code is perhaps not just the ease of implementation, or the demonstration of the adequacy of the language for building such Web tools, but also to provide a browser architecture which is customisable or extensible by mobile code without interfacing between different languages. Chapter 8 shows that LogicWeb programs can be used not only for building applications but to dynamically modify the functionality of the Prolog engine.

Below, we look into different aspects of the implementation pointing out ar-

eas of weaknesses and identifying possible avenues for future implementation work.

Performance. Execution performance of LogicWeb programs has not been the focus of the current implementation. Computation time (or CPU time) is less significant for applications that spend much of their wall-clock execution time waiting for and retrieving information from the Web. Moreover, as mentioned earlier, speed critical utilities (e.g., string matching is used often) are coded in C. Nevertheless, interpreted code generally runs slower than compiled code. Other execution schemes can be considered with their own security and resource control mechanisms such as executing downloaded compiled byte-code.

Naming and content types. LogicWeb programs are named according to the HTTP method and URL that is used to retrieve them. The implementation only supports URLs which name HTML files (i.e., of the `text/html` MIME type) since LogicWeb maps HTML content to logic programs.

User interface. Using only the basic graphical user interface facilities provided by HTML (i.e., forms and anchors) has the advantage of applications being usable with the simplest browsers, namely, browsers which support only forms and anchors. Interfacing Prolog with another mobile code language with a rich set of GUI libraries (e.g., Java) would offer the prospect of combining rule-based inferencing with a more complex GUI. Interfacing Prolog with Java is discussed further in Chapter 9.

Security and resource control. Security is an issue when LogicWeb programs are allowed access to system resources. Interpreting source level programs provides a convenient means for implementing security and resource control mechanisms. Security issues are discussed in detail in Chapter 8.

Availability and portability. Work on the implementation begun in early 1995 when Mosaic was the most popular browser and CCI provided an easy means to utilise the capabilities of Mosaic from an external application. Today, Netscape and Microsoft Internet Explorer have surpassed Mosaic as the more widely used browsers. Also, since 1995, a variety of technologies have developed which can be used to implement LogicWeb. Hence, alternative implementations should be considered.

The platform for the above implementation is UNIX. On UNIX, Netscape does not support CCI or the sending of events (e.g., link activations) to an external application. On Windows NT, DDE and OLE are used to to provide this capability, and on the Mac, AppleEvents can be used with Netscape for this purpose. The Spyglass Software Development Interface [183] defines a library similar to CCI, but which is browser independent. Such an interface has not yet been implemented for UNIX.

A browser independent approach for implementing LogicWeb is to build the LogicWeb client as a proxy. A proxy is a program situated between a browser (the proxy user) and servers, and hence, can intercept the browser's requests. LogicWeb programs can be cached on and queried via the proxy. However, this approach has a weakness. Typically, a proxy listens at a particular host name (on which the proxy runs) and port number. This means that, unless there is a secure communication protocol between the LogicWeb proxy and its user, any party which knows the proxy's location and port number can connect to it and interfere with its operation. The result is that the proxy is no longer dedicated to a single user.

On the other hand, LogicWeb functionality can be integrated into a browser. For instance, a Prolog to Tcl/Tk interface [143] (e.g., as offered by ECLiPSe Prolog) can be used to combine the LogicWeb Prolog engine with an existing Tcl/Tk browser such as SurfIt!. Another possibility is to build a Prolog system into a Web browser such as Netscape Navigator. The tight integration with a browser

can offer enormous advantages since almost all user actions can be made available to a LogicWeb application which then can customise the browser's functionalities. Furthermore, with this approach, a CGI script will not be needed to relay form inputs back to the LogicWeb system as in the current implementation.

Other ways of implementing LogicWeb which build on recent integrations of Prolog technology with the Web are discussed in Chapter 9.

Chapter 5

CiFi

This chapter illustrates programming techniques and advantages of the LogicWeb language for coding tools that search the Web for specific items of information. The main example of this chapter is a LogicWeb application for finding citations of specific Computer Science publications on the Web called CiFi.

Section 5.1 discusses strategies for finding citations of Computer Science publications on the Web, introducing CiFi's approach. Section 5.2 presents the design and implementation of CiFi, where CiFi's components and search strategies are outlined, and the encoding using LogicWeb rules of these components and strategies are given. Limitations of CiFi are discussed in Section 5.3, and related work is reviewed in Section 5.4.

5.1 Looking for Citations on the World Wide Web

The Web is increasingly a rich source of Computer Science research publications. Researchers are making their publications available on the Web from their homepages, university publication pages, and technical report archives. These publications are usually in the form of citations with links to electronic copies of papers. Conference proceedings are appearing on the Web, and extensive biblio-

graphic databases are already available on-line. Often, the most recent research publications can be found on the Web.

Journal editors seek the latest citations of Computer Science publications in order to replace references to outdated and less easily available preprints and technical reports. Graduate students seek the full copies of papers for their literature reviews. Authors may want to look for papers on the Web, and cite the corresponding URLs, even when they can cite the paper as appearing in some hard-copy conference proceedings, or as a technical report. The problem is in finding the specific citation amidst the ocean of information.

Two types of searchable indexes are widely used for information discovery on the Web. The first type indexes pages. These indexes are generated by indexing agents that traverse the Web "off-line", indexing Web pages based on the words they contain. Keywords can be submitted to these search engines to recover links to documents indexed by the keywords. AltaVista and Lycos are notable examples of this approach.[1] However, search engines do not take the user directly to the specific piece of information being sought, but to documents that possibly contain or lead to the required information. These documents become the starting points for browsing. The second type indexes a specific kind of information. Examples of information indexed are subject based bibliographies and departmental technical reports (e.g., the *Unified Computer Science Technical Reports* archive[2]). The contents of these indexes are not searchable from the page indexes.

Even with extensive searchable technical report and bibliographic databases, finding the latest citation or copy of a specific Computer Science publication can still be a challenge. At any time, none of the databases is complete, and the seeker should try several of them.

[1] Many other search engines are listed and evaluated in `http://www.ambrosiasw.com/~fprefect/matrix/matrix.shtml`.

[2] `http://www.cs.indiana.edu/cstr/search`

The seeker may also use page indexes submitting the author's name and the publication title as search keywords. Depending on the query, these indexes can return links to useful pages, such as personal homepages, which usually have links to publications, or HTML versions of papers. Using page indexes alone to find citations is not sufficient since such indexes themselves have several fundamental drawbacks. First, keeping the index up-to-date is difficult, especially faced with the increasing size and growth rate of the Web. For instance, it may take several weeks before a new site is discovered, or before the engine revisits a page and updates its index entry. This means that a new page may contain a citation, but the old version, indexed by the search engine, does not. Pages may have been deleted, moved, or renamed since they were last indexed so that links to them become stale. A related problem will be the size of the page indexes, which will become unmanageably large as the Web continues to grow. Second, if the keywords (e.g., title and author name) happen to be quite common words, then many irrelevant documents will be returned. This will continue to be a problem as the Web grows. Moreover, as noted in [134] a word often has different meanings causing redundant information to appear in search results. For instance, the word "cook" could be someone's last name besides its usual meaning. Third, it is often difficult to determine (even manually), from the information returned by the search engine, if a page contains the citation. Further browsing from the search results will be required. Fourth, the page indexes do not contain information in databases with their own search facilities (e.g., bibliographic databases).

Hence, using multiple searchable indexes coupled with Web browsing starting from search results becomes a viable strategy, and has the advantage of robustness. For example, alternative resources can be utilised when initial resources are temporarily unavailable (e.g., down or busy). But the seeker has to know the appropriate sites, and invest time wading through much data which takes much time to download, especially with the current speed of network connections.[3]

[3] A recent Web user's survey which ran from October to November 1997 found that the speed

The seeker could instead delegate the task to a software entity (often called a *software agent* - e.g., [81, 162]) with the required knowledge to work autonomously. Such an agent does not only save the user's time but also does not require the user to have knowledge about how to find the citations (e.g., the URLs of sites and strategies).

CIFI is a rule-based agent which combines index querying with more extensive conventional browsing, guided by search heuristics tuned for finding citations. Empirical evidence suggests that author's homepages, author's publications pages linked from author homepages, and departmental publications pages are the main sources of citations. Also, if the author's homepage can be obtained, then the other types of pages can usually be found. For instance, the URL of the Computer Science departmental page can usually be extracted from the URL of the author's homepage. Thus, an approach will be to use a search engine to find the author's homepage and then use search heuristics like the one outlined above to guide browsing until the citation is found. This method has the following benefits:

- There is a degree of resilience to change, since it relies only on the homepage being indexed. Changes inside the homepage, or changes in the location and content of the publications pages can be accommodated, since these pages are accessed in real-time, instead of being found via indexed information.

- Browsing takes the user directly to the required information.

CIFI also searches technical report archives, which are not indexed by search engines.

---

of downloading pages is the most common problem with using the Web. The results are available at `http://www.gvu.gatech.edu/user_surveys/survey-1997-10/graphs/use/Problems_Using_the_Web.html`.

5.2 Design and Implementation of CIFI

A paper is uniquely identified by its author's name and title. CIFI takes, as input, the author's last name and given names (if available) and the title (which may be incomplete). It attempts to return the HTML version of the paper, or a citation of the paper, without further user intervention.

The knowledge-based approach is commonly used for building expert systems, and CIFI has been built using such an approach. CIFI is built on the premise that a Web search tool for finding specific information on the Web comprises three components:

1. *the search algorithm*: Web searching involves traversing the graph-like hypertext structure of the Web.

2. *required knowledge*: knowledge is employed in the search algorithm to control searching in order to increase search efficiency, accuracy, and effectiveness.

3. *routines to retrieve and parse Web pages*.

Prolog backtracking search can be used to simplify the coding of depth-first searching of portions of the Web. Relevant knowledge about search strategies and Web site structure can be encoded declaratively and easily as rules. LogicWeb operators take care of the details of page retrieval, storage, and parsing, enabling the programmer to focus on the search algorithm and knowledge. These advantages are illustrated in Section 5.2.2 and 5.2.3.

5.2.1 Alternative Strategies

CIFI uses a number of strategies to search the Web:

1. *Search the Web for the HTML version of the paper*. Words in the title and an author's name (last name and optionally, given names) are used as query keywords to Lycos.

2. *Search from the author's homepage*. The author's name (last name and optionally, given names) are used as keywords to Lycos. Lycos returns a number of links which possibly point to the author's homepage. These become starting points for further browsing.

3. *Search from the university Computer Science department homepage*. The Computer Science department URL is extracted from the homepage URL. The Computer Science department homepage is then explored for its technical report or publications page. If there are several possible homepage links, several possible links to departmental homepages are extracted, and explored for publications.

4. *Search technical report archives*. Queries are sent to technical report search engines: the *Unified Computer Science Technical Reports* archive, and the *Networked Computer Science Technical Reports Library*[4].

5.2.2 Search Algorithm

Each of the strategies outlined above is represented using rules. In addition, a second type of rule, *navigational rules*, is used to encode the actions to take when on a particular type of page (e.g., a homepage).

Page types are names for classes of pages (some of which may contain only one page). They enable individual pages to be referred to by their types rather than by their URLs which would be too specific, and may change over time. Page types also allow collections of pages to be conveniently addressed and relationships to be stated between collections of pages. Examples of page types are: author homepages (called `auth_homepage`), Computer Science department homepages (`cs_homepage`), the page of the first ten Lycos results using the title and author name as keywords (`lycos_auth_title_srch`), and the page of the first ten Lycos results using the author name as keywords (`lycos_auth_srch`).

[4] `http://alvin.cs.cornell.edu/`

In general, a citation is found using a page if: (a) the citation is on the page (the page type is first checked to see if it is likely to contain the citation), or (b) the citation is found using a page which is linked from the page. For example, if a homepage does not contain publications details for the author, then a search is made for a link to the author's publications page.

Strategies for manual citation searches were used as the basis for the CIFI rules. Manual citation searches were carried out to find common ways in which authors and Computer Science departments make their technical reports and publications available on the Web, with particular attention paid to the hypertext structure, and the labels and URLs used. The heuristics mostly relate to the selection of links during browsing, which rely on the type of the page.

Rules for a heuristic-free search over a part of the Web is first presented. These rules describe a search of the Web exploring each page for the required citation, starting from a given page, until the citation is found:

```
find_citation(ProgramID, KeyWords, Citation) :-
  ProgramID#>h_text(Source),
  contains_citation(Source, KeyWords),
  extract_citation(Source, KeyWords, Citation).
find_citation(ProgramID, KeyWords, Citation) :-
  ProgramID#>link(Label, NextURL),
  find_citation(lw(get, NextURL), KeyWords, Citation).
```

In the above, the predicate, `find_citation/3`, is true if the citation is found from the given page, identified by `ProgramID`. `ProgramID` is a LogicWeb program identifier. `KeyWords` are the keywords (author name and title) entered by the user. `Citation` is a text fragment of the page containing the required citation. Note that the above rules permit infinite looping if the link goes back to a previous page. Such looping can be prevented by recording previously visited URLs, and not revisiting pages.

The predicate `contains_citation/2` determines if a page possibly contains the required citation, by checking to see if the page contains all the title keywords and the author's last name. The page's source is retrieved from

h_text/1. The predicate extract_citation/3 extracts the required citation from the page.

In the second rule, the LogicWeb goal retrieves a link from the page, and can be used to obtain different links through backtracking. No parsing is required to examine the links due to the abstraction of page contents as facts in a LogicWeb program. The second rule is not used if the first rule succeeds.

An example of the use of find_citation/3 is the following goal:

```
?- find_citation(lw(get, "http://www.cs.mu.oz.au/~swloke"),
                 ["Logic", "Programming", "World", "Wide", "Web"],
                 Citation).
```

Searching every page in this way is impractical, being much too expensive in time and computation.

To minimise the search space, link selection heuristics and knowledge about page types are employed. Only certain types of pages contain citations. For example, Computer Science department homepages do not, but author homepages may. Also, only particular links will lead to a page containing citations. The above rules, augmented with heuristics and page types, become the following:

```
find_citation(PageType, ProgramID, KeyWords, Citation) :-
  pagetype_with_citations(PageType),
  ProgramID#>h_text(Source),
  contains_citation(Source, KeyWords),
  extract_citation(Source, KeyWords, Citation).
find_citation(PageType, ProgramID, KeyWords, Citation) :-
  ProgramID#>link(Label, NextURL),
  sat_link_criteria(PageType, link(Label, NextURL), KeyWords,
                    NextPageType),
  find_citation(NextPageType, lw(get, NextURL), KeyWords, Citation).
```

PageType and NextPageType are page type identifiers indicating the current and the next page types respectively. The predicate pagetype_with_citations/1 determines if pages of the given page type could contain citations. This predicate ensures that only likely pages are checked

for the required citation (e.g., `auth_homepage` but not `cs_homepage`). Heuristics for link selection are applied in the predicate `sat_link_criteria/4`.

These rules describe a heuristic-guided search of a part of the Web.

An example of the use of `find_citation/4` is the following goal which begins searching from a homepage:

```
?- find_citation(auth_homepage,
                 lw(get, "http://www.cs.mu.oz.au/~swloke"),
                 ["Logic", "Programming", "World", "Wide", "Web"],
                 Citation).
```

5.2.3 Obtaining Starting Points and Link Selection

The predicate `sat_link_criteria/4` determines, given the page type and keywords, whether a link should be followed (i.e., whether the link satisfies some criteria described below), and the next page type to expect if the link is followed. Without this predicate, the search degenerates to exploring all links.

Relationships between a page type and its next page type(s) are expressed using `leads_to/2` facts. These relationships direct the search from one page type to another. Also, `cue_strings/2` facts are used to store strings that are looked for in the label of a link that would lead to each type of page. Below, we look at heuristics for link selection on different pages, giving several examples of these facts:

1. Lycos is queried using the author's name and keywords from the title. To determine which link, if any, in the Lycos results go to the HTML page, the label of the link is checked for all the title keywords. If they are all present, then the link goes to the paper.

2. The following rules are used to determine if a Lycos generated link goes to an author's homepage:

- If the label contains the author's name and the words "Home" (or "Personal") and "Page", then the link goes to the homepage.
- If the label contains parts of the author's name, or the phrase "Faculty Advisor", then the link goes to the author's homepage. In addition, the URL is inspected for the phrases "people", "Faculty", "researcher", "staff", "user", and "fac", or for the form:

  `<server-URL>/~<identifier>`

  For example, the following URL references a homepage:

  `http://www.cs.mu.oz.au/~eas`

  The URL is parsed, and inspected in a predicate. The phrases above are stored in a fact.

To look for the author's homepage after this Lycos search, the relevant facts used by `sat_link_criteria/4` are:

```
leads_to(lycos_auth_srch, auth_homepage).
cue_strings(auth_homepage,["Homepage", "Home page",
                            "Personal Page", "Faculty Advisor"]).
```

The fact `leads_to/2` above specifies the type of the next page to visit (i.e. `auth_homepage`) whenever on a `lycos_auth_srch` page. The strings to look for in a link to an author's homepage is specified in the fact `cue_strings/2` above.

3. If the publications are not found on an author's homepage, then a search is made for links and URLs containing the keywords "publications" or "papers". The check on the URLs ensures that links like "My publications are found here." are examined. The facts used for link selection are:

```
leads_to(auth_homepage, auth_pub_page).
cue_strings(auth_pub_page,["Publications", "Papers"]).
```

4. The Computer Science department server's URL is extracted from the author's homepage URL. For example,

   ```
   http://www.cs.mu.oz.au/
   ```

   is extracted from

   ```
   http://www.cs.mu.oz.au/~eas
   ```

   In the Computer Science department homepage, CIFI looks for link labels containing "technical report" or "publications". The motivation for this is that a Computer Science department homepage usually has a direct link to a technical report page, or to a publications page, with a link to the technical report page. This reasoning leads to the following `leads_to/2` and `cue_strings/2` facts:

   ```
   leads_to(cs_homepage, cs_pub_page).
   leads_to(cs_pub_page, cs_tr_page).
   leads_to(cs_homepage, cs_tr_page).
   cue_strings(cs_pub_page, ["Publications"]).
   cue_strings(cs_tr_page, ["Technical Reports"]).
   ```

The `leads_to/2` facts capture the common (and relevant) Web structure across Computer Science department Web sites.

5.2.4 Extracting the Citation

The citation is extracted by scanning the page, looking for a fragment containing all the title keywords and the author's last name. The size of the fragment extracted was chosen by measuring the sizes of several hundred citations.

Heuristics are employed to ensure what is extracted is indeed a citation: a citation is often given as a bullet point, and a phrase like "selected publications" often precedes it.

5.2.5 Integrating Other Information Sources

As one of its search methods, CIFI sends queries to several technical report archive search engines. The results are checked for the citation required using a method like the one described above. Search engines are queried using LogicWeb goals containing the search engines' URLs appropriately appended with search keywords in the way shown in Section 2.1.2.1.

5.2.6 Implementation

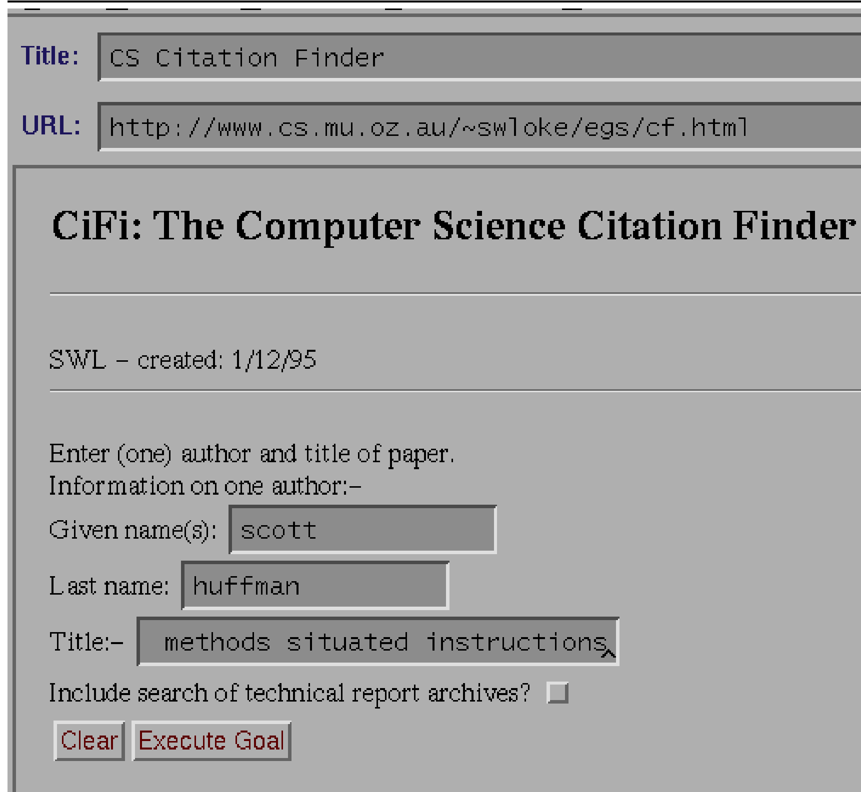
CIFI consists of only one LogicWeb program. CIFI's user interface accepts the author's last name, the given names, and title keywords (see Figure 5.1), and is defined by the following `interface/1` predicate:

```
interface([textnl("Enter (one) author and title of paper."),
           textnl("<BR>Information on one author:-"),
           textnl("<BR>Given name(s):"),
           input("text", "givennames", "", "", "16", "16"),
           textnl("<BR>Last name:"),
           input("text", "lastname", "", "", "16", "16"),
           textnl("<BR>Title:-"),
           input("text", "title", "", "", "30", "70"),
           textnl("<BR>Include search of technical report archives?"),
           input("checkbox", "srchtr", "str", "", "", "")
          ]).
```

`textnl/1` displays text with a newline after the text, and `input/6` displays an input field. These two terms are detailed in Appendix B.

On receiving the inputs, CIFI's `do_query/2` predicate filters out common words and invokes `find_citation/4` with a starting page type, such as `lycos_auth_title_srch` (corresponding to strategy 1 in Section 5.2.1) or `lycos_auth_srch` (corresponding to strategy 2 in Section 5.2.1), a starting program identifier (for a Lycos search, the starting URL is constructed as described in Section 5.2.5), and the filtered inputs in its arguments. The citation, if found,

Figure 5.1 The interface to CIFI.



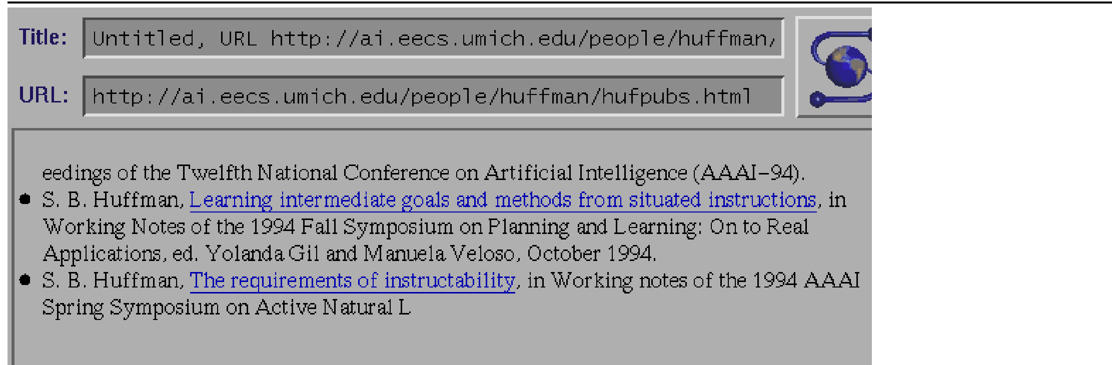
is displayed on the browser (see Figure 5.2), allowing the user to follow links occurring in the citation (if any).

5.3 Limitations of CIFI

Thirty different citation searches were carried out, of which two-thirds were found. These citations were found mostly on authors' publications pages, authors' homepages, and departmental technical report pages. One citation was found by querying the technical report archives.[5]

[5]This is because technical report archives are searched last. Using technical report archives alone is inadequate, since they contain only technical reports of a limited number of participating institutions, and many publications are not technical reports.

Figure 5.2 The result of a search. The required citation is the first citation in the displayed page fragment.



Failure cases highlight the following limitations of the current implementation:

- Insufficient keywords lead to failure or erroneous results. Including the given names of the author helps to avoid problems, and there should be sufficient keywords from the title.

- CIFI relies on Lycos to find the homepage for an author. In several searches, the homepage was not found among the first ten hits. Other homepage databases on the Web could be employed such as indexes of personal homepages categorised by university[6], or indexes of Computer Science departments, such as the Australian Computer Science Department sites[7], if the user can supply information on the affiliation of authors.

- The strategy used to look for the HTML copy of the paper may only find a page containing its abstract. One way to deal with the problem is to examine the context surrounding the link to the page for the word "abstract".

- Additional heuristics are needed for departmental pages where the technical reports are classified by research groups or by year. For instance, such

[6] http://www.utexas.edu/world/personal/index.html

[7] http://www.cs.jcu.edu.au/ftp/web/webAdmin/ozuni.html

a format might allow a paper to be found based on its relatedness to words in the research group titles, or its year of publication.

- One link with the paper title as its label actually pointed to a redirection page, because the page had been moved. Rules can be added to CIFI to deal with redirections (such as those given in Chapter 7).

- Some link labels to papers do not contain all the keywords from the title. For example, for the paper "Citescapes: Supporting Knowledge Construction on the Web", the link has the label "Citescapes Paper". Project or system names, like "Citescapes", could be distinguished in the keywords, and then be used alone to judge each link.

- If the title of a paper is misspelled in a citation, CIFI may fail to find it. Loose string matching could be used to avoid this problem.

- If a link to the (HTML) paper is part of a clickable map, the paper will not be found.

5.4 Related Work

5.4.1 Agents for Paper Search

Constructing tools to find research publication information on the Internet is a challenging task. Two such tools different from CIFI in terms of search strategies and resources used are discussed below.

A tool called WEBFIND for automatically searching for scientific papers is introduced in [150]. The main idea of WEBFIND is to use other (non-Web) information sources to help retrieve information from the Web. It uses the Melvyl database (a University of California library service) to find the institutional affiliations of an author of a paper, and employs NetFind to find the Internet address of a computer with that affiliation. It then uses this address to construct the URL

of the affiliation's server. WEBFIND utilises heuristics to explore the server for the author's homepage and publications page, in a similar way to CIFI's navigational rules. This tool is limited by the contents of the Melvyl database. In contrast to WEBFIND, CIFI looks for a citation to the paper which may, or may not, have a link to the actual paper. Also, CIFI uses multiple strategies consisting of a mixture of Web-based search engines, on-line databases, and heuristics, without depending on external information sources.

BibAgent [174] semi-autonomously navigates over FTP directories (using the Alex file-system that integrates FTP directories) looking for a specified article. BibAgent examines `readme` files to aid its navigation, prioritises the directories to follow, and can retrieve the actual paper, or a completed bibliographic reference, from bibliographic files (e.g.,`.bib` files). BibAgent asks the user for traversal suggestions, and learns useful search paths for future use. In contrast to BibAgent, CIFI navigates over the Web, and searches Web pages for citations.

A limitation of the current implementation of the LogicWeb language is that only resources accessible via HTTP can be used in context switching (i.e., the Melvyl database and FTP sites are excluded).

5.4.2 Web Search Tools

This subsection differentiates CIFI from other tools which use heuristics and agents which automate the Web browsing task.

5.4.2.1 Internet Fish

Internet Fish [120] is a class of information discovery tools with persistent behaviour. The tools run continuously and can remember queries over time. Internet Fish, like CIFI, is based on a knowledge intensive approach. It keeps knowledge about on-line resources (e.g., thesaurus, name servers, and searchable page indexes), the Internet structure (e.g., format of server names), what is interesting

to the user, and rules about what to do with its knowledge.

CIFI works autonomously whereas Internet Fish interacts with the user during searching. For instance, Internet Fish questions the user on the relevance of keywords, and its intermediate search results. Additional knowledge can be added to Internet Fish via a construction toolkit. Internet Fish is implemented using the language Scheme.

5.4.2.2 General Heuristics Involving Web Links

In [182], a number of general heuristics based on link information are outlined, which can be utilised for finding homepages, related pages, new locations of moved pages, and unindexed information. For example, one heuristic is: if P is a homepage and file P′ is in a directory below that of P′, then P′ is likely to be authored by the person identified on page P. In contrast to these heuristics, CIFI's heuristics involve domain-specific page types.

5.4.2.3 An Abductive Framework for Web Searching

Prendinger [167] describes a client-side system implemented in Prolog for applying abductive reasoning in the design of a Web search tool. Reachability relations are used to model the systems's partial knowledge of the Web. This work is still in a preliminary stage, and does not yet offer more functionality than bounded depth-first search guided by the occurrence or non-occurrence of keywords in the page's text.

5.4.2.4 Browsing Agents

Browsing agents "surf the Web" on behalf of the user according to some criteria, or guide the user during browsing. Most of these perform more general searches, rather than targeting a particular domain, such as papers. Also, most learn user interests or permit user feedback. For instance, in [15], user feedback on inter-

mediate search results is used to change the link selection heuristic, improving subsequent search results.

Fish-Search [68, 67], traverses the Web looking for particular documents. The user specifies either keywords, a regular expression, or external filters that the contents of the document must match. Heuristic rules used to guide the search are handcoded-in and include: (1) after following a number of links in a given direction without finding relevant documents, the search stops going in that direction; (2) links in relevant documents are traversed first before those of less relevant ones; (3) links to documents in different sites are preferred.

WebWatcher [107, 9] uses a description of user interests to highlight interesting hyperlinks, and records hyperlinks to related pages. It also remembers the user's interests, based on the pages selected.

Letizia [122] is an agent that infers user interests from browsing behaviour, and explores links using a best-first search with heuristics utilising the inferred user interests. Based on its exploration, Letizia can recommend links to follow.

A case-based approach for information retrieval is utilised in [96]. Past user feedback on example items allows it to suggest potentially relevant new items to the user.

CIFI does not require feedback, and does not learn from user interests because there is a specific target (e.g., a citation, or a HTML paper). The required browsing behaviour is precisely specified using rules.

5.5 Discussion

CIFI uses heuristics about the organisation of sites being searched and Web-based information sources to obtain its results. CIFI shows that logic programming permits the succinct coding of both the procedural and knowledge components of Web search tools with a clean separation between the components. The navigational rules show that it is possible to exploit the unwritten conventions of the

Web, and that Prolog backtracking search maps well to the Web browsing task. The rules also show how the low-level details of Web page retrieval and parsing are abstracted away from the search algorithm. The treatment of LogicWeb program identifiers as first class entities makes it easy for programs to manipulate pages and to access their contents.

The experiments demonstrate the efficacy of CIFI, and point out its limitations. Section 5.3 has pointed out how most of these limitations can be addressed by extending the rule set or by the user supplying more information.

CIFI's performance can improve with better resources and more search strategies. Work on CIFI has continued independently of LogicWeb as described in [97], where additional resources have been used including the use of AltaVista and more recent publication databases, and the detection and use of the search facility on a departmental publications page. This has improved search results from twenty to twenty-four successful searches (using the same thirty queries). Useful results are returned even when the citation can not be obtained such as a page about a conference that mentions the paper. The use of page types to guide search has been investigated for other categories of information (besides citations) in a shell for building information agents [132].

As demonstrated by CIFI, Prolog backtracking search eases the coding of depth-first searching behaviour on the Web. Other search algorithms can be succinctly coded in Prolog [185, 161], and so, other Web search algorithms can be succinctly coded in the LogicWeb language. The Web search algorithms can also exploit structured information on the Web and deal with download failures. For example, the following program traverses the Web starting from a given URL in a bounded breadth-first search manner looking for pages relevant to a given list of topics.

The top-level query to the program takes the form:

```
?- collect(["http://www.cs.mu.oz.au/~ad"],
           ["logic", "AI", "Web"], 3, 20, [], Ps).
```

The first argument is a list of starting addresses. Each page will be scored using a `score_page/3` predicate which utilises the keywords in the second list. If a score greater than 3 (the third argument value) is obtained, then the page's links are collected and subsequently searched. All the collected pages are stored in a list which is eventually returned in `Ps`. The search stops when 20 suitable pages (the fourth argument value) have been found, or there are no more URLs to explore.

`collect/6` is defined as:

```
collect(_, _, _, Max, Ps, Ps) :-      % got enough addresses
  length(Ps, Len), Len >= Max.
collect([], _, _, _, Ps, Ps).         % no more URLs to examine
collect([URL|ToVisit], Keys, PScore, Max, Ps, FPs) :-
  lw(get, URL)#>h_text(Text),
  score_page(Keys, Text, Score),
  act_score(Score, URL, ToVisit, Keys, PScore, Max, Ps, FPs).

act_score(Score, CurrURL, ToVisit, Keys, PScore, Max, Ps, FPs) :-
  Score > PScore,
  setof(URL, [Label]^lw(get, CurrURL)#>link(Label,URL), URLs),
  append(ToVisit, URLs, ToVisit1),
  collect(ToVisit1, Keys, PScore, Max, [URL|Ps], FPs).
act_score(Score, _, ToVisit, Keys, PScore, Max, Ps, FPs) :-
  Score =< PScore,
  collect(ToVisit, Keys, PScore, Max, Ps, FPs).
```

`collect/6` can terminate either when `Max` pages have been collected or when the URLs in the `ToVisit` list have been exhausted. Otherwise, `score_page/3` is used to get a score for the page, which is acted upon by `act_score/8`. `act_score/8` actions depend on whether the page score is higher than the pass score (`PScore`). If it is higher then the page's links are appended to the *end* of the `ToVisit` list and the collection process continues. By appending to the end, a breadth-first search is maintained.

The accuracy and effectiveness of the search can be increased by exploiting structured information on the visited pages. Suppose that departmental research

staff want information about their research interests to be more readily processable and query-able by programs. Then, they can encode this information in the form of the `interested_in/1` rules as shown in Section 3.2. These rules can be used to determine how interesting a page is. For example, if the pages the breadth-first search utility visits contain such a predicate, then a new score predicate can utilise `interested_in/1`:

```
score_url(Keys, URL, Score) :-
  setof(K, (member(K, Keys), lw(get, URL)#>interested_in(K)), Ks),
  length(Ks, Score).
```

The score is the number of keywords of interest to the page's author.

Facts on visited pages which state useful URLs about a topic can be exploited such as:

```
useful_pages("Logic Programming",
   ["http://www-lp.doc.ic.ac.uk/",
    "http://www.comlab.ox.ac.uk/archive/logic-prog.html"]).
```

Then, the URLs in `useful_pages/2` can be added to the `ToVisit` list if they were related to any of the search keywords. Structured information in the form deductive databases is described in Chapter 6.

The breadth-first search utility and CIFI can be easily modified to take into account download failures. Nondeterminism in logic programming can reflect the nondeterministic nature of the Web (as described in Section 2.1.2.3). For example, in the above breadth-first search utility, when a download fails, the "`#>`" goal in the third clause of `collect/6` will fail. This can be avoided by replacing the goal with:

```
pgm_text(URL, Text)
```

which is defined as:

```
pgm_text(URL, Text) :-
  (lw(get, URL)#>h_text(T) ->
    Text = T
  ;
    Text = ""
  ).
```

Instead of returning the empty string, another way to cope with download failure is to retry the request for the page. The following rules retry a request up to `NumReTries` times:

```
pgm_text(NumReTries, URL, Text) :-
  (lw(get, URL)#>h_text(T) ->
    Text = T
  ;
    NumReTries > 1,
    NumReTries1 is NumReTries - 1,
    sleep(30),           % wait for 30 seconds before retrying
    pgm_text(NumReTries1, URL, Text)
  ).
```

Other ways of coping with download failures are considered in Chapter 7.

As mentioned earlier, many search engines tuned for specific domains are emerging which allow searching over not only homepages and scientific publications, but also job advertisements, corporate sites, news, e-mail addresses, software archives, and FAQs. CIFI has demonstrated that tools which add a layer of client-side processing between the user and search engines can be built into the Web as LogicWeb applications. Being available on the Web means that these tools can be integrated into the mores of Web usage in the same way as the search engines themselves.

The tools can utilise knowledge to select the appropriate search engines to answer a user's query, synthesise the results from multiple search engines (e.g., as done in meta-search engines[8]), browse search results to answer specific queries,

[8] Meta-search engines are search engines which use the results of other search

or browse hierarchical site indexes such as Yahoo[9]. Tools can also be built which aid the user in formulating queries. Since search engines employ keyword searching which are based solely on lexical or syntactic content, they are very sensitive to the choice of words used in queries. If a document is indexed on a synonym of the query keyword, then the document will not be retrieved. Knowledge can be built into the processing layer enabling automatic elaboration of the user's query (e.g., using a database of synonyms), or an interface can be provided, which given the user's general topic of interest, suggests words to the user by querying a knowledge-base. For example, in the system described in [142], the user can browse a hierarchy containing generalisations and specialisations of words through menus in a HTML form.

From the software construction viewpoint, such tools encourage the building of tools out of other tools by first choosing from available tools on the Web, and then, combining them to meet new requirements. For instance, CIFI can be built by integrating a tool for searching bibliographic databases, a tool for searching departmental Web sites for publications, and a tool for finding homepages. The LogicWeb language offers a simple mechanism for using other tools, i.e. by invoking a LogicWeb goal. The LW-composition operators allow customisations to be done. For example, certain predicates in a tool can be overridden by new ones.

---

engines. Examples include Metacrawler at `http://www.metacrawler.com/`, Ahoy! at `http://ahoy.cs.washington.edu:6060/`, and Metabot at `http://metabot.kinetoscope.com/`.

[9] `http://www.yahoo.com/`

Chapter 6

Lightweight Deductive Databases

As mentioned in the introduction, structured information can be represented in the logic programming formalism as deductive databases. This chapter investigates deductive databases which are incorporated into Web pages in the style of [75]. These databases are called *lightweight deductive databases*. A lightweight deductive database is a LogicWeb program with clauses categorised according to three main roles: base relations, derived relations, and rules to process queries[1]. In addition, it may include descriptions of the database (e.g., database schemata) written in ordinary HTML. These databases are lightweight in the sense that they lack the functionality of full database systems, such as transaction processing and query optimisation.

Since lightweight deductive databases are distributed on the Web, they have the following features:

- **Distributed maintenance.** A Web page can contain a part of a database. The completed database can be created as necessary by retrieving the relevant pages, and composing them together. This allows the components of a database to be separately maintained, and combined only during query processing.

[1] These categories are borrowed from the field of deductive databases [116].

- **Extensibility.** The dynamic nature of the complete database allows the incremental addition, or removal, of parts during query evaluation. This facilitates the incremental development of lightweight deductive databases. Moreover, users, who are not the database creators, can extend the existing database by writing their own rules, using schemata included on the pages containing the database.

- **Reusability.** Rules and knowledge-bases for database query processing can be placed in Web pages, encouraging them to be reused, or shared.

- **Client-side processing.** Lightweight deductive databases complement the work on building forms-based interfaces to conventional databases. In the LogicWeb system, query processing is carried out on the client-side, rather than on the server machine. This reduces server load, and permits state information, such as the results of previous queries, to be kept on the client-side. With client-side caching in the LogicWeb system, once the databases are loaded, they need not be fetched again.

The above features enable lightweight deductive databases to be used for organising information on the Web into manageable pieces which can be reused and manipulated in a principled way.

Lightweight deductive databases encode information explicitly as facts and rules in the "`<LW_CODE>...</LW_CODE>`" section of Web pages. This may be seen as a disadvantage since it requires coding by the page authors. An alternative strategy is to include the information in a more informal manner, for example, using HTML tables and new tags for defining rules, and extract it using a separate LogicWeb parsing program written especially for the task. Section 6.6.3 discusses several tag-based systems for including structured information.

Another strategy is to use domain-specific syntactic sugar to define a language for encoding special-purpose databases and queries. Such a language can be parsed into LogicWeb rules, or meta-programming can be employed to build

interpreters for specialised languages as strongly advocated in [186].

This thesis proposes that LogicWeb syntax is itself an expressive and high-level representation formalism for data modelling and querying. A basic set of data types is already defined from which complex data structures can be systematically constructed. For query processing, a Turing-complete language is available, and the features of full Prolog are available if required.

The rest of this chapter illustrates the above features of lightweight deductive databases, using databases of citations as examples in the first three sections. Section 6.1 gives a simple lightweight deductive database example. Section 6.2 introduces techniques for combining and extending lightweight deductive databases including the use of LW-composition operators. Section 6.3 explores an application of lightweight deductive databases for organising citation information on the Web in a form which is more maintainable, extensible, and amenable to complex forms of querying. This section emphasises knowledge-based searching and retrieval of distributed databases. When found, these databases are combined using techniques from Section 6.2. Another application of lightweight deductive databases is presented in Section 6.4, where we look into generating *guided tours* of Web pages by querying lightweight deductive databases. This section emphasises software engineering principles of modularity and reusability in the development of LogicWeb applications. The guided tour was invented by the hypertext community many years before the Web existed, and is interesting in its own right. The use of lightweight deductive databases is a novel approach for creating and using guided tours on the Web. Although the emphasis of LogicWeb is client-side evaluation, server-side computations are sometimes necessary. Section 6.5 shows how the LogicWeb language accesses databases requiring query processing on the server-side. Section 6.6 reviews related work, and compares existing deductive database systems with the lightweight counterpart.

6.1 A Simple Lightweight Deductive Database

Consider a lightweight deductive database of publication citations. A citation consists of distinct components (or attributes), and it should be possible to perform queries using these attributes.

A possible schema for citations is:

pub_cit(authors,title,pub_type,collection_name,web_location,date)

The schema describes the components of a typical publication citation: the names of the authors, the title of the paper (which is used as the primary key, indicated by underlining), the type of the publication (e.g., conference, technical report, or journal), the collection in which the paper was published, the URL of an on-line version, and the date of publication.

An instance of the schema is:

```
pub_cit([author("Seng", "Loke"), author("Andrew", "Davison")],
  "Logic Programming with the World-Wide Web",
  conference, "Hypertext '96",
  "http://www.cs.mu.oz.au/~swloke/papers/paper1.ps.gz",
  date(march, 1996)).
```

This fact is not in relational database normal form, as structured data is used. Some of the attribute values can be stored as atoms instead of strings, but strings are easier to manipulate in queries (e.g., we could utilise pattern matching with a tolerance for character mismatches).

From a database of `pub_cit/6` facts, a database of journal citations in 1996 can be formed, which conforms to the schema:

journal_1996_cit(authors,title,collection_name,month)

The new relation is derived using the rule:

```
journal_1996_cit(Authors, Title, CollectionName, Month) :-
  pub_cit(Authors, Title, journal, CollectionName, _,
          date(Month, 1996)).
```

Rules can be written to process queries on the above databases. For example, the following rules find all the titles of papers by an author in a given year:

```
get_titles(Name, Year, Titles) :-
  setof(Title, get_title(Name, Year, Title), Titles).

get_title(Name, Year, Title) :-
  pub_cit(Authors, Title, _, _, _, date(_, Year)),
  member(Name, Authors).
```

A lightweight deductive database can be downloaded to a LogicWeb system, and queried. Alternatively, the base relations (facts) can be stored in one program, and the query processing rules in another. These can be downloaded separately, and combined on the client-side.

6.2 Combining and Extending Lightweight Deductive Databases

Lightweight deductive databases from disparate sources can be integrated using familiar techniques from the field of deductive databases (e.g., virtual relations and relational joins), and LW-composition operators. These techniques are illustrated below.

6.2.1 Virtual Relations and Relational Joins

A virtual relation is formed from other database relations by a set of rules, all defining the same head predicate, but each using a different relation in its body. In the following examples, we shall assume a database of technical report details with the schema:

tr_cit(authors,<u>title</u>,tr_number,web_location,date)

A virtual relation, `cit/6`, of citations can be defined in terms of technical report and publication details, and stored in a LogicWeb program:

```
my_id(get, "http://www.cit.info").

cit(Authors, Title, Type, BookName, WebLocation, Date) :-
  pub_cit(Authors, Title, Type, BookName, WebLocation, Date).
cit(Authors,Title, technical_report, BookName, WebLocation, Date) :-
  tr_cit(Authors, Title, BookName, WebLocation, Date).
```

Recall that `my_id/2` stores the information found in the LogicWeb program's identifier. `cit/6` assumes that `pub_cit/6` and `tr_cit/5` are present in the current program (i.e., the program containing the virtual relation). However, if the publication and technical report facts are stored in different programs (whose locations are assumed to be stored in `location_of/2` facts), context switching can be employed to retrieve them:

```
cit(Authors, Title, Type, BookName, WebLocation, Date) :-
  location_of(publications, PubsURL),
  lw(get, PubsURL)#>
    pub_cit(Authors, Title, Type, BookName, WebLocation, Date).
cit(Authors, Title, technical_report, BookName, WebLocation, Date) :-
  location_of(technical_reports, TRsURL),
  lw(get, TRsURL)#>
    tr_cit(Authors, Title, BookName, WebLocation, Date).
```

In a relational join, two relations are combined based on common attribute values. A derived relation of citations that are both technical reports and publications can be formed using relational join as follows:

```
pub_tr_cit(Authors, Title, PubWebLocation, TRWebLocation) :-
  location_of(publications, PubsURL),
  location_of(technical_reports, TRsURL),
  lw(get, PubsURL)#>pub_cit(Authors, Title, _, _, PubWebLocation, _),
  lw(get, TRsURL)#>tr_cit(Authors, Title, _, TRWebLocation, _).
```

The shared variables, `Authors` and `Title`, are used to select citations common to both relations.

6.2.2 Forming Virtual Databases Using LW-composition Operators

In [41, 12], meta-level operators such as union, intersection, and restriction (introduced in Section 2.2.3) have been used to combine deductive databases (where each database is a logic program). In a similar way, the LW-composition operators are used to query combinations of lightweight deductive databases or create views over them, but the interfaces of the databases must be carefully designed.

For example, a view of citations dated between 1994 and 1997 and available locally (where the meaning of local is as defined below by substring matching with domain names) can be defined by a program containing:

```
my_id(get, "http://www.cit.info/view1.html").

cit(Authors, Title, Type, BookName, WebLocation, date(Month, Year)) :-
  pub_cit(Authors, Title, Type, BookName, WebLocation,
          date(Month, Year)),
  local(WebLocation),
  Year >= 1994,
  Year =< 1997.

local(WebLocation) :-
  ( contains(WebLocation, "cs.mu.oz.au")
  ; contains(WebLocation, "unimelb.edu.au")
  ).
```

Then, authors of such publications can be queried using:

```
recent_pub_authors(PubDBList, Authors) :-
  (+)<>[lw(get, "http://www.cit.info/view1.html")|PubDBList]#>
         cit(Authors, _, _, _, _, _).
```

`PubDBList` is a list of identifiers of citation databases (i.e., LogicWeb programs) containing `pub_cit/6` facts. The citation information is retrieved from the LW-union of all the programs, which is treated as a virtual citation database. If the programs are on different hosts, the virtual database will then span several hosts.

LW-reduce enables queries to be stated involving collections of databases represented conveniently as Prolog lists.

The following rules define a different view where the publications must either be `conference` or `symposium` and must reside on a local host, but where the meaning of local is more constrained:

```
my_id(get, "http://www.cit.info/view2.html").

cit(Authors, Title, Type, BookName, WebLocation, Date) :-
  pub_cit(Authors, Title, Type, BookName, WebLocation, Date),
  ( Type = conference
  ; Type = symposium
  ),
  local(WebLocation).


local(WebLocation) :-
  ( contains(WebLocation, "http://agent-orange.cs.mu.oz.au")
  ; contains(WebLocation, "http://agent-99.cs.mu.oz.au")
  ; contains(WebLocation, "http://www.unimelb.edu.au")
  ).
```

This definition of `local/1` is more restrictive than the previous one since the hostname is specified. Authors of such publications can be found using a query similar to `recent_pub_authors/2` above.

Both these views can be combined to find authors of conference or symposium publications between 1994 and 1997 which reside locally (where the more constrained definition of local is used):

```
recent_pub_authors(PubDBList, Authors) :-
  ( (lw(get, "http://www.cit.info/view1.html") *
     lw(get, "http://www.cit.info/view2.html")) +
    (+)<>PubDBList )#>cit(Authors, _, _, _, _, _).
```

The use of LW-intersection means that both definitions of `cit/6` must be satisfied for every author retrieved. `local/1` goals must satisfy the subgoals in the

clause bodies of both definitions of `local/1`. Thus, the more constrained definition of local dominates.

The views can be made to be context-dependent using the context operator. For example, the definition of `local/1` in the first view can be overridden by that in the local context:

```
local(WebLocation) :-
  contains(WebLocation, "http://localhost.cs.mu.oz.au/").

recent_pub_authors(PubDBList, Authors) :-
  ( ((#) + (lw(get, "http://www.cit.info/view1.html") / (#))) +
    (+)<>PubDBList )#>cit(Authors, _, _, _, _, _).
```

An advantage of LW-composition operators for querying and manipulating lightweight deductive databases is modularity. A view on a database can be derived by defining constraints in a separate program and appropriately composing them with existing programs. New views can also be composed from existing views.

In [8, 12, 41], the operator "in" is used for retrieving information from disparate databases, where each database is a (Horn clause) logic program with syntax similar to the language described in Section 2.2.3. This use of "in" is similar to how context switching is used above for retrieving information from different LogicWeb programs. This use of context switching allows different databases (e.g., with differing schemata) to be integrated, implementing the idea of *mediators* [12]. A mediator is a software component which allows inter-operation of different databases. Mediators form the "middle" layer between data from heterogeneous sources and users, allowing the data to be used without change and without users needing to know the underlying data format differences. The LogicWeb language allows mediators to be defined between lightweight deductive databases. Databases from remote sites which have been given a Web interface can also be similarly integrated using the LogicWeb language (see Section 6.5).

6.3 Knowledge-based Querying of Citation Databases on the Web

On-line citation information for a Computer Science department has to be kept up-to-date, and can be categorised in a number of ways (e.g., by author, conference, date of publication, research group, and topic). This section explores the use of lightweight deductive databases for organising citation information on the Web in order to facilitate maintenance and more sophisticated querying.

The use of lightweight deductive databases adds an extra dimension to query processing. Query processing for traditional databases usually consists of extracting the relevant data satisfying the query, and synthesising the results for the user. However, with lightweight deductive databases, the rules implementing a query may also involve finding and obtaining the relevant databases from the Web before data is extracted. The knowledge-based search technique described in Chapter 5 can be utilised for finding the relevant citation databases at query processing time.

6.3.1 Organising Citation Information on the Web

We shall assume that research in a Computer Science department is organised into sections. A section is divided into groups, and sections and groups are composed from projects. Each project consists of researchers. This follows the organisation of research in the Computer Science department at the University of Melbourne as of May 1996.

To represent the structure of a Web site, the notion of page types and relationships between page types are used as in Chapter 5. The page types used in the example are: `dept`, `research`, `project`, `project_members`, and `researcher`. These page types will be used for describing the strategies for searching the Web for relevant citation databases.

We shall assume that the relevant Web structure for a Computer Science department is as shown in Figure 6.1.[2] The departmental homepage (of type `dept`) has a link to a page (of type `research`) containing research information (in HTML). This information includes a description of the relationships between sections, groups, and projects, and links to pages describing projects (of type `project`). Each `project` page has a link to a page containing information about its members (of type `project_members`), and each `project_members` page has links to the members' homepages (which are of type `researcher`). The publication citations for each project are distributed in lightweight deductive databases which are accessible from the homepages of the researchers. This allows the citation databases to be separately maintained by authors.

Another advantage of distributed citation databases is that publications for the department or research groups can be obtained by combining the individual databases. This avoids redundancy that occurs when a department's publications page and an author's homepage store the same information, and ensures that group citation information is always up-to-date with respect to the individual databases. Queries are used to create the department's and a research group's publication list on-the-fly. These queries are written as a LogicWeb application and placed on the Web.

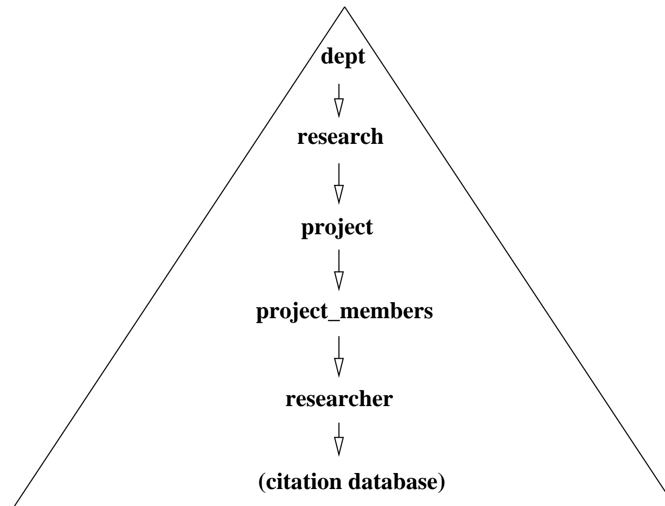
6.3.2 Searching for Citations

As an example, code that generates a research section's publication list is developed. Two approaches are possible:

- The URLs of all the relevant citation databases for each section can be stored in a pre-determined program, and the citations are queried using it. The `location_of/2` predicate of Section 6.2.1 is an example of how these

[2] The entire Web site of a Computer Science department would be much larger but only part of the site is needed in this application.

Figure 6.1 A representative diagram of the hypertext structure rooted at a departmental homepage. The arrows denote the sequence in which the various page types are reached, starting from the `dept` page.



URLs are stored.

- Starting from some pre-determined page, search the Web site for the URLs of the citation databases.

The first method requires the maintenance of a LogicWeb program containing the citation database URLs. For example, if a researcher leaves or joins a section, or if the URLs of pages change, then the program has to be modified. For this reason, the second method is preferred.

Finding the relevant citation databases consists of two main steps:

1. use the given section name to find the constituent project names; and

2. using those project names, search the Web site by following the sequence of page types shown in Figure 6.1, starting from the `dept` page, until the relevant databases are found.

The search starts from the `dept` page, rather than the `research` page, in order

to accommodate changes to the `research` page's URL. The URL of the `dept` page is least likely to change.

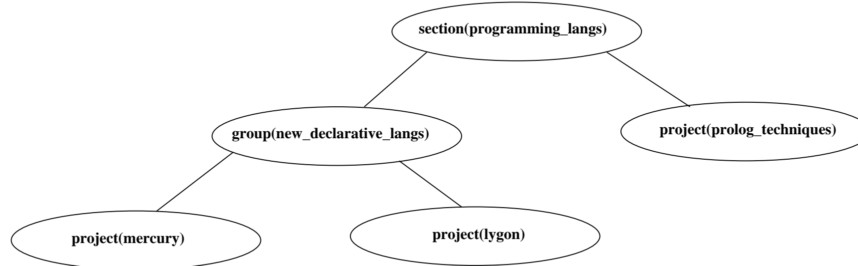
In order to implement the above strategy, the relationships between sections, groups, and projects (to carry out (1)) need to be formalised, and the Web structure specified (in order to do (2)).

6.3.3 Representing Knowledge

Representing Research Information. One way to find the projects contained in a given section is to parse the `research` page, and extract the required information. However, this would need to be done each time a query is processed. A more efficient alternative is to represent the knowledge on that page as logic programming facts and rules, and reason with them.

The relationships between sections, groups, and projects, are depicted as a hierarchy of concepts in Figure 6.2. This is a representative diagram; a typical department would have many more sections, groups, and projects.

Figure 6.2 A hierarchy of sections, groups and projects. The edges represent the `has_part/2` relationships.



The relationships are represented using `has_part/2` facts, which specifies how sections, groups, and projects are related. `contains_part/2` defines the transitive closure of the `has_part/2` relation.

```
has_part(section(programming_langs), group(new_declarative_langs)).
has_part(group(new_declarative_langs), project(mercury)).
```

```
has_part(group(new_declarative_langs), project(lygon)).
has_part(section(programming_langs), project(prolog_techniques)).

contains_part(X, Y) :- has_part(X, Y).
contains_part(X, Z) :- has_part(X, Y), contains_part(Y, Z).
```

Representing the Web structure. The sequence of page types in Figure 6.1 can be captured using `leads_to/2` facts:

```
leads_to(dept, research).
leads_to(research, project).
leads_to(project, project_members).
leads_to(project_members, researcher).
```

6.3.4 An Implementation of Citation Finding

A section name (e.g., "Programming Languages") can be selected by a user from a menu displayed on the browser. This is translated into a `section/1` term, and a goal like the following is generated:

```
get_citations(section(programming_langs), Citations)
```

Evaluation of the goal results in `Citations` being instantiated with a list of `pub_cit/6` facts. These are formatted and displayed by the browser.

`get_citations/2` uses the logic programming representation of the research information (`contains_part/2` and `has_part/2`) to retrieve all projects belonging to the specified section, and then collects the citations from those projects:

```
get_citations(Section, Citations) :-
  setof(project(Name), contains_part(Section, project(Name)),
        Projects),
  collect_citations(Projects, Citations).
```

`collect_citations/2` uses the Web structure information (coded as `leads_to/2` facts) and the project names to search over the Web site for

citation database identifiers. A LW-union of these databases is accessed to obtain the citations for the projects.

The definition of `collect_citations/2` is:

```
collect_citations(Projects, Citations) :-
  location_of(dept, DeptURL),
  setof(ProgramID,
        find_citpgm(dept, lw(get, DeptURL), Projects, ProgramID),
        ProgramIDs),
  retrieve_citations(ProgramIDs, Citations).
```

The call to `location_of/2` retrieves the URL of the `dept` page. The URLs of the citation databases are found by calling `find_citpgm/4` in a `setof/3` call. `find_citpgm/4` is similar to CIFI's `find_citation/4` described in Section 5.2, but is different in two main ways: (1) the page types and `leads_to/2` facts used in `find_citpgm/4` are those described in Section 6.3.3, and (2) `find_citpgm/4` returns the program identifier of a citation database as its result (instead of a citation). Project names in `Projects` are used to identify links to relevant `project` pages. Cue strings are used to select the links that would lead towards the citation databases. For instance, the word "Research" is a cue string for pages of type `research`:

```
cue_strings(research, ["Research"]).
```

The predicate `retrieve_citations/2` retrieves citations from the LW-union of the programs using code very much like that in Section 6.2.2.

The facts and rules described above would reside in a separate program at the Web site. This program is loaded first, and other pages and citation databases are loaded during query evaluation.

6.3.5 Summary

This example illustrates how the techniques of knowledge representation and automated Web searching are used to support queries on lightweight deductive

databases. Logic programming rules are used to represent a hierarchy of sections, groups, and projects, and to specify search behaviour over Web pages. The former maps a section name to its component project names, while the latter handles changes to the contents of the Web pages and the retrieval of relevant databases.

The concept hierarchy needs to be updated if a research section, group, or project, is added or removed. This can be done manually (especially if the change is small) or automatically by parsing the information on the `research` page into logic programming facts.

6.4 Generating Guided Tours

A *guided tour* organises Web pages by collecting thematically related pages together [198]. A page may be a member of more than one tour, with each tour providing a different view of the pages it uses. For instance, a guided tour of an on-line book on Malaysian cooking aimed at beginners would present a very different view of its component pages than a tour for experienced chefs.

A guided tour restricts and structures the links between pages. Hence, following a guided tour alleviates the "lost-in-hyperspace" problem one experiences when navigating through complex hypertext, such as the Web.

This section describes how guided tours can be generated on the Web by using logic programming techniques. The LogicWeb language is used to represent and query lightweight deductive databases containing information about pages, and to manipulate lists representing tour details. The pages of a guided tour are selected by querying the databases, and also by the dynamic extraction of information from Web pages themselves. In addition, code for generating tours, tour navigation, and tour visualisation, are each in their own programs.

The advantages of the LogicWeb approach for building guided tours are:

- *simplified tour generation*: tour generation is expressed as queries upon (com-

binations of) lightweight deductive databases. An arbitrary number of queries can be applied to the same database to generate different tours.

- *modularity*: as mentioned earlier, lightweight deductive databases have the advantage of reusability. Utilities to create, navigate, and display tours are also located in separate LogicWeb programs encouraging their reuse. For instance, a program for displaying tours can be used with different tours.

- *decentralisation and customisability*: the client-side generation of guided tours encourages decentralisation and customisability. This means that the pages which form a tour can come from many servers, in the same way as the databases. Also, local creation of tours allows them to be easily augmented or replaced by the user's own pages or databases.

The rest of this section is structured as follows. Section 6.4.1 describes the structure of a guided tour and the components of a LogicWeb application implementing a guided tour. Section 6.4.2 explores queries for generating tours. In Section 6.4.3, more complex tour generation is discussed. Section 6.4.4 outlines how user interaction with tours is implemented. Section 6.4.5 reviews other work on guided tours.

6.4.1 Structure of a Guided Tour and a Guided Tour Application

A guided tour consists of a sequence of nodes, and its index. Figure 6.3 shows a tour of Computer Science lecturers' homepages at the University of Melbourne. The index has links to each of the nodes in the sequence. Each node consists of the page's contents, and links to the index, the node on its left, the node on its right, the first node, and the last node. The first node has no left link and the last node no right link.

The index node of the tour is represented on-screen as shown in Figure 6.4. A node is represented on-screen as shown in Figure 6.5.

Figure 6.3 The structure of a guided tour of lecturers' homepages in the University of Melbourne Computer Science department. The arrows represent the links of the tour; the rectangles are Web pages.

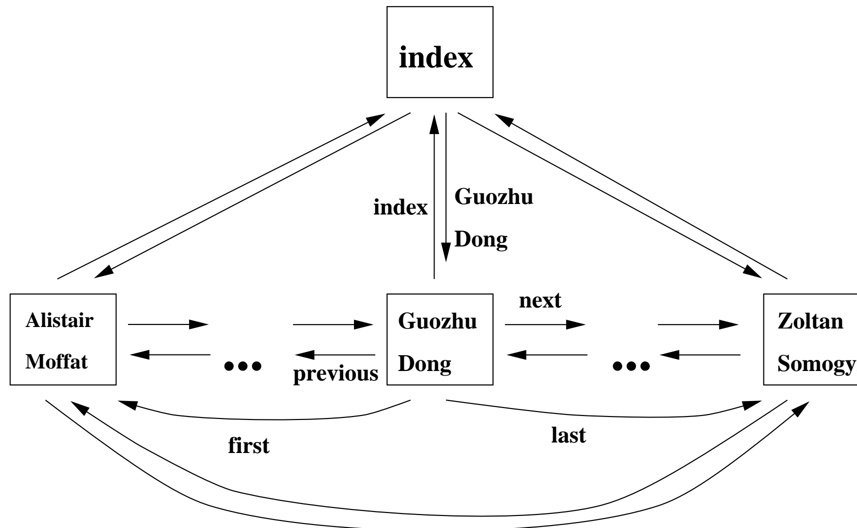


Figure 6.6 shows the structure of a guided tour application. A guided tour application consists of a set of LogicWeb programs:

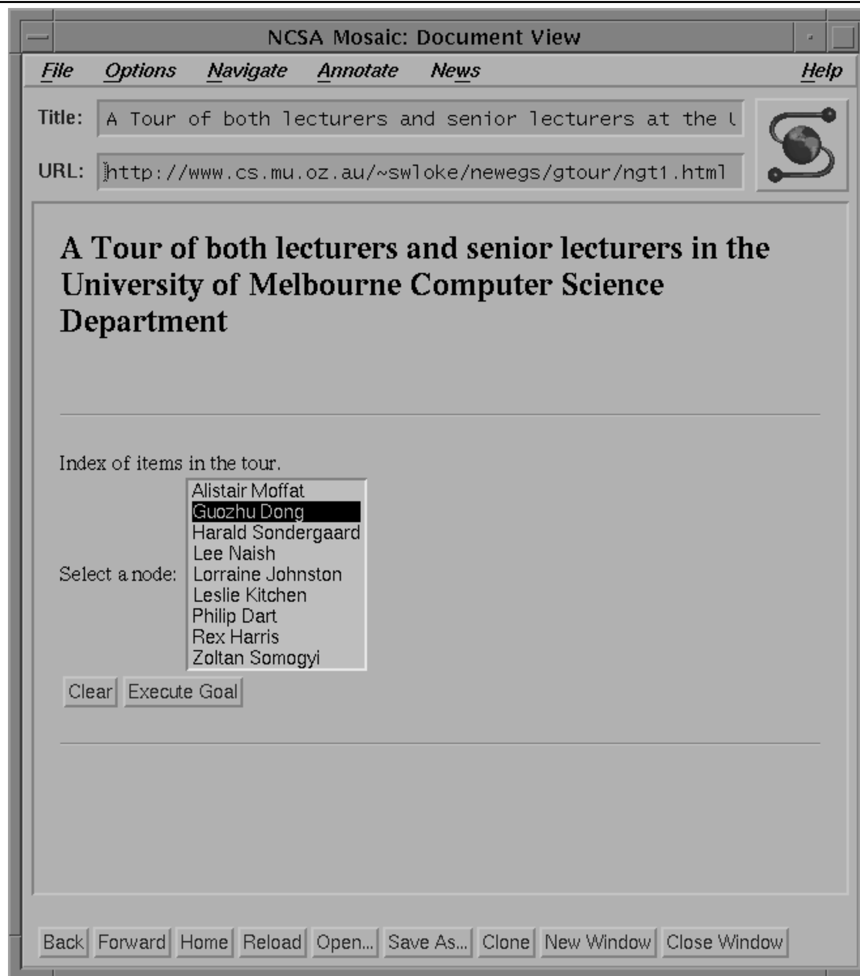
1. the main program defines the user interface to the guided tour and a predicate `generate_tour/1` which invokes a query to generate the sequence of tour nodes returning the sequence as its argument;

2. the tour utility program contains predicates to control navigation and display tour nodes and indexes; and

3. lightweight deductive databases are used by the main program when generating the tour list.

To start a guided tour application, the main program is downloaded, together with a utility program. The `interface/1` predicate in the main program calls predicates in the utility program to set up the tour and create the tour index:

Figure 6.4 The index node for the tour of lecturers' homepages.




```
interface(TourIndex) :-
   my_id(_, TourURL),
   lw(get, "http://www.cs.mu.oz.au/gtour/utility_program.html")#>
      set_up_tour(TourURL,
                  "A guided tour of lecturers and senior lecturers.",
                  TourIndex).
```

This definition of `interface/1` is different from the earlier examples in that the terms in its argument are dynamically generated instead of statically specified. To set up the tour, the utility program invokes `generate_tour/1` in the main

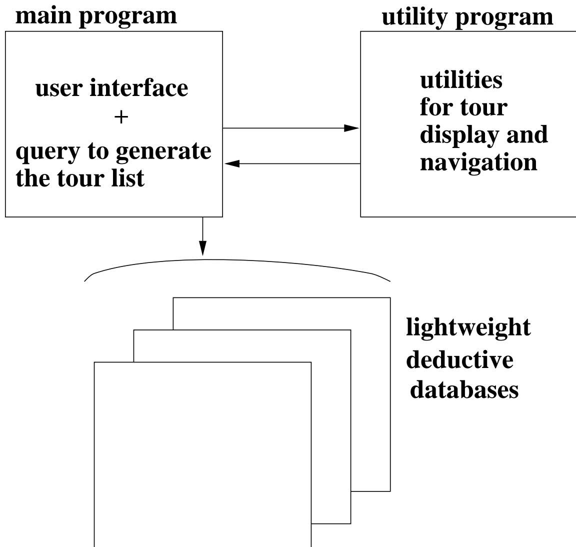
Figure 6.5 A typical tour node.

program. Evaluation of the tour-generation query causes lightweight deductive databases to be downloaded so that they can be utilised.

The sequence of tour nodes is stored as a Prolog list. Each element of the list is a `node/3` term containing the URL of the page, the type of tour node, and a string describing the node. A page type can be either `page` or `tour`. `page` means that the node represents a Web page, while `tour` means that the node corresponds to a subtour as explained in Section 6.4.3.2.

For the Computer Science lecturers example, the predicate

Figure 6.6 The guided tour application consists of a set of LogicWeb programs (represented by the boxes). An arrow indicates a “uses” relationship.



`generate_tour/1` is defined to generate the following list of nodes:

```
[node("http://www.cs.mu.oz.au/~alistair",page,"Alistair Moffat"),
 node("http://www.cs.mu.oz.au/~dong",page,"Guozhu Dong"),
 node("http://www.cs.mu.oz.au/~harald",page,"Harald Sondergaard"),
 node("http://www.cs.mu.oz.au/~lee",page,"Lee Naish"),
 node("http://www.cs.mu.oz.au/~ljj",page,"Lorraine Johnston"),
 node("http://www.cs.mu.oz.au/~ljk",page,"Leslie Kitchen"),
 node("http://www.cs.mu.oz.au/~philip",page,"Philip Dart"),
 node("http://www.cs.mu.oz.au/~rph",page,"Rex Harris"),
 node("http://www.cs.mu.oz.au/~zs",page,"Zoltan Somogyi")]
```

Such a list is generated by querying a page information database (or databases), as discussed in the next section.

6.4.2 Tour Generation

Tours can be generated using two kinds of information:

- *static*: this consists of predefined tour information, stored as lightweight deductive databases.
- *dynamic*: this kind of information is extracted from a page on-the-fly during tour generation.

Tour generation using both kinds of information is carried out below.

6.4.2.1 Tour Generation Using Static Information

The tour list in Section 6.4.1 was generated by querying page information stored in the following format:

staff(<u>user_name</u>,sname(first_name,last_name),academic_position)

For instance:

```
staff(ljj, sname("Lorraine", "Johnston"), lecturer).
staff(dong, sname("Guozhu", "Dong"), senior_lecturer).
```

Tour generation also utilises the `dept_server/1` fact to build absolute URLs:

```
dept_server("http://www.cs.mu.oz.au/").
```

The `staff/3` and `dept_server/1` predicates are stored in a LogicWeb program. Another program holds the predicates which construct a tour.

The top-level query for generating the list of nodes in the tour is defined in the body of `generate_tour/1`:

```
generate_tour(NodeList) :-
  staff_program_location(melbourne_cs, URL),
  setof(Node, lecturer(URL, Node), NodeList).
```

`staff_program_location/2` returns the URL of the LogicWeb program containing staff information when given the name of the department. `lecturer/2` is invoked in a `set_of/3` call in order to retrieve all the matching nodes.

`lecturer/2` builds a single `node/3` term:

```
lecturer(ProgramURL, node(HomePageURL, page, NameString)) :-
  lw(get, ProgramURL)#>( staff(UserName, Name, lecturer)
                       ; staff(UserName, Name, senior_lecturer)
                       ),
  lw(get, ProgramURL)#>dept_server(ServerURL),
  make_homepageurl(ServerURL, UserName, HomePageURL),
  make_namestring(Name, NameString).
```

The disjunction of `staff/3` goals allows the final tour to contain both lecturers and senior lecturers.

`make_homepageurl/3` constructs the URL of a homepage from the departmental server's URL (contained in `dept_server/1`), and the username of the staff member.

`make_namestring/2` creates a string from the elements of the `sname/2` term in a `staff/3` fact.

6.4.2.2 Domain Knowledge as Static Information

Tours can be formulated by using additional domain knowledge which permits tours to be created which could not be formulated by looking at the text of a page

alone. For example, assume that the Computer Science staff database also contains information about teaching positions:

```
teaches(professor).
teaches(associate_professor).
teaches(senior_lecturer).
teaches(lecturer).
teaches(tutor).
```

These additional details allow the generation of a tour of teaching staff homepages by replacing `lecturer/2` with `teaching_staff/2` in the top-level query:

```
generate_tour(NodeList) :-
  staff_program_location(melbourne_cs, URL),
  setof(Node, teaching_staff(URL, Node), NodeList).
```

`teaching_staff/2` uses `teaches/1` as a filter so that only staff with teaching positions are selected for the tour:

```
teaching_staff(ProgramURL, node(HomePageURL, page, NameString)) :-
  lw(get, ProgramURL)#>
          (staff(UserName, Name, Position), teaches(Position)),
  lw(get, ProgramURL)#>dept_server(ServerURL),
  make_homepageurl(ServerURL, UserName, HomePageURL),
  make_namestring(Name, NameString).
```

6.4.2.3 Tour Generation Using Dynamic Information

Dynamic information is obtained from the meta-information of a page, or from the page itself via parsing.

Using Meta-information. The following example uses meta-information to generate a tour of senior lecturers' homepages which have been modified since 1st November 1996.

The top-level query will now invoke `updated_senior/2` in its `setof/3` call:

```
updated_senior(ProgramURL, node(HomePageURL, page, NameString)) :-
  lw(get, ProgramURL)#>staff(UserName, Name, senior_lecturer),
  lw(get, ProgramURL)#>dept_server(ServerURL),
  make_homepageurl(ServerURL, UserName, HomePageURL),
  modified_since(HomePageURL, '1996;11;01;00:00'),
  make_namestring(Name, NameString).
```

`modified_since/2` compares the last modified time of a page with the given date.

```
modified_since(URL, Date) :-
  lw(head, URL)#>about(last_modified, LastModifiedTime),
  getime(LastModifiedTime, Date).
```

`getime/2` compares two dates and is described in Appendix B.

Using the Page Contents. Empirical evidence suggests that the homepages of academics contain the word "student" when they talk about their graduate students. This observation is used as the basis for a tour of academic home pages which discuss graduate students.

As before, the top-level tour generation query must be modified to call a different predicate in its `setof/3` call. For this tour, the predicate is `academic_with_student/2`. In addition, the staff program must contain `academic_position/1` facts which define academic positions (e.g., lecturer, professor). `academic_with_student/2` is:

```
academic_with_student(ProgramURL, node(HomePageURL, page,
                                       NameString)) :-
  lw(get, ProgramURL)#>(staff(UserName, Name, Position),
                        academic_position(Position)), % only academics
  lw(get, ProgramURL)#>dept_server(ServerURL),
  make_homepageurl(ServerURL, UserName, HomePageURL),
  lw(get, HomePageURL)#>h_text(Source),
  contains(Source, "student"),            % page mentions "student"?
  make_namestring(Name, NameString).
```

`contains/2` is used here to check whether an academic homepage includes the string "student".

Similarly, a tour of academics' homepages with their images can be defined:

```
academic_with_imageapplets(ProgramURL, node(HomePageURL, page,
                                             NameString)) :-
  lw(get, ProgramURL)#>(staff(UserName, Name, Position),
                        academic_position(Position)), % only academics
  lw(get, ProgramURL)#>dept_server(ServerURL),
  make_homepageurl(ServerURL, UserName, HomePageURL),
  lw(get, HomePageURL)#>image(_, ImageURL),
  contains(ImageURL, UserName),
  make_namestring(Name, NameString).
```

The above rule assumes that the URL (or filename) of an academic's image contains the academic's username, and no other image URLs on the homepage contain the username.

6.4.3 More Complex Tour Generation

Tours can be built by queries which combine multiple databases using LW-composition operators in the way shown in Section 6.2.

Other ways of constructing tours are possible. Since the result of a tour-generation query is a list of terms, the results of different queries can be easily combined to form new tour lists. Moreover, a list node can also represent a sub-tour, so allowing tours to be nested inside others. Tour creation can also be customised using user inputs. Examples of these methods for constructing tours are given below.

6.4.3.1 Tours by Appending Lists

Suppose that the Computer Science department at the University of Queensland has a program containing information about staff homepages. This can be used to construct a tour of lecturers' homepages in that department which can be combined with one for the department at the University of Melbourne. The staff database at the University of Queensland contains the same information as the

University of Melbourne database, but does not conform to the schema given in Section 6.4.2.1. This means that code to translate the staff information from the two universities into `node/3` terms will be different. Hence, the tour is constructed by combining the tours obtained from each programs separately.

The tour of the University of Melbourne lecturers is created using `lecturer/2` (as defined in Section 6.4.2.1). But for the tour of the University of Queensland lecturers, `que_lecturer/2` is used to translate their staff information into `node/3` terms.

The top-level tour query will retrieve the locations of both programs, create two tours using `lecturer/2` for the University of Melbourne lecturers and `que_lecturer/2` for the University of Queensland lecturers, and append the results together:

```
generate_tour(NodeList) :-
  staff_program_location(melbourne_cs, MelURL),
  staff_program_location(queensland_cs, QueURL),
  setof(Node, lecturer(MelURL, Node), MelNodeList),
  setof(Node, que_lecturer(QueURL, Node), QueNodeList),
  append(MelNodeList, QueNodeList, NodeList).
```

The list representation allows tours to be combined, sorted, or filtered using standard list manipulation predicates.

6.4.3.2 Tours Containing Tours

It is possible to build a guided tour, where some or all of the nodes are tours themselves [100].

We consider a tour where the nodes are tours of professors from the Computer Science departments at three universities: Melbourne, Queensland, and Canberra. This tour is represented by the list:

```
[node("http://www.cs.mu.oz.au/prof_tour.html",tour,
      "Professors at the University of Melbourne"),
 node("http://www.cs.uq.edu.au/prof_tour.html",tour,
      "Professors at the University of Queensland"),
 node("http://beth.canberra.edu.au/prof_tour.html",tour,
      "Professors at the University of Canberra")]
```

The URL in each `node/3` term is the location of the program containing the query which generates that particular tour's nodes.

The subtour functionality extends the linear tour structure shown in Figure 6.3 to allow each node to be an index node, with links to its subtour nodes. In setting up a tour, `set_up_tour/3` in the utility program is invoked recursively on each node of type `tour`. For example, with the information in the first node above, `set_up_tour/3` is invoked to generate the index for the subtour:

```
?- set_up_tour("http://www.cs.mu.oz.au/prof_tour.html",
               "Professors at the University of Melbourne",
               SubTourIndex).
```

6.4.3.3 Tours Constructed Using User Inputs

So far, the tours have been generated by pre-defined queries over databases. Here, we look at an example where the query depends on inputs from the user. The example is an application for planning a tour of sites in four Asian countries.

The application starts by querying the user for the number of days he/she wishes to spend in each country, and for his/her interests (e.g., "nature", "culture", or others). Figure 6.7 shows the application's user interface. Using the user's inputs and information from a tourist site database, the application constructs a guided tour of sites. Figure 6.8 shows a tour constructed with the inputs of 3 days in Thailand, 4 in Malaysia, 3 in Indonesia, and 4 in Taiwan, and "nature" and "culture" as interest areas. For each tourist site, the name, location, category, and the suggested time to spend at the site are displayed.

The tourist site database contains details about a number of tourist sites in the following format:

Figure 6.7 The user interface to the Asian tour planner.

NCSA Mosaic: Document View

File Options Navigate Annotate News Help

Title: Tour Planner Application

URL: http://www.cs.mu.oz.au/~swloke/newegs/gtour/tp.html

Tour Planner

Enter the number of days you would like to spend in each country:–

- Thailand: 3
- Malaysia: 4
- Indonesia: 3
- Taiwan: 4

Indicate your interests:–

- nature:
- culture:
- besides nature and culture:

Clear Execute Goal

Back Forward Home Reload Open... Save As... Clone New Window Close Window

Figure 6.8 A guided tour constructed with sites in Thailand, Malaysia, Indonesia, and Taiwan, for 3, 4, 3, and 4 days respectively.

site(<u>site_id</u>,name,country,state,category,days_to_spend)

An instance of the schema is:

```
site(bot, 'Borobudur Temple', 'Indonesia', 'Java', culture, 1).
```

This fact represents a site in Java, Indonesia, which comes under the culture category. A visitor is recommended to spend one day at the site.

On receiving the user's inputs, the database is queried for sites to make up a guided tour. For each country, a number of sites is chosen, where each chosen site's category matches the user's interests and the *days_to_spend* values for the sites sum up to what the user has specified for that country. The procedure for choosing the sites is captured by the rule below, which takes a country, the user's interests, and the number of allocated days, and returns a list of tourist site IDs:

```
country_sites(_Country, _UserInterests, 0, SiteList, SiteList).
country_sites(Country, UserInterests, MaxDays, SiteList,
              NewSiteList) :-
  MaxDays > 0,
  site(Id, _, Country, _, Category, DaysForSite),
  member(Category, UserInterests),
  not(member(Id, SiteList)),    % check if site already  chosen
  MaxDaysLeft is MaxDays - DaysForSite,
  country_sites(Country, UserInterests, MaxDaysLeft, [Id|SiteList],
                NewSiteList).
```

The required sites are first obtained by invoking `country_sites/5` for each country using a goal like:

```
?- country_sites('Indonesia', [nature, culture], 3, [], SiteList).
```

Then, `node/3` terms similar to those used in the tours discussed earlier are generated representing the tour over all the countries, and the tour is rendered as shown in Figure 6.8.

Each time a tour node (i.e., a tourist site) is visited, a page is retrieved from a tourist information Web server using the site's name, country, and state, and information is extracted from the page and displayed together with the tour links like those in Figure 6.5.

6.4.4 Implementation

In the utility program (see Figure 6.6), after a list of `node/3` terms has been generated, each term of type `page` is converted into a `node/7` fact of the form:

```
node(ID,URL,IndexNodeID,page,Description,PreviousNodeID,NextNodeID)
```

The fields in `node/7` are: the node's ID, the URL of the page, the ID of its index node, its type (either `page` or `tour(_)`), its description as contained in the original `node/3` term, and the IDs of the nodes to its left and right.

The node ID is necessary because the same page can be used in several tours (e.g., in a tour of tours), which means that its URL cannot be used as a unique node identifier.

For each `node/3` term of type `tour`, the program which generates the subtour is downloaded, and the subtour's `node/3` terms created. Then, a `node/7` fact like that above but with the type `tour(Index)` is asserted. `Index` is a list of strings which can be used as items in the index menu, as shown in Figure 6.4. This list of strings are the descriptions of the subtour's nodes as contained in the subtour's `node/3` terms. The subtour's `node/3` terms are then translated into `node/7` facts in the same way.

Representing tours as lists makes them easier to modify and combine. However, converting them to facts makes the rules which compute tour links much simpler. This simplification is a result of storing more information in each `node/7` fact than in the corresponding `node/3` term. For example, retrieving the destination node of the "next" link is simply a matter of retrieving the relevant `NextNodeID`.

The user interacts with the tour via the browser. The index node becomes the first page shown to the user (see Figure 6.4). User selection of tour links invokes the `do_query/2` predicate in the main program which, in turn, invokes predicates in the tour utility program. For example, when the user selects a tour link (e.g. "next"), a query is passed via the main program to the utility program in the

Prolog engine. The next node is computed and the corresponding page is sent to the browser to be displayed as shown in Figure 6.5.

A tour node's page is prepared by first determining the node's links, and then constructing a HTML document consisting of the page contents and the link information. The document is displayed by invoking `display_page/2`.

6.4.5 Other Work on Generating Guided Tours

Beynon-Davies *et al* [26] use semantic database management systems to implement their hypermedia systems. Their system enables guided tours to be created by querying databases in a similar way as the LogicWeb-based guided tours. Their databases contain typed binary relationships between entities, where each entity consists of a triple containing object and type descriptions. Pre-defined relationships include IS-A, A-KIND-OF, HAS-A, and PART-OF. Their query language allows the retrieval of database entities and attributes, but does not allow the specification of rules. In contrast, LogicWeb-based guided tours utilise Prolog-based databases allowing rules to be stored in addition to relations and a rule language as the query language. Also, the guided tours are Web-based, whereas their system is neither networked nor distributed.

In Nicol *et al* [157], a tour is represented by a list of URLs in a file. In Hauck [100], each page in a tour is given an additional URL naming it in the tour. Both of these projects use server-side mechanisms which fetch the tour pages, and augment them with tour links before sending them to the client. In contrast, the LogicWeb approach is client-based: all the tour components (the databases, the tour generation rules, and the tour navigation and display utilities) are downloaded in separate programs, and executed on the client's machine.

The server-side approaches assume that the nodes of a tour have been specified manually. The LogicWeb approach generates tours using static and/or dynamic information.

6.4.6 Summary

This section has presented a logic programming approach to constructing guided tours in the World Wide Web. This approach has the following advantages:

- Tour creation can be expressed in terms of logic programming queries, allowing tours to be specified in a high-level fashion without the need for manual processing.
- Tour generation is more expressive due to the use of information stored in lightweight deductive databases, and the ability to dynamically extract information from pages by parsing.
- By representing tours as logic programming lists, they are easier to change and manipulate. For instance, tours can be sorted, filtered, or combined with other tours using standard list processing predicates.
- Due to the use of LogicWeb programs, tour components are reusable, customisable, and easily available.
- All processing is client-based, thereby reducing server load.

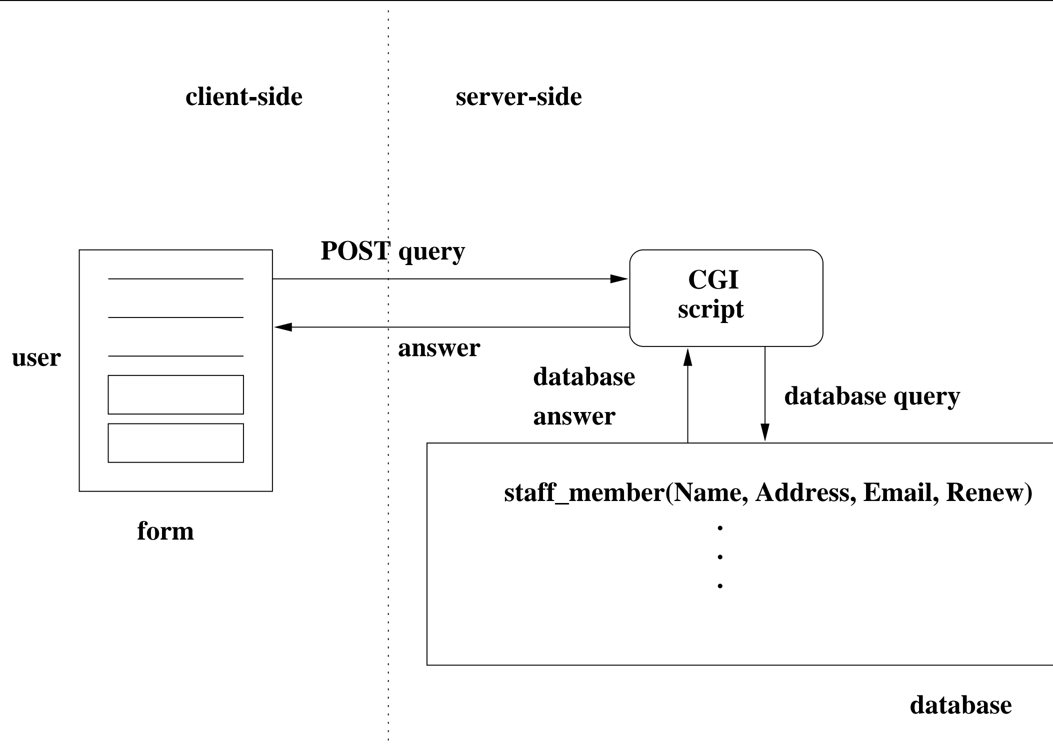
This approach requires the user to write tour generation rules. However, these rules are reusable and enable the precise specification of a tour.

6.5 Server-side Databases

The LogicWeb system's use of client-side processing for database manipulation can be a disadvantage since a database must be downloaded to the system before it is evaluated. This reduces the server-side load of using the database, but there are still many reasons why the processing might be restricted to the server-side. For instance, the database may be too large to be easily moved over the Web, or it

may contain confidential information that should not be made universally available. Commercial reasons may mean that the database cannot be freely sharable. Also, having a single, central database makes issues such as transaction control and maintaining a consistent state easier. This subsection discusses LogicWeb's mechanism for accessing such server-side databases.

Figure 6.9 A server-side database and its interface.



A typical server-side database and its interface is represented in Figure 6.9. A user poses a query to the database via a form on a Web page available from the database site. The form details are transmitted to a server-side CGI script which is named within the form. In the following discussion, the script is assumed to be located at

`http://www.cs.mu.oz.au/cgi-bin/db-query`

The form details are encoded using the HTTP POST method in the way described

in Section 2.1.2.3. These are read by the CGI script which converts them into a query suitable for the database. The script also converts the database answer into an appropriate Web format (usually a Web page) which is sent back to the client.

In Figure 6.9, the database is assumed to contain Prolog facts of the form:

```
staff_member(Name, Address, Email, ContractRenewalDate).
```

For example:

```
staff_member(name("Guozhu Dong"), address("University of Melbourne"),
             email("dong@cs.mu.oz.au"), renew(november, 2000)).
```

The forms interface contains four fields labelled with “Name”, “Address”, “Email”, and “Renew”. The fields can be filled in or left blank (with the value “none”). These field names and values are converted by the CGI script into suitable arguments in a goal, and applied to the database. After the database engine has evaluated the query, the script converts the results into a Web page for the user.

A LogicWeb program can interact with the server-side database using context switching. For the scenario outlined above, a possible query would be:

```
lw( post([field("Name", "none"),
          field("Address", "Univ. of Melbourne"),
          field("Email", "none"),
          field("Renew", "none")
         ]),
    "http://www.cs.mu.oz.au/cgi-bin/db-query" )#>
                        staff_member(Name, _, Email, renew(_, 2000)).
```

The `lw/2` term can be viewed as a *specification* of the program containing the query results against which the `staff_member/4` goal will be evaluated. In this case, the retrieved program will contain all the staff members from the University of Melbourne, and the goal will extract the name and e-mail address of someone who should renew during 2 000 (through backtracking all the Melbourne people in this situation can be collected).

The server-side database may not be in the form of Prolog facts, but could be any database system such as a relational database system which has been given a Web interface (and so, query-able via GET or POST requests). A LogicWeb program can be written which parses and converts the query results into Prolog facts of a specific format.

6.6 Related Work

6.6.1 Database on Web Pages

Dobson *et al* [74, 75] utilise additional HTML tags to embed relational databases (called lightweight databases) in Web pages. Essentially, an entity-relationship diagram is mapped onto the hypertext structure of the Web. Relationships between entities on different pages are specified by hypertext links, with attributes defining the relationships. Lightweight databases have been used to generate HTML documents, for indexing, and for implementing databases which spread over several servers.

Lightweight deductive databases are motivated by their use of relational databases. However, lightweight deductive databases provide more powerful modelling capabilities because of rules. LogicWeb operators are also present for composing lightweight deductive databases.

Sandewall [175] proposes the World-Wide Data Base, where a database consists of downloadable short text files, each file containing an object description. An object consists of properties, represented in a specialised language, and can reference other database files (i.e., objects), or HTML pages. Other object-oriented concepts, such as message-passing, are not employed. The main application discussed is HTML page generation, where objects store resources for the generation of pages. In particular, values of properties may be scripts (in LISP) which specify how to generate HTML expressions.

The databases in this chapter are deductive, whereas theirs are object-based. However, their objects can be represented in our language. Also, they do not provide a uniform query language for their objects database, while the LogicWeb language is used for that purpose.

Their use of one file per object may incur heavy network transmission costs. Lightweight deductive databases can contain multiple relations.

6.6.2 Standardised Knowledge-bases on the Web

Boley [30] proposes the use of Horn clauses to represent knowledge-bases on the Web, and outlines issues in building a server-side search engine for querying them in the style of AltaVista. LogicWeb provides the framework for building specific applications on the Web using logic programming, whereas Boley is proposing a standardised repository of knowledge in Horn-logic. The LogicWeb language would be well-suited for building applications which manipulate such knowledge and interact with Boley's search engine, since the search engine's results will be in a subset of the LogicWeb language.

Boley also suggested the use of Web-based Horn-logic knowledge-bases as *distributed corporate memory*, i.e. an organisation's knowledge-base residing on an Intranet (the organisation's network). The use of lightweight deductive databases in an Intranet setting is an interesting area for future work. The LogicWeb language can be used both to represent the knowledge-bases and to build the applications which utilise them.

6.6.3 Marked-up Text on the Web

There has been much recent work on adding machine-processable data to the Web in the form of marked-up text.

The recent HTML 4.0 standard [171] contains the META tag for meta-data based around name-value pairs. The Extensible Markup Language (XML) [110]

is a mark-up language extending HTML by providing the ability to define new tags and new attributes for tags. The definitions are linked from the document itself. The advantage of XML is to allow tags to be defined which are tailored to an application. For example, more structure can be imposed on data which needs to be processed automatically.

These mark-up languages prescribe tags or allow the definition of new tags but do not specifically prescribe the information to be included using such tags. Mark-up tags have been used to include specific attributes to improve page indexing such as the Dublin Core meta-data set [208], ontology descriptions in the form of IS-A class hierarchies and instance-instance relationships (e.g., a subsection can be related to another page by a named relationship) [134], and information for generating code in languages such as C/C++, Java, and Visual Basic [207]. The World Wide Web Consortium's Resource Description Language (RDL) [189] is being developed as a data model for encoding meta-data. The preliminary RDF specification describes a data model consisting of nodes (representing Web resources such as pages) and properties of nodes stated as attribute-value pairs. Tags are used to include information conforming to this data model. RDF uses XML encoding as its syntax. RDL is intended to be used for a variety of applications such as encoding site-maps, search engine data collection (to support page indexing), and distributed authoring.

This chapter proposes the inclusion of machine-processable data in the form of LogicWeb rules on Web pages. LogicWeb rules have a simple syntax which is no more complex than tagged information, and is no less expressive than the above tag-based systems. For example, Dublin Core meta-data, ontology descriptions, and RDL's data model can be encoded in rules, and manipulated directly without parsing tags. For simple forms of data (e.g., attributes and values), tags would suffice. But as mentioned in the introduction, Prolog rules have proven their utility for modelling and reasoning over complex data (e.g., in expert systems). In addition, data encoded as LogicWeb rules on different pages

can be combined using techniques with a solid semantic basis.

6.6.4 Comparison with Deductive Database Systems

Lightweight deductive databases uses Prolog with LogicWeb extensions. However, existing deductive database systems [116, 98] differ from Prolog systems in several ways, including:

- *Query optimisation.* Query processing often finds all answers to the query, i.e. the "set at a time" paradigm is more efficient than the "tuple at a time" paradigm of Prolog. To facilitate this, optimised bottom-up evaluation is often used, rather than the top-down evaluation of Prolog systems.

- *Restrictions on rules.* The rules in deductive database systems are range-restricted, i.e. all variables that appear in the head of the clause must appear in the body. This implies that all facts must be ground. This removes the need for full unification, thereby increasing efficiency.

  Another common restriction is that all terms in the program are variables or constants. This ensures that logical entailment is decidable. A logic programming language with this restriction is Datalog [116].

A major difference between the LogicWeb system and existing deductive database systems is that support for remote updating of lightweight deductive databases is lacking. If updates were possible, transaction processing, and concurrency control would also have to be available. Updates have not been considered because the main focus of lightweight deductive databases has been on using the Web to disseminate structured information.

6.6.5 Knowledge-based Access to Information

Knowledge-bases have been used to facilitate (e.g., to speed-up) access to information.

The Information Manifold [113, 112] is a system for building a knowledge--base representing the user's interests. This uses a combination of Horn rules and the CLASSIC knowledge representation language to describe information sources, and taxonomy relationships among them. The knowledge-base is also used to process the results of searches submitted to multiple Web index servers.

LogicWeb supports knowledge-based query processing. Users can build their own knowledge-base and query processing rules, perhaps on top of those provided by information servers.

Web pages have been generated from knowledge-bases by using user profiles [102], and user queries [82]. A similar functionality can be achieved when the results of lightweight deductive database queries are Web documents.

Barcaroli *et al* [19] represents hypertext at a Web site using a knowledge-base. Their system answers user queries by returning a sequence of links leading to the page containing the answer. With lightweight deductive databases, the information provider can provide a similar capability by mapping user queries to appropriate URLs.

These approaches typically make use of knowledge representation languages based on description logics in order to represent concept models. The LogicWeb language extends Horn clauses since Horn-logic has been shown to be a versatile and expressive data modelling language for both AI programming and deductive databases [185, 116].

Context logic, an extension of first order logic where sentences are true with respect to a given context, has been used to integrate databases [84]. Axioms are written which lift sentences from several contexts into a common one. This is similar to the idea of mediators discussed in Section 6.2.

6.7 Summary

The Web should be enhanced with richer and more machine-processable information content, as argued in [24, 23]. LogicWeb rules provide a sufficiently structured, readable, expressive, and high-level representation formalism for Web data modelling and querying.

As illustrated using an example on citation information, lightweight deductive databases can be queried based on attributes and can be separately maintained. At query evaluation, relevant databases are located using heuristics-guided search and combined using well-established techniques from the areas of deductive databases and LW-composition operators. These databases take advantage of the Web as a widely used and cost-effective transport mechanism and exploit client-side processing.

Familiar techniques from compositional logic programming can be employed not only to combine databases but also to structure LogicWeb applications (of which lightweight deductive databases may be part), such as the guided tour application depicted in Figure 6.6. As mentioned in Chapters 2 and 3, program structuring ideas such as object-oriented and contextual logic programming can be modelled using the LogicWeb operators, allowing LogicWeb applications to be built based on these ideas.

Chapter 7

Extending the Semantics of Web Links

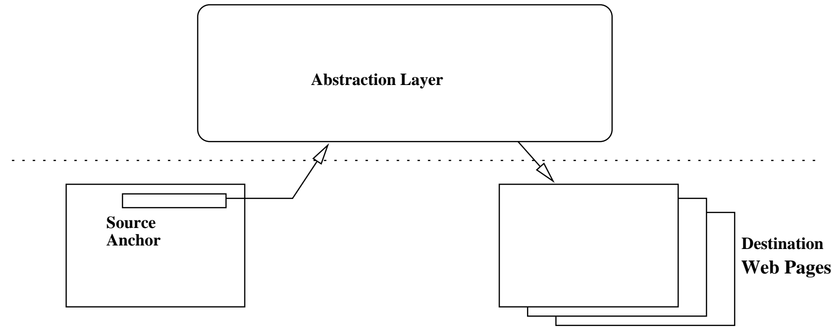
The Web's link mechanism is uni-directional, with fixed source and destination pages. Normally, selecting a link on a Web browser simply retrieves a page and displays it (as mentioned in Chapter 2). This chapter employs LogicWeb to extend the expressiveness of links, whose semantics are captured using rules.

A LogicWeb-based conceptualisation of the Web as a two-layered hypertext model is presented: Web pages are at the first level, but the links between them pass through a link abstraction layer. The abstraction layer can take many forms, such as a semantic network, concept hierarchy, or simply procedures to compute destination pages. The computation associated with a link is the link's *behaviour*.

Logic programming is explored for coding the link abstraction layer, exploiting its ability to represent declarative structures in a concise and readable manner, its nondeterminism, and its dynamic database update mechanism.

This chapter is organised as follows. The two-level Web model is described in Section 7.1. Section 7.2 illustrates links which utilise structured information such as semantic networks and databases, Section 7.3 illustrates links which cope with the dynamic and unpredictable nature of the Web, and Section 7.4 illus-

Figure 7.1 The two-level Web model. A link abstraction layer separates the source and destination of links.



trates links whose behaviour is dependent on state, such as the history of previous link selections. In Section 7.5, modular approaches to building complex link behaviours are described, where link behaviours are encoded in separate LogicWeb programs and combined as required. Section 7.6 describes how the LogicWeb system is extended to allow the user to specify link behaviours which are applicable to all pages. Section 7.7 discusses related work.

7.1 The Two-level Web Model

Figure 7.1 depicts the two-level Web model: the first level contains Web information (e.g., Web pages, multimedia data), and the second level determines the meaning of links (the link abstraction layer).

A link in the existing Web is uni-directional, have a fixed configuration, and go from one source page to one destination page. Links at the abstraction layer can be bi-directional, dynamically configured, and go from many sources to many destinations.

The link abstraction layer can implement many link formalisms. Two key aspects that logic programming is particularly suitable for are:

- *structured information representation*: as mentioned in Chapters 1 and 6, logic

programming can represent structured information in the form of deductive databases, or knowledge-based structures. Such structures can be used to create abstract views of the Web or specify page meta-data, i.e. data about pages, which will help improve a user's understanding of the Web, and facilitate linking based on semantic criteria.

- *nondeterminism*: the nondeterminism in logic programs allows links to deal with the dynamic and unpredictable nature of the Web.

In addition, Prolog's non-logical `assert/1` and `retract/1` for dynamic database manipulation can be used to implement *history-based linking*, where state information is maintained.

7.2 Utilising Structured Information for Linking

This section shows how various kinds of structured information (e.g., semantic networks, databases) can be implemented in the link abstraction layer. These structures can be pre-defined by the programmer, or be created dynamically by extracting details from Web pages.

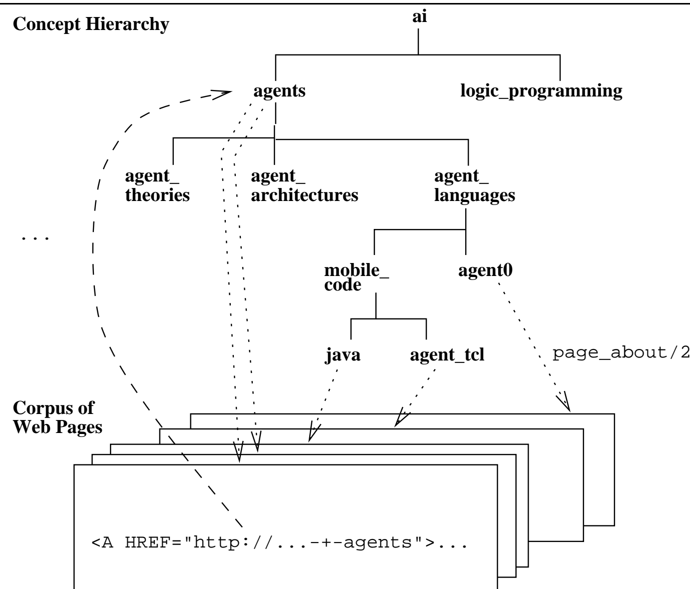
7.2.1 IS-A Hierarchy Links

The *IS-A hierarchy* about AI, logic programming, and agents described here is loosely based on the Canto hypertext data model [156]. Its overall structure is shown in Figure 7.2.

`isa/2` represents an IS-A relation between concepts in the form of `isa(Subconcept, Concept)` links:

```
isa(logic_programming, ai).
isa(agents, ai).
isa(agent_theories, agents).
isa(agent_architectures, agents).
```

Figure 7.2 An IS-A hierarchy. The dashed arrow shows a link referring to a concept in the hierarchy. The dotted arrows show mappings from concepts to Web pages as specified by `page_about/2`.




```
isa(agent_languages, agents).
isa(mobile_code, agent_languages).
isa(java, mobile_code).
isa(agent_tcl, mobile_code).
isa(agent0, agent_languages).
```

To build an IS-A hierarchy, the transitive closure on `isa/2` is defined:

```
trans_isa(Concept1, Concept2) :-
  isa(Concept1, Concept2).
trans_isa(Concept1, Concept2) :-
  isa(Concept1, Mid),
  trans_isa(Mid, Concept2).
```

URLs are associated with concepts using `page_about/2`:

```
page_about(agents,
  "http://machtig.kub.nl:2080/infolab/egon/Home/agents.html").
page_about(agents, "http://www.cs.mu.oz.au/~swloke/res1.html").
page_about(java, "http://java.sun.com/").
```

```
page_about(agent_tcl,
  "http://www.cs.dartmouth.edu/~agent/agenttcl.html").
page_about(agent0,
  "http://www.scs.ryerson.ca/~dgrimsha/courses/cps720/agent0.html").
   :  % more page_about/2 facts
```

`trans_about/2` defines a transitive version of `page_about/2` which associates a concept with a URL if one of its sub-concepts is associated with that URL:

```
trans_about(Concept, URL) :-
  page_about(Concept, URL).
trans_about(Concept, URL) :-
  trans_isa(SubConcept, Concept),
  page_about(SubConcept, URL).
```

Note that the pages come from different servers. The IS-A structure defines an abstract view over these pages classifying them by topic and virtually integrating information from different servers.

The IS-A structure is stored in the page:

```
http://www.cs.mu.oz.au/~swloke/link_kb.html
```

Note that the IS-A structure (if large and complex) may be decomposed into modules (using partitioning techniques described in [83]), where each module is a relatively small hierarchy stored in its own page and each such module is a node in a hierarchy of modules.

An IS-A query will be phrased as a syntactic extension of the above URL:

```
http://www.cs.mu.oz.au/~swloke/link_kb.html-isa-<concept-name>
```

A URL of this type will be understood to mean: retrieve a page associated with that `concept-name` in the IS-A hierarchy. For example, a page about agents can be obtained by dereferencing the anchor:

```
<A HREF="http://www.cs.mu.oz.au/~swloke/link_kb.html-isa-agents">
on software agents</A>
```

This behaviour is implemented by redefining the meaning of `link_action/1`, which is invoked when a link is activated. `link_action/1` must extract the concept from the string passed to it, search the IS-A hierarchy for a suitable URL, and then display that page:

```
link_action(HREFString) :-
  link_parts(HREFString, ISA-URL, Concept),
  lw(get, ISA-URL)#>page_about(Concept, ConceptURL),
  my_id(_, MyURL),
  ConceptURL \= MyURL,  % ensure that a different page is fetched
  show_page(ConceptURL).


link_parts(HREFString, ISA-URL, Concept) :-
  append(ISA-URL, [0'-,0'i,0's,0'a,0'-|ConceptStr], HREFString),
  atom_chars(Concept, ConceptStr).   % convert string to atom


show_page(URL) :-
  lw(get, URL)#>h_text(Src),
  display_page(URL, [data(Src)]).
```

`link_action/1` uses `link_parts/3` to split the URL string into the URL for the IS-A hierarchy page and the concept. A suitable URL related to the concept is obtained by calling `page_about/2`, and the corresponding page is displayed with `show_page/1` (but only if it is different from the current page).

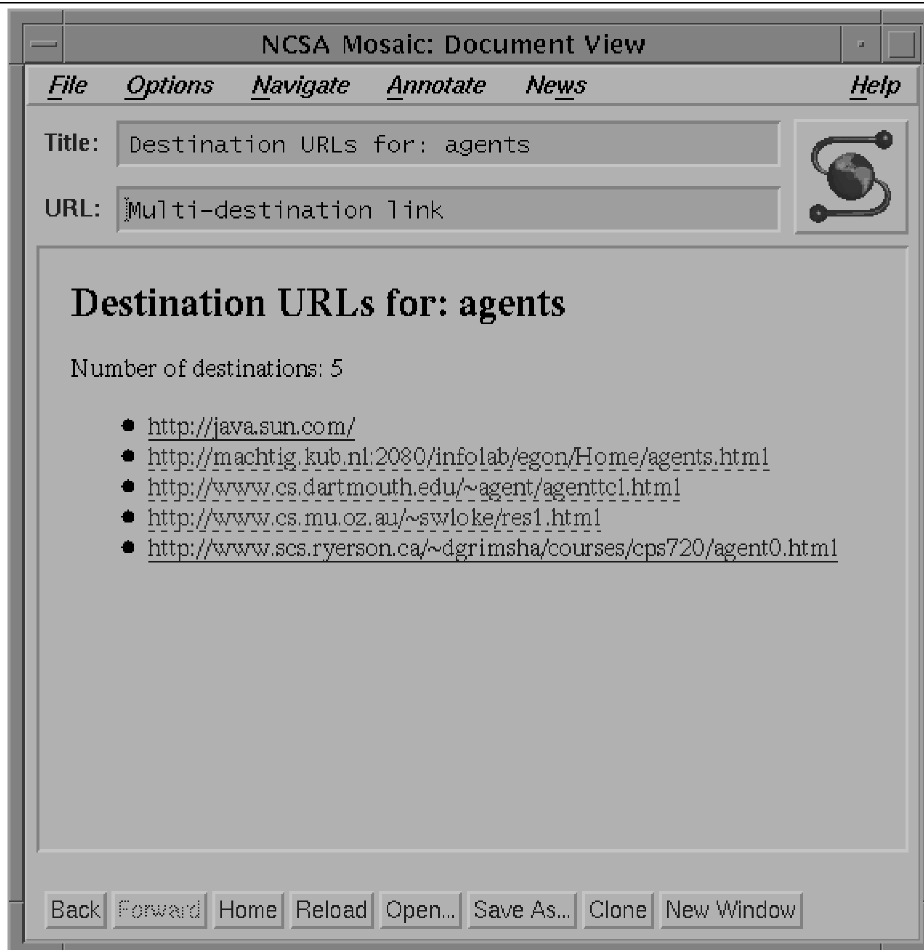
By changing the meaning of `link_action/1`, the same link can have quite different semantics. For instance, all pages related to the given concept can be returned. `link_action/1` uses `setof/3` applied to `trans_about/2` to obtain all suitable URLs. If only a single URL is found then `show_page/1` is called, otherwise the list of URLs is assembled into a HTML page by `report_urls/3` and printed.

```
link_action(HREFString) :-
  link_parts(HREFString, ISA-URL, Concept),
  setof(DestURL, lw(get, ISA-URL)#>
                      trans_about(Concept, DestURL), DestURLs),
```

```
([DestURL1] = DestURLs ->
   show_page(DestURL1)
;
   report_urls(DestURLs, Concept, Links),
   display_page("Multi-destination link", [data(Links)])
).
```

Figure 7.3 shows the page constructed when the link to “agents” is followed.

Figure 7.3 The page constructed when the link to “agents” is selected.



7.2.2 Page Name Links

We now consider an alternative link abstraction, which maps names to URLs. This technique is similar to Uniform Resource Names (URNs) introduced in Chapter 2.

Currently, when a Web page is moved, the links to it from other pages will cease to work. One solution is to define links in terms of page names rather than addresses, and use a central database (or databases) to map these names to their actual URLs. When an author moves a page he must update its URL in the relevant database, but the page name remains the same. This means that users of a page (who use the page name as a link reference) are unaffected by the movement of the page.

We imagine a database of page names and URLs of the form:

```
page_details(Name, URL)
```

e.g.

```
page_details("Seng's_Page", "http://www.cs.mu.oz.au/~swloke").
page_details("Andrew's_Page","http://fivedots.coe.psu.ac.th/~ad").
```

This database will be stored on a Web page at:

```
http://www.db.com/details.html
```

The URL syntax is extended to include a page name reference:

```
http://www.db.com/details.html-nm-<page-name>
```

e.g.

```
http://www.db.com/details.html-nm-Seng's_Page
```

When such a link is selected on a page, its meaning must be defined with `link_action/1`:

```
link_action(HREFString) :-
  append(DbURL, [0'-,0'n,0'm,0'-|PgName], HREFString),
  lw(get, DbURL)#>page_details(PgName, PgURL),
  show_page(PgURL).
```

The same technique can be used to simplify page referencing on a server host. We assume that each server host has a database called `map.html` of `page_map/2` facts. The database maps page names to actual pages on the server host. For example:

```
page_map("LogicWeb", "http://www.cs.mu.oz.au/~swloke/logicweb.html").
```

Users can employ a particular server's `map.html` database by writing URLs of the form:

```
http://<server-address>/map.html-nm-<page-name>
```

e.g.

```
http://www.cs.mu.oz.au/map.html-nm-LogicWeb
```

This will return Seng's `logicweb.html` page.

An advantage of a server host offering page names as opposed to globally unique page names is not only that the name space required is smaller, since the page names need only be unique within the server host, but also the increased maintainability: registration of page names is handled on a per server basis rather than by a global authority. Offering page names does not rule out pages from being explicitly referred to via their URLs, but the incentive for linking with page names is robustness.

7.2.3 Linking Based on Logical Relationships Between Pages

Relationships between pages are readily defined using rules. This subsection considers structural and temporal relationships, and shows how these relationships can be used to define useful Web links.

7.2.3.1 Structural Relationships

As we have seen in the previous chapter, a guided tour represents a sequence relationship between the pages in the tour. The tour links inserted into the tour

nodes (pages) allow navigation based on this relationship (e.g., to the next or previous node in the sequence). Other structural relationships between pages can be similarly defined, and used for providing meaningful paths through the page corpus. The paths guide the reader as he/she navigates through the pages providing a fast and efficient way to find related pages, so improving access to information.

Empirical evidence for the value of meaningful paths through pages is exhibited by Webrings[1], which offer a way of navigating the Web by following paths in the form of rings. Each page in a Webring contains links to two other pages in the Webring (similar to the back and forward links in a guided tour). The URLs used in the links are manually inserted. There are currently 15 000 Webrings, each on a different topic and containing dozens to hundreds of pages. LogicWeb can be used to create and manage more complex structural relationships between pages.

The organisation of a large collection of pages can be represented as a logic program, enabling the organiser of these pages to get a feel for the overall structure of the collection. For example, the following rules organise a collection of URLs into a hierarchy:

```
parent("URL1", ["URL2", "URL3", "URL4"]).
parent("URL2", ["URL5", "URL6"]).
parent("URL3", []).            % a leaf node
   :  % more parent/2 facts

sibling(URL, SiblingURL) :-
  parent(_, ChildURLs),
  member(URL, ChildURLs),
  member(SiblingURL, ChildURLs),
  SiblingURL \= URL.

child(ChildURL, URL) :-
  parent(URL, ChildURLs),
  member(ChildURL, ChildURLs).
```

[1] http://www.webring.org/

```
root(URL) :-
  not has_parent(URL).

has_parent(URL) :-
  child(URL, _).
```

`parent/2` defines a parent to children relation among a set of URLs. `sibling/2` states that the sibling of a URL is a URL which has the same parent. `child/2` defines the inverse of the parent relation. The root URL is the URL which has no parent. Useful navigational links whose destinations are inferred dynamically from these relationships can be added to pages.

Suppose the above rules are stored in the page:

```
http://www.cs.mu.oz.au/~swloke/hierarchy.html
```

A page can link to its parent by specifying an anchor with a HREF string of the form:

```
http://www.cs.mu.oz.au/~swloke/hierarchy.html-fam-parent
```

A link of this type will query the logic program representing the hierarchy of pages for the parent URL. This behaviour is implemented by defining `link_action/1` in the page as follows:

```
link_action(HREFString) :-
  append(HURL, "-fam-parent", HREFString),
  my_id(_, MyURL),
  (lw(get, HURL)#>child(MyURL, ParentURL) ->
    show_page(ParentURL)
  ;
    true
  ).
```

If there is no parent, then `child/2` will fail. Otherwise, the parent page will be displayed.

Links to the page's child, sibling, or the root are made in a similar way (e.g., by replacing `parent` with `child` in the anchor URL and the `link_action/1` rule).

To enable a page to link to its parent, siblings, and children, three anchors are added to the page, together with the corresponding `link_action/1` rules. These links form paths through the pages whose structure is defined by the URL hierarchy.

Typically, Web sites are dynamic in that new pages are often created and removed. An advantage of dynamically inferring the link destination is that the pages containing the child, parent and sibling links need not be changed when the hierarchy is updated. For instance, if a new `parent/2` fact is added, since the child, sibling, and root relationships, and the link destinations are computed, they need not be changed. It is also easy to change the structure of the relationships. For example, the hierarchy could be changed to a binary tree without affecting the pages.

To maintain the hierarchy, it can be queried for particular properties. For instance, to ensure that there is only one root, the program can be queried with:

```
?- setof(URL, root(URL), [URL]).
```

Hypertext links can be classified into types, each type representing a particular kind of relationship. HTML 4.0 [171] defines the REV and REL attribute in HTML anchors whose value is a keyword specifying a link type. HTML link types have yet to be standardised, but a range of keywords for describing HTML link types some of which are similar to the relationships we have seen is given in [137]. Examples of these keywords include "child", "parent", and "sibling" describing hierarchical relationships, "begin", "next" , and "previous" describing sequence relationships, and "citation", "definition", "footnote", and "author" describing related documents.

There has been much work in the hypertext community on link typing. For example, in [28], a set of domain-independent relationships between hypertext

nodes is proposed, and in [73], a taxonomy of link types according to their function, structure, and preferred means of implementation is given.

Link types may depend on the subject matter of the Web pages, instead of being domain-independent. For instance, in the Open Meeting system [103] for supporting Web-based on-line discussions, a set of link types is proposed for structuring its pages such as "agree" which describes a link to a document supporting an action, and "answer" which describes a link to a document answering a question.

This thesis does not prescribe a set of link types, but argues that the flexibility of the LogicWeb linking mechanism permits different kinds of link types to be implemented on the Web.

7.2.3.2 Temporal Relationships

To illustrate temporal relationships between pages, page versioning is considered. Version management is an important concern for the Web due to the dynamic nature of its content. When a page is updated, its previous versions may just be as valuable as the latest version.

Since each page is identified by its URL, different versions of a page (which are themselves pages) would have their own URLs. A mechanism is needed to allow a document to be referred to independently of its versions. For instance, it should be possible to refer (or link) to someone's homepage without requiring knowledge about how many versions there are and what the latest version is, but it should also be possible to link to a specific version of the homepage. A means to achieve this is to link to a database of page versions.

As an example, consider a page containing the following facts storing the URLs of four different versions of a homepage:

```
version(1, "http://www.cs.mu.oz.au/~swloke/a.html").
version(2, "http://www.cs.mu.oz.au/~swloke/b.html").
version(3, "http://www.cs.mu.oz.au/~swloke/c.html").
version(4, "http://www.cs.mu.oz.au/~swloke/d.html").  % latest
```

Then, the URL of this page (say, `http://www.cs.mu.oz.au/~swloke`) becomes the version-independent identity of the homepage.

A page can link to the second version of the homepage by using the URL:

```
http://www.cs.mu.oz.au/~swloke-v-2
```

and by defining `link_action/1` as follows:

```
link_action(HREFString) :-
  append(VersionDBURL, [0'-,0'v,0'-|Version], HREFString),
  name(VersionNumber, Version), % convert string to atom
  lw(get, VersionDBURL)#>version(VersionNumber, URL),
  show_page(URL).
```

Rules capturing relationships between page versions can be added to the version database:

```
latest(LURL) :-
  setof(VersionNumber-URL, version(VersionNumber, URL), VURLs),
  keysort(VURLs, SortedVURLs),
  last(LURL, SortedVURLs).

earliest(EURL) :-
  setof(VersionNumber-URL, version(VersionNumber, URL), VURLs),
  keysort(VURLs, [EURL|_]).

precedes(URL1, URL2) :-
  version(VersionNumber1, URL1),
  version(VersionNumber2, URL2),
  VersionNumber1 < VersionNumber2.

supercedes(URL1, URL2) :-
  version(VersionNumber1, URL1),
  version(VersionNumber2, URL2),
  VersionNumber1 > VersionNumber2.
```

`latest/1` retrieves the latest version, and `earliest/1` the earliest version. `precedes/2` and `supercedes/2` compare the version number of two URLs.

With versioning, links to pages can be given a new interpretation: an anchor's HREF string which does not specify a version number, but points to a version

database, refers to the latest version of a page. For instance, the following anchor implicitly refers to the latest version of Seng's homepage:

```
<A HREF="http://www.cs.mu.oz.au/~swloke">Seng's homepage</A>
```

This is implemented by defining `link_action/1` as follows:

```
link_action(URL) :-
  lw(get, URL)#>latest(LURL).
  show_page(LURL).
```

Similarly, the `earliest/1`, `supercedes/2`, and `precedes/2` relationships can be used to navigate over page versions by suitably defining the format of anchor's HREF strings (e.g., suffixing the version database's URL by "`-v-earliest`") and the corresponding `link_action/1` rules.

Spatial and temporal relationships can be combined. For instance, the following rule states that a relationship `R` (e.g., `child`) holds between two pages `URL1` and `URL2` if the same relationship holds between their earlier versions.

```
holds(R, URL1, URL2) :-
  precedes(VURL1, URL1),
  precedes(VURL2, URL2),
  Rel =.. [R, VURL1, VURL2],
  Rel.
```

This rule enables structural relationships to hold for newly added versions of pages.

7.2.4 Links Based on Page Information

The preceding examples have used information external to Web pages (i.e., an IS-A hierarchy, a page name database, relationships between URLs). The meaning of a link can also be determined from the source and/or destination pages, by executing their LogicWeb code, by parsing their text, or by looking at their meta-information.

For instance, the following link filtering rule ensures that only up-to-date, interesting, and non-bookmarked pages are displayed:

```
link_action(URL) :-
  up_to_date(URL),
  interesting(URL),
  not(bookmarked(URL)),
  show_page(URL).


up_to_date(URL) :-
  lw(head, URL)#>about(last_modified, LastModifiedTime),
  getime(LastModifiedTime, '1998;04;31;01:00').

interesting(URL) :-
  lw(get, URL)#>h_text(Src),
  lw(get, "http://www.cs.mu.oz.au/~swloke")#>interested_in(KeyPhrase),
  contains(Src, KeyPhrase).

bookmarked(URL) :-
  lw(get, "http://www.cs.mu.oz.au/~swloke/book_marks.html")#>
                             link(_, URL).
```

`up_to_date/1` examines the page's meta-information. `getime/2` checks if the last modified time is equal to or after 1 am on the 31st of April 1998. `interesting/1` assumes that Seng's homepage contains an `interested_in/1` predicate (as defined in Section 3.2), and checks if any of its key phrases are in the text of the page. `bookmarked/1` assumes the existence of a `book_marks.html` page which contains links to Seng's bookmarked pages. These are tested against the page being considered.

7.2.5 Dynamically Constructing Pages

A destination page can be dynamically constructed by composing pre-stored elements together. For instance, activating a link to a page may retrieve the page augmented with extra details, such as footnotes, background links, or advertising material.

As an example, we assume that the page

`http://annotations.url.db/`

contains facts relating the URLs of pages to notes that must be added to the bottom of those pages. The facts are in the form:

```
annotation_page(URL, NotesURL).
```

`NotesURL` is the URL of the page containing the notes about the `URL` page. For a given page, the following rule retrieves its notes and displays the page text followed by those notes:

```
link_action(URL) :-
  lw(get, "http://annotations.url.db/")#>
                              annotation_page(URL, NotesURL),
  lw(get, URL)#>h_text(Src),
  lw(get, NotesURL)#>h_text(Notes),
  display_page("Annotated Page", [data(Src), data(Notes)]).
```

Customising Web pages during link traversal is similar to an application of *perspectives* in [168, 169]. Perspectives are graph structures that can be combined and operated on in various ways. Applied to the Web, perspectives can represent combinations of pages or page fragments. As noted in [169], this idea is implemented by extending a URL with a perspective expression:

```
http://<host>:<port>/<path>:<perspective expression>
```

A perspective expression specifies a combination of several perspectives. A link with a URL of this form triggers a computation on the (perspective-enabled) server host to construct a page using the perspective expression. In contrast, computations with the LogicWeb system are client-based, but LogicWeb goals can retrieve information from perspective-enabled servers by using URLs with perspective expressions.

7.3 Handling Nondeterminism in Web Links

Due to the dynamic, unpredictable nature of the Web, link traversal is nondeterministic. Depending on the current state of the Web, selecting a link can result in

a number of possible outcomes including a page being displayed, a redirection message, or an error message (e.g., the server is busy, or the page no longer exists). Nondeterminism in link traversal is easily modelled with logic programs. This section presents examples of LogicWeb rules for handling redirection pages and download failures.

7.3.1 Redirection Pages

A Web page that has been moved is often replaced by a redirection page, which typically contains a message like "This site has moved. We are now at...(the new URL)". It is possible to automate the task of following a redirection link so that the user need never see the redirection page.

`link_action/1` examines the target page. If it is a redirection page, then `link_action/1` is invoked (recursively) with the new URL. Otherwise, the page is displayed:

```
link_action(URL) :-
  (redirection_page(URL) ->
    lw(get, URL)#>link(_, NewURL),
    link_action(NewURL)
  ;
    show_page(URL)
  ).

redirection_page(URL) :-
  lw(get, URL)#>title(Title),
  ( contains(Title, "has moved")
  ; contains(Title, "site moved")
  ; contains(Title, "redirection to new location")
  ; contains(Title, "redirect URL")
  ).
```

A redirection page is recognised by looking for specific phrases in the title. The first link on a redirection page is assumed to contain the new URL.

Alternatively, instead of relying on the page's HTML text, a `redirect/1` fact

containing the new URL can be added to a page to indicate a redirection. Assuming that the user wants to see the contents of redirection pages, the following `link_action/1` rules specify how to follow a chain of redirections till the actual page is reached, whereupon the actual page and the text of all the redirection pages are displayed.

```
link_action(URL) :-
  lw(get, URL)#>redirect(RedirectURL), !,
  ((#) + lw(get, URL))#>link_action(RedirectURL).
link_action(URL) :-
  setof(Src, h_text(Src), Srcs),
  concat_string(Srcs, Str),
  display_page(URL, [data(Str)]).
```

The first rule adds the new program to the current context when a `redirect/1` fact is found on it, and recursively invokes `link_action/1` in the extended context. If no such fact is found, the second rule is invoked which retrieves the data from the `h_text/1` facts in the current context, and displays them. The `h_text/1` facts are accumulated by the first rule as programs are added.

An assumption here is that `link_action/1` is defined only in the page containing the above `link_action/1` predicate, and not in the other pages visited.

7.3.2 Broken Links

A broken link is a link whose destination page is no longer available (e.g., the page has been moved or deleted).

Rules can be written to automatically perform tasks which the user might manually do when a broken link is encountered. For instance, if a page referred to by a URL is not retrieved, its parent directory's URL could be tried. This procedure is repeated until a page is retrieved, or the server's URL has been tried. The following rule implements this behaviour:

```
link_action(URL) :-
  (lw(get, URL)#>h_text(HTMLSrc) ->
    display_page(URL, [data(HTMLSrc)])
  ;
    show_parent(URL)
  ).

show_parent(URL) :-
  parent_directory(URL, ParentDirURL),
  link_action(ParentDirURL).
```

`parent_directory/2` returns the URL of the parent directory but fails if its first argument has no parent. For example, given
`http://www.cs.mu.oz.au/~swloke/`
as the first argument, `parent_directory/2` returns
`http://www.cs.mu.oz.au/`.

Alternatively, a server could provide a service enabling clients to report broken links. Each page offered by such a server could contain a `link_action/1` rule which informs the server whenever a page is not retrieved. When a report of a broken link is posted to the server, the server may e-mail the page's owner to fix the link, and/or return information to the client to determine the subsequent course of action, which may be to show the broken link's parent, show a mirror site given by the server, or retry the HTTP request after a period specified by the server.

```
link_action(URL) :-
  (lw(get, URL)#>h_text(HTMLSrc) ->
    display_page(URL, [data(HTMLSrc)])
  ;
    my_id(_, MyURL),
    lw(post([field("On Page", MyURL), field("Broken Link", URL)]),
       "http://www.cs.mu.oz.au/cgi-bin/report_failure")#>
            client_action(Information),
    react(Information, URL)
  ).
```

```
react(page_being_updated, URL) :-
  show_parent(URL).        % show its parent
react(try_mirror(MirrorURL), _) :-
  link_action(MirrorURL).  % try a mirror page
react(retry_later(N), URL) :-
  sleep(N),                % delay N seconds before retrying
  link_action(URL).        % retry the link action
```

In the `link_action/1` rule, a POST message to report a broken link is generated automatically. The POST message consists of the page containing the broken link, the broken URL, and the CGI script of the report handling program on the server host. `react/2` determines what to do based on the information returned by the server host.

HTTP returns error messages to the client when a request fails, but the above mechanism is programmable at the application-level, and hence, is more general. For example, an application-specific protocol can be implemented.

7.4 History-based Linking

State information can be kept in the abstraction layer for use in future link traversals, and updated using Prolog's dynamic database update facility (i.e., via `assert/1` and `retract/1`). A LogicWeb program can use this facility to update itself in response to link selections, allowing link behaviours to reuse results of previously executed behaviours (similar to memoing where previously calculated answers are stored and later used). As an example, we consider how to order the links on a page so that a link can be traversed only if some other links have been followed. One use of such an ordering of links is to present course notes with dependency relationships among them (e.g., one is a prerequisite of another).

The predicate `depends_on/2` is used to describe the dependency between the links, and a `selected(HREFString)` fact is asserted when the link identi-

fied by HREFString is selected for the first time.

```
link_action(HREFString) :-
  selected(HREFString), !,     % selected previously
  show_page(HREFString).
link_action(HREFString) :-
  setof(HREFString1, depends_on(HREFString, HREFString1), Hs),
  all_selected(Hs), !,         % check dependencies
  assert(selected(HREFString)),% record the link selection
  show_page(HREFString).
link_action(_).

% check that all links have been selected
all_selected([]).
all_selected([H|Hs]) :-
  selected(H),
  all_selected(Hs).

depends_on("http://www.cs.notes/operating_system.html",
           "http://www.cs.notes/data_structures_algorithms.html").
     :  % more depends_on/2 facts
```

The above link_action/1 rules enforce the ordering on the links. If the required links as specified by depends_on/2 have not been followed, then the third link_action/1 rule is chosen so that the destination page is not displayed. To represent indirect dependencies, a transitive closure on depends_on/2 can be defined (similar to trans_isa/2 shown earlier). The link_action/1 can also be modified to report URLs which are depended upon but not yet selected.

7.5 Using Multiple Link Behaviours

A range of useful link behaviours have been presented. These can serve as the basis for constructing more complex link behaviours.

Within a page, multiple link behaviours can be utilised *independently* or *cooperatively*, or in both modes. In the independent mode, each kind of link is distin-

guished by a specific format for the anchor's HREF string. The page would contain a collection of `link_action/1` rules (e.g., one for each kind of link). The rule for processing a link selection is chosen by pattern matching on the HREF string format.

In the cooperative mode, the link behaviours interact more closely. There are many different ways of combining simpler link behaviours into more complex ones. For instance, a link behaviour which uses page names links and also performs automatic redirection can be coded up from scratch. However, a more modular approach would allow simple link behaviours to be specified separately and combined in a principled way. Two methods for combining link behaviours which encourages modularity are illustrated below.

7.5.1 LogicWeb Operators

A link can utilise the `link_action/1` rules of other pages using LogicWeb operators. For instance, the rule below defines links whose semantics are exactly those of the links found in another page:

```
link_action(HREFString) :-
  lw(get, "URL0")#>link_action(HREFString).
```

This mechanism can be used to provide consistent link behaviours. All the pages on a host can utilise the same link behaviour (e.g., to report broken links as shown in Section 7.3.2) by invoking the `link_action/1` rule in the same program using context switching.

It is also possible to build customisable link behaviours by leaving particular predicates open (or undefined). For example, suppose `lw(get, "URL0")` contains the `link_action/1` rule described in Section 7.2.4, but not the definitions for the predicates in the rule's body:

```
link_action(URL) :-
  up_to_date(URL),
  interesting(URL),
  not(bookmarked(URL)),
  show_page(URL).
```

The following rule utilises the `link_action/1` predicate in `lw(get, "URL0")` but retains predicates in the current context which have not been defined in `lw(get, "URL0")`:

```
link_action(HREFString) :-
  (lw(get, "URL0") + ((#) / lw(get, "URL0")))#>
                              link_action(HREFString).
```

If the current context defines the predicates `up_to_date/1`, `interesting/1`, and `bookmarked/1`, then these predicates will be used in evaluating the LogicWeb goal. The result is that the link behaviour depends on the current context.

Constraints can be imposed on link behaviours using LW-intersection and LW-restriction similar to how database views are constructed.

7.5.2 Link Transducers

The idea of *link transducers* captures a recurrent operation in link behaviours which is to convert one kind of link to another. More specifically, each link transducer takes a HREF string (e.g., a URL) of a specific format and converts it into another string of the same or different format. For instance, the link behaviour from Section 7.2.2 converts

```
http://www.db.com/details.html-nm-Seng's_Page
```

to

```
http://www.cs.mu.oz.au/~swloke
```

Also, the link behaviour from Section 7.2.3.2 converts

```
http://www.cs.mu.oz.au/~swloke-v-2
```

to

```
http://www.cs.mu.oz.au/~swloke/b.html
```

The link behaviours in Sections 7.2.3.1 and 7.3.2 perform a similar conversion. Such link behaviours can be implemented in distinct LogicWeb programs with a well-defined interface and utilised in a standard way. The interface could take the form:

```
<conversion_operation>(<input HREF string>, <output string>)
```

Using this interface and assuming that a set of link behaviours are defined on the server `http://www.link.org/`, the following `link_action/1` rule converts a concept link (i.e., the URL of the knowledge-base appended with a concept name) into a page name link (i.e., the URL of the page name database appended with the page name), the page name link into a version link (i.e., the URL of the version database and a version number), the version link into a URL, and finds and displays a mirror site of the URL performing a redirection if necessary:

```
link_action(HREFString) :-
  lw(get, "http://www.link.org/ca.html")#>
                       concept_to_actual(HREFString, PageNameLink),
  lw(get, "http://www.link.org/pa.html")#>
                       pagename_to_actual(PageNameLink, VersionLink),
  lw(get, "http://www.link.org/va.html")#>
                       version_to_actual(VersionLink, URL),
  lw(get, "http://www.link.org/m.html")#>
                       mirror(URL, MirrorURL),
  lw(get, "http://www.link.org/r.html")#>
                       redirect(MirrorURL, RedirectedURL),
  show_page(RedirectedURL).
```

The following bindings illustrate a sequence of conversions:

```
HREFString    = http://cat.cs.mu.oz.au/~swloke/link_kb.html-isa-agents
→
PageNameLink  = http://cat.db.com/details.html-nm-MUAgentLab
→
VersionLink   = http://cat.cs.mu.oz.au/agentlab.html-v-3
```

```
→
URL           = http://cat.cs.mu.oz.au/agentlab3.html
→
MirrorURL     = http://munta.cs.mu.oz.au/agentlab3.html
→
RedirectedURL = http://munta.cs.mu.oz.au/agentlab3/agentlab3.html
```

The above definition of `link_action/1` performs a rigid sequence of conversions from the concept link to the redirected URL. A more flexible way is to recursively pattern match on the form of the URL (e.g., `-isa-`, `-nm-`, and `-v-`) until no conversion is required:

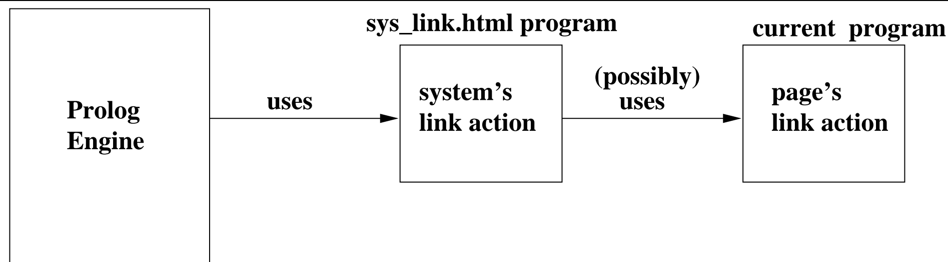
```
link_action(HREFString) :-
  ( (contains(HREFString, "-isa-"),
     lw(get, "http://www.link.org/ca.html")#>
                    concept_to_actual(HREFString, HREFString1))
  ; (contains(HREFString, "-nm-"),
     lw(get, "http://www.link.org/pa.html")#>
                    pagename_to_actual(HREFString, HREFString1))
  ; (contains(HREFString, "-v-"),
     lw(get, "http://www.link.org/va.html")#>
                    version_to_actual(HREFString, HREFString1))
  ), !,
  link_action(HREFString1).
link_action(HREFString) :-
  show_page(HREFString).
```

The above `link_action/1` rules permit various sequences of conversions (e.g., a page name link → another page name link → concept link → version link → page name link → ordinary URL).

7.6 System Link Actions

So far, the above link behaviours are specified on downloaded pages. The LogicWeb system's user may want to use particular link behaviours (e.g., automatic redirection) for all links regardless of the page containing the links. An extension

Figure 7.4 Components for handling a link selection.



of the LogicWeb system is given here to allow the user to build link behaviours into the system. These behaviours are defined in a LogicWeb program with a default identifier.

Figure 7.4 shows the components for handling a link selection. A predicate `sys_link_action/2` in the LogicWeb program `sys_link.html` is defined by the user and invoked whenever a link is selected. The first clause of the predicate `handle_link/1` in the Prolog engine described in Section 4.4 is modified to use `sys_link.html`:

```
handle_link(SelectedURL) :-
  current_page(CurrentPageURL),
  demo(empty, lw(get, "http://www.cs.mu.oz.au/lw_sys/sys_link.html")#>
                      sys_link_action(CurrentPageURL, SelectedURL)).
```

`sys_link_action/2` can directly invoke `link_action/1` in the current page:

```
sys_link_action(CurrentPageURL, SelectedURL) :-
  lw(get, CurrentPageURL)#>link_action(SelectedURL).
```

Alternatively, `sys_link_action/2` can perform a computation similar to the link behaviours described earlier, possibly downloading other LogicWeb programs in the process. For example, page filtering, or the handling of broken links, IS-A hierarchy links, and page name links can be built into the system in `sys_link.html`.

sys_link_action/2 need not use the link_action/1 rule in the page containing the link, i.e. the link behaviours prescribed by page authors can be ignored. Prioritisation schemes can also be employed, where conflicts between the two rule sets (the one on the page and the system's) are resolved. For instance, the following definition adds to the current context only the predicates on the current page which have not been defined in the current context:

```
sys_link_action(CurrentPageURL, SelectedURL) :-
  ( (#) + (lw(get, CurrentPageURL) / (#)) )#>
                                link_action(SelectedURL).
```

7.7 Related Work

7.7.1 Hypertext Tools

Hypertext systems which utilise semantic networks to structure their information and which allow links to invoke computations are surveyed in [139]. An important difference between these systems and LogicWeb is that they deal with a closed information base whereas the Web is open. With closed information, it is possible to build semantic networks to represent the entire corpus.

In the HyperBase hypertext tool [123, 4], Prolog code can be invoked when buttons are pressed. Applications include a system to help users fill in forms, and a system for teaching neuropathology. LogicWeb is similar in that Prolog-like code is also invoked, but this is in the context of the Web, and the manipulation of pages as logic programs.

7.7.2 Web-based Tools

7.7.2.1 Structured Maps

The knowledge-based structures describe earlier are built above existing Web information, in a similar way to Structured Maps [136]. Structured Maps repre-

sent semantic information using an entity-relationship diagram (ERD) expressed in SGML, and have links to the underlying information. A three-level model is utilised: the top-most level is the entity-relationship schema, the middle level is an entity-relationship instance, and the bottom level is the information base. The reader browses the Maps, moving between levels to locate information. A query language for Structured Maps has not yet been developed.

For the knowledge-based structures in this chapter, the query language is the knowledge representation language, and the data modelling power of rules not found with Structured Maps is present.

7.7.2.2 CGI

A common technique for extending the meaning of a Web link is to use CGI scripts. The key difference with this work is that CGI scripts execute on the server, whereas the LogicWeb system is client-based. This reduces the server load, and localises computation. Also code can be combined together in one place, and state information can be maintained easily.

7.7.2.3 JavaScript

JavaScript is designed to allow the elements of a Web page (e.g., form input fields, links, text attributes) to be manipulated via a programming language [155]. For example, it is possible for the selection of a link to invoke a JavaScript function. Two restrictions with JavaScript are that all downloaded pages must be displayed, and only the forms and anchors of downloaded pages can be accessed within a program.

The LogicWeb system permits pages to be processed in more general ways. Also, its emphasis on the rule-based specification of link behaviour offers more direct ways for representing and manipulating knowledge-based structures. For example, using an imperative language to implement the link behaviours will involve manipulating an assortment of data structures and result in code with

data which is not explicitly stored (e.g., as a set of facts), and so, less readable and more difficult to update.

7.7.2.4 Link Management Systems

A number of link management systems have been proposed which aim to prevent or fix broken links. For example, the ATLAS system [165] consists of a network of servers (called ATLAS servers) each of which is tightly coupled to a Web server and contains databases about link information. Changes in links on Web pages are communicated between ATLAS servers. The CLT/WW system [60] periodically traverses the links on a site to repair broken links and notifies page authors via e-mail of changes. LogicWeb offers robust linking through the abstraction layer (e.g., page names links) and enables algorithms to be triggered to handle broken links whenever they are detected during browsing.

7.8 Discussion

A two-level Web model based on LogicWeb has been presented together with examples of its use. The examples show that this model buys significant expressiveness for defining Web links, and demonstrate the following advantages of logic programming for implementing link behaviours:

- Structured data crucial to many of the link behaviours, such as conceptual structures and databases, can be coded up easily and declaratively, and the processing of the data can be programmed in the language the data is represented.

- It is advantageous that such data is easy to revise being stored as distinct units of facts and rules since the data (e.g., versions of pages) is typically dynamic.

- Pattern matching via unification enables the right `link_action/1` rule to be selected automatically.

- Dynamic database manipulation allows history-based linking.

Finally, the use of LogicWeb programs provides a modular approach to reusing and constructing link behaviours.

The examples also show that linking to meta-data about pages stored as LogicWeb programs (e.g., version information, page names, and classification of pages by topics) addresses four areas of weaknesses in the original Web link mechanism:

1. *inability to link to generic resources*: The idea of a *generic resource* was introduced by Tim-Berners Lee in [22]:

   > "...a resource may be generic in that as a concept it is well specified but not so specifically specified that it can only be represented by a single bit stream."

   IS-A hierarchy links provide an example of linking to generic resources (e.g., a concept) rather than to a specific file.

2. *links are easily breakable*: page name links provide more robust linking, thereby reducing the broken links problem caused by the movement or deletion of link destination pages. Moreover, as shown in Section 7.3.2, algorithms can be coded to automatically handle broken links.

3. *lack of clarity in the semantics of links to support navigation on the Web*: meaningful link relationships can be defined which allow multiple organisations of a given collection of pages in a maintainable manner. These relationships provide meaningful paths through the Web.

4. *no mechanism for integrating versioning with linking*: versioning can be amalgamated with Web link semantics by linking to databases containing version information.

This chapter has shown the benefits of generalising the source of Web links from pages to LogicWeb programs. Linking from a LogicWeb program allows state information to be kept as part of the program and utilised for linking (as shown in Section 7.4), and link behaviours to be specified by the link's source.

An important issue in linking is the response time. It is unreasonable to expect the user to wait long for the computation results after selecting a link. Link behaviours normally spend the majority of their wall-clock execution time interacting with the Web. A link behaviour which performs a single page retrieval is not significantly slower than an ordinary link traversal (i.e., one where `link_action/1` is not invoked). But a link behaviour which involves multiple page retrievals could take more time. Time and other resource bounds can be imposed on goal evaluations, and are discussed in the next chapter.

Chapter 8

Security in the LogicWeb System

Security is an important and non-trivial issue in using mobile code (i.e., in *mobile computing*). To quote Meseguer and Talcott [147]:

> "Security is probably the biggest technical challenge that needs to be solved to achieve widespread acceptance of mobile computing."

This chapter is devoted to the security issues in the LogicWeb system.

8.1 What are the Security Issues in the LogicWeb System?

In the LogicWeb system, programs are downloaded from remote sites and executed locally. A malicious or buggy program whose execution is not controlled can cause problems on the local host. As noted in [43, 163], the local host could encounter three types of attacks from such programs:

1. *integrity attacks*: attempts to delete or modify information in the local environment in unauthorised ways.

2. *privacy attacks*: attempts to steal or leak information to unauthorised parties.

3. *denial of service attacks*: attempts to occupy resources to the extent which interferes with the normal operation of the local host.

The above attacks are possible if the foreign program has unregulated access to local system resources such as the file system, network services (e.g., socket connections), CPU cycles, internal memory, input/output devices, program environment (e.g., environment variables), and operating system commands. An example of an integrity attack is a LogicWeb program issuing the operating system command `rm` (via the SWI-Prolog built-in predicate `system/1`) to delete files from the local file system. An example of a privacy attack is a LogicWeb program reading a file (using the SWI-Prolog built-in predicate `open/3`) and transmitting the contents to another host using a POST HTTP request. An example of a denial-of-service attack is a LogicWeb program going into an infinite loop resulting in CPU cycles being wasted, heap space being depleted when a huge number of LogicWeb programs is downloaded, or disk space being exhausted when a file is incessantly being written into.

On the other extreme, a foreign program which is denied access to all resources cannot do any damage. But such a program cannot also perform useful work. The problem addressed in this chapter is how to provide safe (i.e., host-protecting) execution of LogicWeb programs without overly restricting their capabilities.

The aim is to develop a model for the safe execution of LogicWeb programs which is:

- *flexible*: the security model should support varying degrees of trust and access to resources. More trusted programs should be given more privileges to achieve greater functionality. The model must determine who has access to what resources in what way.

- *formally specified*: a formally specified security model provides a precise meaning of what *safe* means. An additional advantage is the ability to

translate the specification into an implementation.

- *easy to implement and trust*: the design of the security model should not be unduly complex so that it is easy to implement and the implementation can be trusted to work correctly.

The subsequent sections present a security model with the above three properties, and describe how the LogicWeb system presented in Chapter 4 has been extended using this model.

8.2 Overview of Security Model

The key characteristics of the LogicWeb implementation relevant to the design of a security model are:

- The program source is transmitted and downloaded. Hence, the local host has access to the source rather than a compiled form (e.g., binary executable or byte-code). This means that each goal (e.g., built-in predicates) invoked is visible at the interpreter level. The source, if unencrypted, is open to tampering, and can be intercepted during transmission and modified.

- Downloaded programs are executed by an interpreter based on the operational semantics in Chapter 3. This semantics is extended to specify a safe execution model for LogicWeb programs.

- A LogicWeb program can invoke goals in other programs. The model must deal with such communication between programs of different levels of trust. A less trusted program should not be able to use the privileges of a more trusted program.

A *sandbox* model is adopted where each program is executed in its own sandbox limiting its access to resources and controlling the way the resources are used.

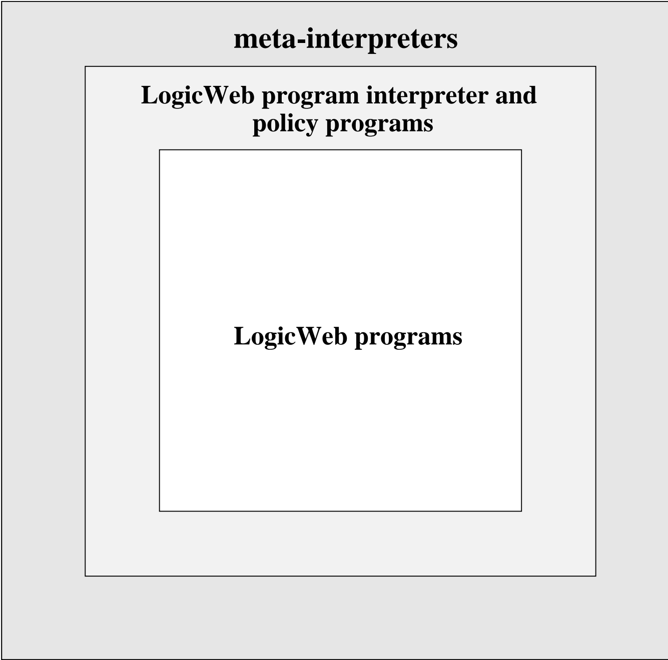
A sandbox for a LogicWeb program consists of its interpreter and a *security policy*, or *policy*, for short. The policy controls the program's access to system resources. Security policies allow flexibility in handing out privileges to LogicWeb programs of varying trust. A more trusted LogicWeb program is assigned a policy which allows less restricted access to resources, and conversely, a less trusted program is assigned a more restrictive policy. Security policies would be defined by the LogicWeb system's user.

A LogicWeb program accesses the local environment and system resources via *system calls*, which are SWI-Prolog built-in predicates and the built-in predicates for LogicWeb applications described in Appendix B. A policy specifying allowed system calls and the way they are to be used is encoded as a set of predicates in a LogicWeb program called a *policy program*. The policy program also specifies the programs that a LogicWeb program can utilise via context switching. The LogicWeb program interpreter invokes system calls by calling predicates in policy programs contained in one of the interpreter's arguments (see Section 8.7). Policy programs are stored in remote sites or locally (as long as they are protected from tampering) and integrated into the system as needed while the system is running.

The policy program assigned to a LogicWeb program is determined by how much the program is trusted, which in turn, depends on the program's origin (e.g., its owner or creator). The program's origin is authenticated using PGP (Pretty Good Privacy) digital signatures [90].

Policy programs restrict the use of system calls, thereby preventing integrity and privacy attacks. Denial-of-service attacks are addressed by controlling the use of resources using policy programs (as shown in Section 8.8.1), and meta-interpreters, which control the execution of the LogicWeb program interpreter (augmented with policy programs). These meta-interpreters can be viewed as an outer sandbox as depicted in Figure 8.1, and are discussed further in Section 8.8.2.

Figure 8.1 The security model with two concentric sandboxes. The inner sandbox consists of the LogicWeb program interpreter and policy programs, and the outer sandbox consists of meta-interpreters.



The following sections describe how digital signatures are used with LogicWeb programs (Section 8.3), how security policies are specified (Section 8.4), the need to combine security policies during goal evaluation (Section 8.5), how policies are enforced by incorporating policy programs into the operational semantics of LogicWeb programs (Section 8.6), an implementation of the security model (Section 8.7), and how denial-of-service attacks are addressed (Section 8.8).

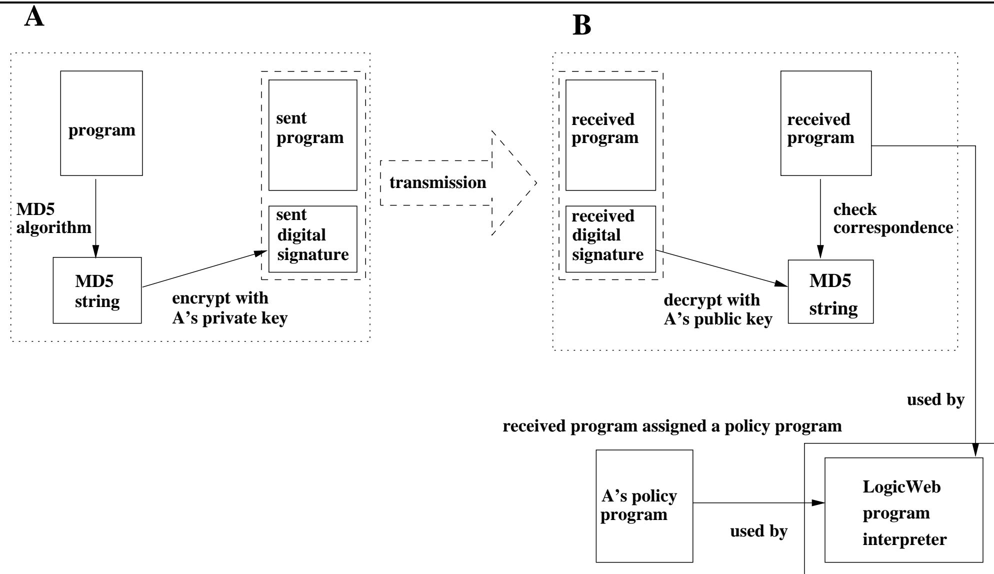
8.3 Digital Signatures for LogicWeb Programs

With PGP, information is encrypted using a secret *private key*, and decrypted using a corresponding *public key* which is distributed publically. PGP keys are used for authentication in the following way. Suppose A encrypts a message and sends it to B. B wants to ensure that the encrypted message is really from A, and that the message has not been tampered with. To do this, B attempts to decrypt the message using A's public key. If the decryption succeeds, it means that the message was encrypted using A's private key and that it has not been modified, and since the key is only known to A, the message must have been from A. If the decryption fails, B cannot conclude that A encrypted the message.

Similarly, PGP keys are used to sign documents. A PGP digital signature for a message is formed by associating the message (e.g., a program) with a signature. The signature is formed by first mapping the message to a string (actually, a single large number) using the MD5 algorithm [90], which almost uniquely identifies the message.[1] Then, the MD5 string is encrypted using the sender's private key. The encrypted MD5 string is the digital signature. The receiver of a signed message (the message and its signature) must first use the sender's public key to decrypt the signature producing a MD5 string, and then check that the message corresponds to the MD5 string, i.e. the message has not been modified in any

[1] In practice, no two different messages have been found to map to the same MD5 string though this is theoretically possible [90].

Figure 8.2 The use of a digital signature when sending a program from A to B, and the subsequent assignment of a policy program.



way.

On successful authentication of a signed LogicWeb program, the identity of the signatory is obtained. Then, a policy program is assigned to the LogicWeb program by consulting a database maintained in the system which maps signatories to policy programs. Unsigned programs or programs where authentication failed should not be trusted and must be restricted in its access to system resources. Figure 8.2 shows the use of a digital signature when sending a program from A to B, and the subsequent assignment of a policy program.

Digital signatures prove who sent the program and that the program was not altered either by error or design, and provides non-repudiation, which means the sender cannot easily disavow his signature on the program. It is faster to encrypt and decrypt an MD5 string than to directly encrypt and decrypt the entire program. Also, since the program is not encrypted, the program source is available even if authentication fails, and so, can still be executed (though with limited privileges). However, if confidentiality of the program is required during

transmission, the whole program must be encrypted.

PGP is used since it is difficult to break, equipped with public and private keys generation, widely available, supported on a range of operating system platforms, and one of the most popular encryption techniques. PGP public keys can be obtained either through personal communication between the LogicWeb system user and the program owners, or via the Web: PGP public keys which are increasingly being placed on homepages can be extracted and added to the system. The distribution of PGP keys is discussed further as an avenue for future work in Section 8.8.

8.4 Specifying Security Policies

Besides system calls, a LogicWeb program accesses system resources via context switching. The resources used via context switching are socket connections for downloading pages and local storage for programs. A policy program defines predicates for controlling the use of both the system calls and context switching:

- `valid_program(Type, URL)`: this predicate defines the programs (identified by the values for `Type` and `URL`) which the program associated with this policy can use via context switching. For example, a program that can only use programs from the domain `www.cs.mu.oz.au` is specified by the rule:

  ```
  valid_program(get, URL) :-
    contains(URL, "http://www.cs.mu.oz.au/").
  ```

  This means that a LogicWeb goal occurring in the program such as the following is disallowed by the policy (and will fail):

  ```
  ?- lw(get, "http://www.cs.rmit.edu.au/")#>h_text(Src).
  ```

- `valid_systemCall(P)`: this predicate defines the set of system calls the program is allowed to invoke. `P` is a term representing a predicate (see example below).

- `call_system(P)`: this predicate is a wrapper for the allowed predicates defined by `valid_systemCall/1`. `P` is a term representing a predicate (see example below). The purpose of this predicate is to enable indirect hooks to system calls. This can be used to implement more restrictive versions of system calls. For example, suppose that a system call is allowed to be used only once. Then, `call_system/1` is invoked on that system call, and upon its success, records that the call has been performed. `valid_systemCall/1` disallows system calls (i.e., imposes static restrictions) whereas `call_system/1` imposes run-time restrictions.

`valid_program/2` and `valid_systemCall/1` prevent integrity and privacy attacks by invalidating system calls required for such attacks. These predicates are called from the LogicWeb interpreter as shown later (in Section 8.7).

The following is an example of a policy program which permits the retrieval (using the GET method) of all URLs except `http://www.cs.mu.oz.au/~swloke/private.html`, and allows access to all system calls except `system/1` and `open/3` which can only be used to read a specific file:

```
.../* description of the policy program */
<LW_CODE>
valid_program(get, URL) :-
  URL \== "http://www.cs.mu.oz.au/~swloke/private.html".

valid_systemCall(open('/home/pgrad/swloke/lws/dump.txt', read, S)).
valid_systemCall(P) :-
  P \= open(_, _, _),
  P \= system(_).
```

```
call_system(P) :-
  built_ins:call_builtin(P).
</LW_CODE>
...
```

The LogicWeb system has been extended with the predicate `call_builtin/1` pre-defined in the SWI-Prolog module `built_ins`. `call_builtin/1` is a policy independent wrapper to built-in predicates and performs type checks to detect and report errors before finally invoking the built-in predicates. For example, for `open/3`, `FileName` must be an atom, `Mode` is either `read` or `write`, and `Stream` is a variable. The type checks increase the robustness of the system by preventing instantiation faults when the arguments of the wrong type are utilised.

```
call_builtin(open(FileName, Mode, Stream)) :-
  atom(FileName),
  ( Mode = read
  ; Mode = write
  ),
  var(Stream),
  open(FileName, Mode, Stream).
```

State information can be kept in policy programs to implement history-based policies. For example, the following policy program permits access to the Web if no files have been previously accessed.

```
file_accessed(no).  % default is no files have been accessed


valid_program(_, _) :-
  file_accessed(no).

call_system(P) :-
  P \= open(_, _, _),
  built_ins:call_builtin(P).
```

```
call_system(open(F, R, S)) :-
  built_ins:call_builtin(open(F, R, S)),
  (file_accessed(no) ->
    retract(file_accessed(no)), % record that a file has been accessed
    assert(file_accessed(yes))
  ;
    true
  ).
```

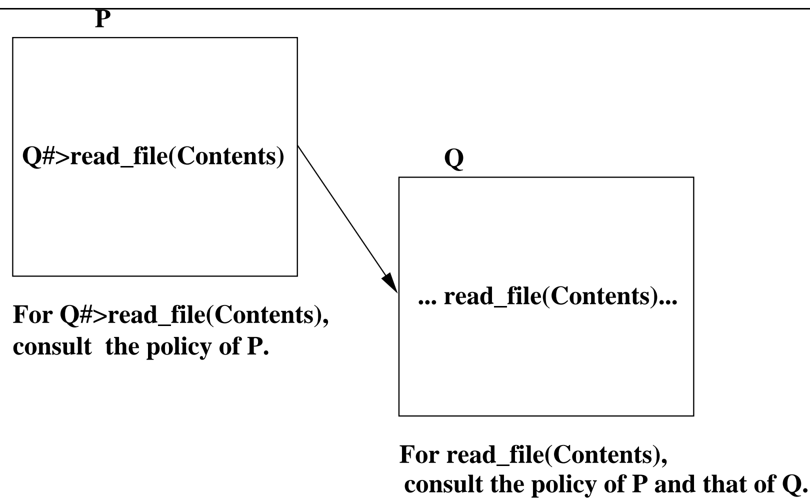
The fact `file_accessed(yes)` asserted into the policy program records that a file has been accessed. This policy prevents leaking of information contained in local files to another host by disallowing HTTP requests after a file has been read.

8.5 Combining Security Policies

An illegal system operation is performed by a program when it invokes a system call or downloads a program which it does not have the privilege for. A program can invoke goals in other programs either using context switching, or through being part of a LW-composition. A program must not be allowed to perform an illegal system operation by invoking goals in other more privileged LogicWeb programs. Hence, the security model for LogicWeb programs must ensure the following:

1. *Context switching must not transfer privileges between programs.* The system has to ensure that an untrusted program does not illegally access resources by invoking a goal in a trusted program. For example, if a program `P` is not allowed to read a file (as determined by its policy) while another trusted program `Q` can (as determined by `Q`'s policy), then `P` should not be able to call a goal like `Q#>read_file(Contents)` to read the file. This means that for such a LogicWeb goal, it is too simplistic to validate `read_file/1` using only the policy of `Q`. The policy of `P` must also be used to validate `read_file/1`.

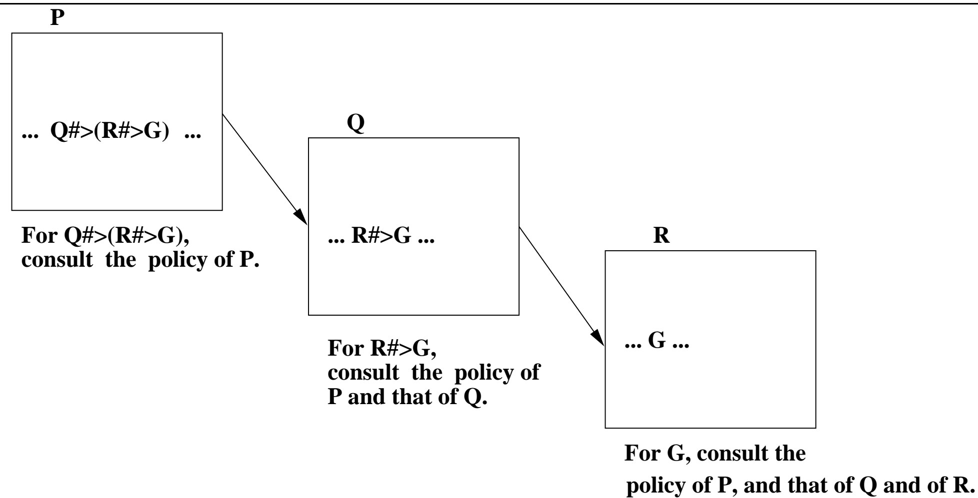
Figure 8.3 The invocation (with P's policy consulted) of the LogicWeb goal `Q#>read_file(Contents)` in P leads to the invocation of the goal `read_file(Contents)` in Q (where both the policy of P and that of Q must be consulted).



Similarly, a LogicWeb goal cannot be allowed to perform an illegal system operation because it is invoked in a trusted program. For example, if P can read a file which Q can not, then the goal `Q#>read_file(Contents)` invoked in P to read the file should not be allowed to succeed. This means that although P uses Q, P should not pass its privileges to Q, since the system trusts P but not Q. Hence, both the policy of P and that of Q must be consulted to determine if `read_file/1` should be allowed, as Figure 8.3 depicts.

When more than two programs are involved, a similar situation arises. For example, consider a goal `Q#>(R#>G)` invoked in the program P, where G is a system call. If G is performed because it is allowed by the policies of P and R, disregarding Q's policy, and suppose that Q is not allowed G, then Q is performing an illegal system operation through R. In the same way, R performs an illegal system operation if only the policies of P and Q are con-

Figure 8.4 The change in the context (represented by the rectangle) of goal evaluation starting from the goal `Q#>(R#>G)` in program `P` and ending in goal `G` in program `R`, and the policies for validating each subgoal.



sidered, and `R` is not allowed `G`. Disregarding the policy of any one of the programs potentially allows an illegal system operation. This means that, to determine if `G` should be allowed, the policy for every program must be consulted.

Figure 8.4 depicts the change in the context of goal evaluation starting from the goal `Q#>(R#>G)` in program `P` and ending in goal `G` in program `R`, and the policies that should be used for validating each subgoal. As Figure 8.4 suggests, the policies of all the programs involve in the evaluation of a goal starting with the first program (i.e., `P`) are required. Hence, the proof rules must incorporate a notion of the *current* set of policy programs which grows as goal evaluation progresses.

2. *LW-compositions must not transfer privileges between programs.* A program can invoke the clauses of another program in the LW-composition it is part of. For example, consider a goal `G` evaluated in the LW-composition `P + Q`: `(P + Q)#>G`. Suppose that the evaluation of `G` utilises a clause in `P` whose

subgoal invokes a clause in Q which, in turn, invokes a system call. Then, the system call must be allowed by both the policies of P and Q. If only the policy of P is considered, and P is trusted with the privilege of using that system call but Q is not, then Q performs an illegal system operation. On the other hand, if only the policy of Q is considered, and Q is trusted with the privilege of using that system call but P is not, then P performs an illegal system operation through Q.

In both cases, in order to prevent illegal system operations, the system call or download must be permitted by every policy program involved. Given a set of policy programs (e.g., {P,Q}), the system calls and downloads permitted by the LW-intersection of the LW-encapsulations of the policy programs in the set (e.g., @P * @Q) corresponds to the system calls and downloads permitted by every policy program in the set. LW-encapsulation is used to model the fact that each policy program must separately validate the system calls and downloads. LW-intersection is used to model the fact that all the policy programs must agree, i.e., a system call or download must be permitted by every policy program. For example, the following goal, on succeeding, means that `open/3` is permitted by both P and Q:

```
?- (@P * @Q)#>valid_systemCall(open(_,_,_)).
```

During goal evaluations, `valid_program/2` and `valid_systemCall/1` are invoked in this LW-composition as the following semantics show.

8.6 Enforcing Security Policies

This section defines a new derivation relation extending the operational semantics in Chapter 3 to use policy programs.

Policy programs are integrated into the execution model of LogicWeb programs. A goal is evaluated not only in the current context and with the current

set of created programs, but also with the current set of policy programs. The derivation relation in Chapter 3 is extended as follows:

> For any goal formula G and program expression E,
>
> $$\Sigma, S, E \vdash_{\theta}^{S'} G$$
>
> denotes the fact that there exists a *top-down derivation* of G in E starting with the set S of existing LogicWeb programs and the *ordered* set of policy programs Σ, and ending with computed answer substitution θ and created program set S'.

The proof rules which define this new relation is an extension of the rules in Chapter 3. Rules (3.1) to (3.9) are extended to form the corresponding rules for the new derivation relation using the following syntactic mapping:

> Replace every occurrence of an expression involving the derivation relation of the form
>
> $$S, E \vdash_{\theta}^{S'} G$$
>
> with the corresponding expression of the form
>
> $$\Sigma, S, E \vdash_{\theta}^{S'} G$$

For instance, the following rule is obtained by applying this mapping to rule (3.2) (conjunction):

$$\frac{\Sigma, S, E \vdash_{\theta}^{S'} G_1 \quad \wedge \quad \Sigma, S', E \vdash_{\gamma}^{S''} G_2\theta}{\Sigma, S, E \vdash_{\theta\gamma}^{S''} G_1, G_2}$$

In this rule, the same set of policy programs Σ is used for each of the conjuncts since the conjuncts occur in the same rule (and hence, the same program).

Rule (3.10) is extended to incorporate policy programs. To aid the discussion which follows, two functions are defined: one assigns a LogicWeb program to its

policy program, and the other maps a set of LogicWeb programs to their policy programs. Φ denotes the set of all policy programs used by the system.

Policy programs are permitted free access to system resources, and so, are not themselves assigned any policy program. This means that policy programs can download any other program and freely utilise system calls.

DEFINITION 5 (POLICY PROGRAM ASSIGNMENT)
The function

$$pol : (LWProgramIDs \setminus ids(\Phi)) \rightarrow \Phi$$

takes a program identifier (which is not that of a policy program) and returns a policy program for that program from Φ*.*

pol is not defined on policy programs, and the empty program ξ is not assigned a policy program.

DEFINITION 6 (POLICIES FOR A SET OF PROGRAMS)
The function

$$pols : \wp(LWProgramIDs \setminus ids(\Phi)) \rightarrow \wp(\Phi)$$

takes a set of (non-policy) program identifiers I *and returns the set of policy programs assigned to the programs in* I*.* $pols$ *is defined by:*

$$pols(I) = \{pol(i) \mid i \in I\}$$

$*\Sigma^{@}$ denotes the LW-intersection of the LW-encapsulation of the policy programs in Σ (i.e., if $\Sigma = \{P_1, ..., P_n\}$ then $*\Sigma^{@} = @P_1 * ... * @P_n$), and G denotes a goal. Evaluating a goal against $*\Sigma^{@}$ has the effect of evaluating the goal against each program in Σ separately.

Recall that rule (3.10) for context switching from Chapter 3 is the following:

$$\frac{I \subseteq ids(S') \quad \wedge \quad S', F' \vdash_{\theta}^{S''} G}{S, E \vdash_{\theta}^{S''} F \; \#\!> G} \tag{8.1}$$

where $F' = insertCC(F, E)$, $I = expids(F)$, and $S' = add_programs(S, I)$.

The new definition of rule (3.10) is then:

$$\dfrac{\begin{array}{l}\Sigma \neq \emptyset \quad \wedge \\ \left\{ \begin{array}{l} Where\ each\ \texttt{lw(Type}i\texttt{, URL}i\texttt{)} \in expids(F') \setminus ids(\Phi), \\ \quad \emptyset, S, \xi \vdash_{\gamma}^{S'} \texttt{*}\Sigma^{@}\texttt{\#>(valid_program(Type1, URL1),} \\ \qquad\qquad \ldots\texttt{,valid_program(TypeN, URLN))} \end{array} \right\} \wedge \\ I \subseteq ids(S') \quad \wedge \quad (pols(I \setminus ids(\Phi)) \cup \Sigma), S', F' \vdash_{\theta}^{S''} G \end{array}}{\Sigma, S, E \vdash_{\theta}^{S''} F \texttt{ \#> } G}$$

The new rule has been augmented with:

1. A test to determine if each non-policy program to be used in the new context is allowed by the current set of policy programs Σ. The test is done by invoking the predicate `valid_program/2` in $\texttt{*}\Sigma^{@}$ for each such non-policy program identified by `lw(Type`i`, URL`i`)` using a goal of the form:

$$\emptyset, S, \xi \vdash_{\gamma}^{S'} \texttt{*}\Sigma^{@}\texttt{\#>(valid_program(Type1, URL1),}\ldots\texttt{,} \\ \texttt{valid_program(TypeN, URLN))}$$

   `N` is the number of program identifiers in $expids(F') \setminus ids(\Phi)$. A LogicWeb goal is used when evaluating the `valid_program/2` goals in order to download the policy programs. Note that goal evaluation fails if not all the policy programs are downloaded, a policy program can invoke non-policy programs in its rules, and the evaluation of the `valid_program/2` goals begins with an empty policy program set. The rule for the case where Σ is empty is given below.

2. An extension of Σ with new policy programs. New policy programs are added to the front of the ordered set Σ:

$$pols(I \setminus ids(\Phi)) \cup \Sigma$$

The last program in the ordered set is, chronologically, the first policy program, and its significance is explained below.

The following variant of above rule caters for the case where Σ is empty. If Σ is empty, then no policy programs are employed, and no checks are made:

$$\frac{I \subseteq ids(S') \quad \wedge \quad pols(I \setminus ids(\Phi)), S', F' \vdash_{\theta}^{S''} G}{\emptyset, S, E \vdash_{\theta}^{S''} F \texttt{ \#> } G}$$

The above two rules are called *context switching rules*.

Two new rules called *system call rules* are added to specify how policy programs are used with system calls. The first rule specifies a test to determine if a goal is a valid system call, and invokes `call_system/1` in the last policy program in Σ. The last program in Σ is the policy program for the main program of the LogicWeb application, provided that the proof rules are utilised starting with a LogicWeb goal of the form $MainProgram$`#>`$Goal$ (see Section 8.7.3 on invoking the new interpreter). System calls may update state information (e.g., record that a file has been accessed as shown in Section 8.4). State information is maintained in one program (i.e., the main program's policy program) by always invoking `call_system/1` in the main program's policy program.

$$\frac{\begin{array}{l} \Sigma \neq \emptyset \quad \wedge \\ \emptyset, S, \xi \vdash_{\gamma}^{S'} \texttt{*}\Sigma^{@}\texttt{\#>valid_systemCall(}G\texttt{)} \quad \wedge \\ \Sigma = \Sigma' \cup P_n \quad \wedge \\ \emptyset, S', \xi \vdash_{\theta}^{S''} P_n\texttt{\#>call_system(}G\texttt{)} \end{array}}{\Sigma, S, E \vdash_{\theta}^{S''} G}$$

Substitutions resulting from calling `valid_systemCall/1` are discarded. Instead, the substitutions computed from `call_system/1` are used. Again, proofs occurring in policy programs proceed with an empty policy program set.

For the case where $\Sigma = \emptyset$, the rule is

$$\frac{\emptyset, S, E \vdash_{\gamma}^{S'} \texttt{built_ins:builtin(}G\texttt{)} \quad \wedge \quad \emptyset, S', E \vdash_{\theta}^{S''} \texttt{built_ins:call_builtin(}G\texttt{)}}{\emptyset, S, E \vdash_{\theta}^{S''} G}$$

A call to the built-in predicates in `built_ins` is represented by the following rule:

$$\frac{G\ succeeds\ with\ \theta}{\emptyset, S, E \vdash_{\theta}^{S} \texttt{built_ins:(}G\texttt{)}}$$

The evaluation of the goal in `built_ins` is outside the scope of the inference rules, but θ is assumed to be the computed answer substitution.

As will be shown below, the above new set of inference rules disallows any illegal system operation. *context sequence* and *illegal system operation* are first defined.

Goal derivation with LogicWeb goals may result in several sequences of context switches, each represented by a sequence of contexts. For example, the sequence of contexts in Figure 8.4 is

```
P, Q, R
```

The only rules which change the context are the context switching rules.

DEFINITION 7 (CONTEXT SEQUENCE)

Given a sequence of applications of inference rules in a top-down derivation, suppose that there are $n \perp 1$ applications of the context switching rules, and that the ith application of such rules causes the context to switch from E_i to E_{i+1}, where $1 \leq i \leq n \perp 1$, then the $context\ sequence$ is the sequence of contexts $E_1, \ldots, E_n$.

Informally, given a context sequence $E_1, \ldots, E_n$, an illegal system operation is performed in the context E_i if a system call, or the use of a program in context switching, is disallowed by the policy of some program in E_i, but is attempted in E_i or in some later context.

DEFINITION 8 (ILLEGAL SYSTEM OPERATION)

Given a context sequence $E_1, \ldots, E_n$, for some E_i, $P \in pols(expids(E_i) \setminus ids(\Phi))$, $i \leq j \leq n$, and system call G, an illegal system operation is performed in E_i if

the goal `call_system(`G`)` *is invoked when the context of* G *is* E_j *but* $\emptyset, S, P \vdash_{\theta}^{S'}$ `valid_systemCall(`G`)` *does not hold for any* S, S', *and* θ, *or the oracle function is called on the identifier* `lw(Type, URL)` *when the current context is* E_j *but* $\emptyset, S, P \vdash_{\theta}^{S'}$ `valid_program(Type, URL)` *does not hold for any* S, S', *and* θ.

The context switching rules guarantee that all policy programs which must be consulted during a derivation are present.

LEMMA 8.6.1
Given the context sequence $E_1, \ldots, E_n$ *for a top-down derivation, for any* $i \in \{1, \ldots, n\}$, *at the node with context* E_i, *the current set of policy programs* Σ *contains the policy programs of all the non-policy programs occurring in* $E_1, \ldots, E_i$.
PROOF: *The proof is by induction on* i. *For* $i = 1$, *the goal evaluation starts in the empty program* ξ *with* $\Sigma = \emptyset$. *For* $i \geq 2$, *(by the inductive hypothesis) when the goal is evaluated in* E_{i-1}, Σ *contains the policy programs of all the non-policy programs occurring in* $E_1, \ldots, E_{i-1}$. E_{i-1} *changes to* E_i *using one of the context switching rules. This rule updates* Σ *(say, to* Σ'*) by adding to* Σ *the policy programs of the non-policy programs occurring in* E_i. *Hence, the goal evaluation continues in* E_i *with* Σ' *containing all the policy programs for the non-policy programs in* $E_1, \ldots, E_i$.
■

The theorem below implies soundness of the security model: all goals which have a successful derivation have the desired security property of not performing an illegal system operation (as given by Definition 8).

THEOREM 8.6.1 (SAFETY PROPERTY)
No illegal system operation is performed in any context during a top-down derivation using the above rules.
PROOF: *Given the context sequence* $E_1, \ldots, E_n$, *for any* $i \in \{1, \ldots, n\}$, *suppose that an illegal system operation was performed in* E_i, *say when the current con-*

text was E_j, where $i \leq j \leq n$, the set of policy programs was Σ and the set of created programs was S. To perform this operation either (1) the system call rule for $\Sigma \neq \emptyset$, or (2) the new context switching rule for $\Sigma \neq \emptyset$ must have been used. In case (1), according to the definition of the system call rule used, since `call_system(G)` *was invoked, the goal*

$$\emptyset, S \cup \Sigma, \star\Sigma^{@} \vdash_{\gamma}^{S'} \texttt{valid_systemCall(G)}$$

must have succeeded. In case (2), suppose that the LogicWeb goal was F#>G and $F' = insertCC(F, E_j)$, and the oracle function was called on the identifier `lw(Type, URL)` $\in expids(F') \setminus ids(\Phi)$*, then according to the definition of the context switching rule used, the goal*

$$\emptyset, S \cup \Sigma, \star\Sigma^{@} \vdash_{\gamma}^{S'} \texttt{valid_program(Type, URL)}$$

must have succeeded. But by the definition of an illegal system operation, for some $P \in pols(expids(E_i) \setminus ids(\Phi))$, in case (1), $\emptyset, T, P \vdash_{\theta}^{T'}$ `valid_systemCall(G)` *does not hold for any T, T', and θ, and in case (2), $\emptyset, T, P \vdash_{\theta}^{T'}$* `valid_program(Type, URL)` *does not hold for any T, T', and θ. In either case, it must have been that $P \notin \Sigma$. By the above lemma, Σ contains the policy programs of all the non-policy programs occurring in $E_1, \ldots, E_j$, namely, $\Sigma \supseteq pols(expids(E_i) \setminus ids(\Phi))$ which means $P \in \Sigma$, and hence, there is a contradiction.*

■

8.7 Implementation

8.7.1 A New Interpreter

The inference rules presented in the previous section provides the basis for an interpreter for evaluating goals in the presence of policy programs. Program 8.1 shows how the interpreter in Chapter 4 has been extended. In

the first clause of demo/3, establish_context/6 extends the definition of establish_context/3 in Chapter 4 by carrying the current list of policy program identifiers, and returning new policy program identifiers for the programs in the new context. In the second last clause of establish_context/6, add_policyID/3 retrieves the policy program identifier from policyID/2. The assignment of policy programs is explained below. In the last clause of establish_context/6, the goal allowed_programs(OPL, C) checks that the programs in the expression C are allowed to be used by every policy program mentioned in OPL. allowed_programs/2 (not given in Program 8.1) is defined by calling allowed_program/2 for each program identifier in C.

The second clause of demo/3 checks for and invokes an allowed system call. In allowed_systemCall/2, valid_systemCall/1 is invoked in each policy program individually (or equivalently, in the LW-intersection of the LW-encapsulations of the policy programs). Policy programs known to the system are recorded in pgpID_to_policyID/2 described below. invoke_systemCall/2 invokes the system call in the last (or chronologically the first) program mentioned in the list PL of policy program identifiers.

The third clause of demo/3 allows policy programs to invoke goals in the SWI-Prolog module built_ins.

The next three clauses of demo/3 implements the vanilla meta-interpreter. In the sixth clause of demo/3, select_clause/3 extends the definition of ::/2 in Chapter 4 to carry as an argument the list of policy program identifiers for use in LW-encapsulation.

8.7.2 Installing Programs

The procedure for installing a program is slightly more complex than in Chapter 4 because for each downloaded program, a policy program is assigned to it. The following predicate specifies download/2:

Program 8.1 The interpreter for pure LogicWeb programs modified to use policy programs. This program extends Program 4.2.

```
 % demo/3 with LogicWeb goal
demo(PL, E, F#>G) :-
  establish_context(PL, F, E, PL, NPL, F1),
  demo(NPL, F1, G).
demo(PL, E, G) :-
  allowed_systemCall(PL, G), invoke_systemCall(PL, G).
demo(_PL, P, built_ins:G) :-
  pgpID_to_policyID(_, P), call(built_ins:G).
demo(_PL, _E, true).
demo(PL, E, (A, B)) :- demo(PL, E, A), demo(PL, E, B).
demo(PL, E, A) :- select_clause(PL, E, (A :- B)), demo(PL, E, B).

 % establish a context
establish_context(OPL, E + F, C, PL, PL2, E1 + F1) :-
  establish_context(OPL, E, C, PL, PL1, E1),
  establish_context(OPL, F, C, PL1, PL2, F1).
establish_context(OPL, E * F, C, PL, PL2, E1 * F1) :-
  establish_context(OPL, E, C, PL, PL1, E1),
  establish_context(OPL, F, C, PL1, PL2, F1).
establish_context(OPL, E / P, C, PL, PL2, E1 / P1) :-
  establish_context(OPL, E, C, PL, PL1, E1),
  establish_context(OPL, P, C, PL1, PL2, P1).
establish_context(OPL, @E, C, PL, PL1, @E1) :-
  establish_context(OPL, E, C, PL, PL1, E1).
establish_context(OPL, (/)<>(E, L), C, PL, PL2, (/)<>(E1, L1)) :-
  establish_context(OPL, E, C, PL, PL1, E1),
  establish_contextL(OPL, L, C, PL1, PL2, L1).
establish_context(OPL, Op<>L, C, PL, PL1, Op<>L1) :-
  establish_contextL(OPL, L, C, PL, PL1, L1).
establish_context(OPL, lw(T, U), _C, PL, NPL, lw(T, U)):-
  allowed_program(OPL, lw(T, U)), download(T, U),
  add_policyID(lw(T, U), PL, NPL).
establish_context(OPL, (#), C, PL, PL, C) :-
  allowed_programs(OPL, C).

establish_contextL(_OPL, [], _C, PL, PL, []).
establish_contextL(OPL, [E|Es], C, PL, PL2, [E1|Es1]) :-
  establish_context(OPL, E, C, PL, PL1, E1),
  establish_contextL(OPL, Es, C, PL1, PL2, Es1).

 % predicate to ensure that only allowed system calls are invoked
allowed_systemCall([], G) :-
  built_ins:builtin(G).
allowed_systemCall([Pol|Pols], G) :-
  demo([], empty, Pol#>valid_systemCall(G)),
  allowed_systemCall(Pols, G).
```

Program 8.1 (Continued)

```
 % predicate to invoke system calls
invoke_systemCall([], G) :- !,
  built_ins:call_builtin(G).
invoke_systemCall(PL, G) :-
  last(P, PL), demo([], empty, P#>call_system(G)).

 % adding a policy program identifier
add_policyID(Id, PL, PL) :-
  pgpID_to_policyID(_, Id), !.% no policy program for policy programs
add_policyID(Id, PL, NPL) :-
  policyID(Id, NewPolicyId),
  (member(NewPolicyId, PL) ->
     NPL = PL
  ;
     NPL = [NewPolicyId|PL]
  ).

 % predicate to ensure that only allowed programs are downloaded
allowed_program([], _Id).
allowed_program([Pol|Pols], lw(Type, URL)) :-
  demo([], empty, Pol#>valid_program(Type, URL)),
  allowed_program(Pols, lw(Type, URL)).

 % definition of select_clause/3
select_clause(_PL, lw(Type, URL), A :- B) :-
  lw(Type, URL)::(A :- B).
select_clause(PL, E + _F, A :- B) :-
  select_clause(PL, E, A :- B).
select_clause(PL, _E + F, A :- B) :-
  select_clause(PL, F, A :- B).
select_clause(PL, E * F, A :- (B,C)) :-
  select_clause(PL, E, A :- B), select_clause(PL, F, A :- C).
select_clause(PL, E / P, A :- B) :-
  select_clause(PL, E, A :- B), not defined(A, P).
select_clause(PL, @E, A :- true) :- demo(PL, E, A).

select_clause(PL, (/)<>(E, []), A :- B) :-
  select_clause(PL, E, A :- B).
select_clause(PL, (/)<>(E, [P|Ps]), A :- B) :-
  select_clause(PL, (/)<>[(E / P)|Ps], A :- B).

select_clause(PL, _Op<>[E], A :- B) :-
  select_clause(PL, E, A :- B).
select_clause(PL, Op<>[E1,E2|Es], A :- B) :-
  C =.. [Op, E1, E2], select_clause(PL, Op<>[C|Es], A :- B).

defined(A, P) :-
  functor(A, Functor, Arity), functor(H, Functor, Arity),
  P::(H :- _B).
```

```
download(Type, URL) :-
  created(Type, URL), !.                 % program already exists
download(Type, URL) :-                   % program does not exist
  retrieve(Type, URL, Contents),         % retrieve from the Web
  create_program(Type, URL, Contents),   % create the program
  assign_policyID(Type, URL, Contents).  % assign a policy program
```

`assign_policyID/3` uses the following procedure to assign a policy program identifier to a downloaded program. A downloaded HTTP response object is first authenticated to determine the identity of the signatory. The system keeps a database of `pgpID_to_policyID/2` facts which maps the identity of a signatory to the identifier of a policy program (chosen by the LogicWeb system user). For example, a policy program for Seng is recorded by mapping Seng's PGP ID to the identifier of the policy program:

```
pgpID_to_policyID('Seng W. Loke <swloke@cs.mu.oz.au>',
          lw(get, "http://www.cs.mu.oz.au/~swloke/my_policy.html").
```

The first argument is Seng's PGP ID, and the second argument is a policy program identifier. If the HTTP response object is not PGP signed or authentication fails, the created program will be assigned the default policy program for untrusted programs, i.e. the policy program for `unknown` stated in `pgpID_to_policyID/2`:

```
pgpID_to_policyID(unknown,
      lw(get, "http://www.cs.mu.oz.au/~swloke/default_policy.html").
```

Otherwise, a policy program for the signatory specified by `pgpID_to_policyID/2` is assigned to the program. A policy program assignment is recorded by asserting a `policyID/2` fact into the system. The fact is not asserted into the program itself since the location of policy programs must be protected from applications and `policyID/2` must not be redefined.

```
assign_policyID(Type, URL, Contents) :-
  determine_policyID(URL, Contents, PolicyID),
  record_policyID(Type, URL, PolicyID).
```

```
determine_policyID(URL, Contents, PolicyID) :-
  (pgp_signed(URL) ->                    % if digitally signed
    authenticate(Contents, SignatoryID), % try authentication
    pgpID_to_policyID(SignatoryID, PolicyID)
  ;
    pgpID_to_policyID(unknown, PolicyID)
  ).

record_policyID(Type, URL, PolicyID) :-
  (pgpID_to_policyID(_, lw(Type, URL)) ->     % if policy program
    true         % lw(Type, URL) is not assigned a policy program
  ;
    assert(policyID(lw(Type, URL), PolicyID))
  ).
```

`pgp_signed/1` checks if the contents are digitally signed by inspecting the URL for the extension ".lwpgp.html". Digitally signed LogicWeb programs are assumed to end with this extension. `authenticate/2` performs the authentication by invoking the PGP system's authentication procedure on the `Contents`, which contains HTML text and its digital signature. This procedure is depicted in Figure 8.2 on side B. PGP extracts the digital signature from `Contents` and attempts to decrypt it to obtain the MD5 string. PGP attempts to decrypt the signature using its collection of public keys. Each of these public keys has been added to the PGP system by the LogicWeb system's user and is labelled with a PGP ID. On successful decryption, PGP checks the HTML text from `Contents` against the MD5 string, and returns the PGP ID labelling the key that decrypted the signature. `SignatoryID` is instantiated with this PGP ID which is the signatory's PGP ID. On authentication failure (i.e., if the signature is not decrypted or if the MD5 string does not match the HTML text), `SignatoryID` is instantiated to `unknown`. The PGP system is invoked externally from Prolog.

8.7.3 Invoking the New Interpreter

In order to utilise the new interpreter, calls to `demo/2` in Program 4.1 of Chapter 4 are modified as follows:

> Replace each call of the form `demo(empty,` $LogicWebGoal$`)` by a call of the form `demo([], empty,` $LogicWebGoal$`)`.

Goal derivation begins with the empty context and without any policy programs.

8.8 Control of Resource Usage

This section addresses denial-of-service attacks by controlling the usage of an allowed operation or resource using policy programs and meta-interpreters.

8.8.1 Resource Control Using Policy Programs

Resource usage can be monitored by keeping contextual or state information within a policy program, allowing decisions to access resources to be made based on execution history. For example, a limit can be imposed on the frequency of system calls. The following definition of `call_system/1` permits up to ten files to be opened at a time, thereby limiting the number of file descriptors allocated.

```
call_system(P) :-
  P \= open(_, _, _),
  P \= close(_),
  built_ins:call_builtin(P).
call_system(open(F, R, S)) :-       % opening a file
  open_count(N),
  N < 10,
  increment_open_count,             % count an open file
  built_ins:call_builtin(open(F, R, S)).
```

```
call_system(close(S)) :-            % closing a file
  open_count(N),
  N > 0,
  decrement_open_count,             % decrement count of opened files
  built_ins:call_builtin(close(S)).
```

`increment_open_count/0` and `decrement_open_count/0` updates the value stored by `open_count/1`. Similarly, the number of bytes which can be written into a file can be restricted.

8.8.2 Resource Control Using Meta-interpreters

Resource control not specific to a particular program or application is needed. For instance, a goal evaluation involving multiple programs can go into an infinite loop. But, as pointed out in [163], denial-of-service attacks are not as severe as the other attacks since the user can always hit the "kill key" and exit the system. However, more graceful termination of goal evaluation under such circumstances is preferred, at least allowing the system and the application to remain up. Control of execution behaviour can be incorporated via meta-interpreters [184, 185]. The use of meta-interpreters for loop checking and detecting two resource limits are described below.

8.8.2.1 Loop Checking

Loop checking uses two meta-interpreters: `solve_ad/2` which stores as one of its arguments the ancestor goals and the recursion depth during goal evaluation, and `solve_t/1` which terminates goal evaluation whenever some pre-defined condition (called a *termination condition*) holds. By using `solve_ad/2` to evaluate `demo/3` goals, ancestor goals and recursion depth can be recorded during goal evaluation. By comparing the current goal with ancestor goals, loops within LogicWeb programs, and loops involving multiple programs can be detected (such as a goal `G` in `P` calling a goal in `Q` which, in turn, calls `G` in `P` again).

A loop is detected if the current goal is a variant of an ancestor goal (i.e., the goals subsume each other) and the goals are evaluated in the same context, or if a recursion depth limit is exceeded. Goal evaluation is terminated when a loop is detected. The termination condition is checked in `solve_t/1` which executes `solve_ad/2`. Other loop checking techniques are studied in [29].

To utilise `solve_ad/2` and `solve_t/1`, instead of directly invoking the interpreter `demo/3` as explained in the previous section, `demo/3` is invoked in `solve_ad/2`, and `solve_ad/2` is invoked in `solve_t/1`:

`solve_t(solve_ad(0 - [], demo([], empty, ` *LogicWebGoal*`)))`

Program 8.2 shows a version of `solve_t/1` simplified for pure Prolog. Termination conditions are encoded in the predicate `terminate/1`, which is invoked in the second clause of `solve_t/1` on every goal to find a terminating goal. Goal evaluation terminates when `terminate/1` succeeds.

Program 8.2 A version of `solve_t/1` for pure Prolog.

```
solve_t(A) :-
  solve_t(A, _).
solve_t(true, _).
solve_t(A, T) :-
  terminate(A), !, T = terminated. % check termination conditions
solve_t((A, B), T)  :-
  solve_t(A, T),
  (T == terminated ->
     true
  ;
     solve_t(B, T)
  ).
solve_t(A, T) :-
  clause(A, B), solve_t(B, T), (T == terminated, ! ; true).
```

To do loop checking, a goal `spy_point/2` is inserted into `solve_ad/2`. Program 8.3 shows a version of `solve_ad/2` simplified for pure Prolog. `copy_term/2` is a SWI-Prolog built-in predicate for creating a copy of a term.

The following shows the termination condition for loop checking:

Program 8.3 A meta-interpreter for pure Prolog with an argument carrying the recursion depth and a list of ancestor goals.

```
solve_ad(_D_Ancs, true).
solve_ad(D - Ancs, (A, B)) :-
  solve_ad(D - Ancs, A),
  solve_ad(D - Ancs, B).
solve_ad(D - Ancs, A) :-
  copy_term(A, A1),           % make a copy of the term A
  clause(A, B),
  spy_point(D - Ancs, A),     % spy_point inserted
  D1 is D + 1,                % increment recursion depth
  solve_ad(D1 - [A1|Ancs], B).
```

```
terminate(spy_point(Depth - Ancs, demo(_, E, G))) :-
  (
    member(demo(_, E1, G1), Ancs),
    E = E1,          % same context
    variant(G, G1),  % goals are variant of one another
    write('loop found')
  ;
    Depth > 40,
    write('maximum recursion depth exceeded')
  ).
```

`demo(_, E, G)` is the current goal, `Depth` is the current recursion depth, and `Ancs` is a list of ancestor goals. Note that the condition `E = E1` will not detect loops where the context grows indefinitely. To detect such loops, the size of the context can be compared against a preset limit.

8.8.2.2 Two Resource Limits

Besides loop checking, two resource counts, the number of LogicWeb programs downloaded and the number of clause applications, are used in termination conditions in `solve_t/1`. The first count is allowed to persist across `solve_t/1` invocations, since programs are not automatically purged after each top-level

query evaluation. The second resource count is reset at each invocation of `solve_t/1`, placing a resource bound on the evaluation of each top-level query.

For each invocation of the predicate `download/2` which downloads a page and translates it into a LogicWeb program, the `program_count/1` storing the number of LogicWeb programs created is checked against a preset limit. `create_program/3` (called by `download/2`) is modified to not only translate a page into a program but also to increment `program_count/1` each time a new program is created. For each use of the predicate `clause/2` to retrieve clauses of the LogicWeb interpreter, the `clause_count/1` storing the number of clause applications is checked against a preset limit.

The following two clauses extends `terminate/1` defining termination conditions based on the built-in predicates invoked by `solve_ad/2` (not shown in Program 8.3):

```
terminate(invoked_builtin(download(_,_))) :-
  program_count(N),
  N > 100,           % permit up to 100 programs only
  write('maximum LogicWeb program count exceeded').
terminate(invoked_builtin(clause(_,_))) :-
  retract(clause_count(N)),
  N1 is N + 1,
  assert(clause_count(N1)),
  (N > 500 ->
    writef('maximum clause count exceeded')
  ;
    !,
    fail
  ).
```

The termination checks can be implemented more efficiently by directly modifying `demo/3`. But using multiple meta-interpreters reduces the complexity of a single meta-interpreter allowing each functionality to be separately implemented and introduced. The efficiency costs are not severe if an application involves frequent Web requests, and hence, would spend most of its wall-clock execution

time interacting with the Web. Moreover, partial reduction techniques [185] for translating meta-interpreters into specialised forms can be explored to remove levels of interpretation.

8.9 Comparison with Security Models in Other Mobile Code Systems

Numerous languages have been used in mobile code systems and security in these systems is an active research area [202, 43, 61]. Language features and the form in which code is transmitted greatly influence the design of a security model. Below, we review security models with ideas in common with the LogicWeb security model, namely, models for interpreted languages, and models which use the idea of security policy modules and authentication.

8.9.1 Security Models in Two Interpreted Languages

8.9.1.1 Safe-Tcl

Tcl is an interpreted imperative language. Access to system resources is via permitted commands of the interpreter. In the Safe-Tcl security model [163], security is enforced by making dangerous commands unavailable to scripts running in a *safe interpreter*. Potentially dangerous operations such as opening sockets can still be carried out via wrappers or aliases which invoke dangerous commands. The wrappers ensure that the commands are used in a controlled manner (e.g., only socket connections to some hosts are permitted). The security policy in the Safe-Tcl model is the set of all commands available to scripts.

The security model in the LogicWeb system is partly motivated by their model in that their commands correspond to the set of valid system calls and `call_system/1` corresponds to wrappers for their commands. A safe interpreter in Safe-Tcl corresponds to the LogicWeb program interpreter with appro-

priate policy programs. However, the Safe-Tcl security model has not incorporated authentication.

The Safe-Tcl model allows Tcl programs running in different interpreters (each with its own security policy) to communicate. Such communication effectively composes the security policies of the Tcl programs, i.e. a program can use the privileges of the other. Although the dangers of this is highlighted in [163], the paper did not attempt to address this problem in a structured way. In the LogicWeb model, the combined use of policy programs addresses the dangers of one program using other programs with different access privileges.

8.9.1.2 Java Applets

Java applets are transmitted in byte-code format, which is then executed by an interpreter on the local host. This contrasts with Safe-Tcl and LogicWeb where source code is transferred.

Security for Java applets is based on strong type checking, pointer-free code, garbage collection, access control for class variables and methods, and distinct namespaces for packages [18]. Numerous security loopholes have been found in implementations of the security architecture in Java due mainly to implementation flaws, but there has been recent work on strategies to securely support Java applets [205].

LogicWeb programs benefit from Prolog's automatic garbage collection and pointer-free code from the security viewpoint. The LogicWeb system does not yet support concurrency or sophisticated GUI programming. Hence, the LogicWeb security model does not need to deal with multi-threading or control screen resources (e.g., windows). Java programs are typed, and the Java language itself is designed to enforce type safety. This means the compiler ensures that class methods and programs do not access memory in ways that are inappropriate (e.g., through type conversions) [188]. LogicWeb programs are not explicitly typed and there is no class abstraction. Type conversions (e.g., strings to atoms)

are done via system calls, and hence, dangers of type conversions are avoided. There are two trust levels for classes in the current Java security model: trusted classes are local and part of the Java system, and untrusted classes are downloaded. Multiple levels of trust are possible with signed LogicWeb programs.

Security for Java applets, as implemented in Microsoft's Internet Explorer 3.0 and Netscape Navigator 3.0, disallows applets from reading or writing local files, or establishing network connections except to the originating host. However, there are ways around these problems, including the use of server-side databases and proxy servers. In the direction of using encryption, there has been recent work on supporting the digital signing of JAR (Java archive) files (e.g., as supported in JDK 1.1.5[2]), which bundles Java code and related files into one [188]. Digital signing of JAR files prevent code tampering if data from the server is intercepted. In a similar way, a collection of programs making up a LogicWeb application can be bundled, signed, and transported as one archive file.

8.9.2 Security Policy Modules in Two Mobile Code Systems

8.9.2.1 SERC's Safer Erlang (SSErl)

The idea of policy programs is motivated by the policy modules implemented for the safe execution of programs in SSErl, a declarative language for programming concurrent and distributed systems [44, 45]. In their framework, a policy module is not associated with a particular program as in the LogicWeb security model, but with a SSErl node, a platform in which multiple processes may run concurrently. System operations are performed via built-in functions. A policy module specifies the allowable built-in functions for all the processes running at the node.

In contrast to SSErl, execution of LogicWeb programs is single-threaded though the thread of execution can proceed from one program to another. At

[2]JDK 1.1.5 (final version) is available at `http://java.sun.com/products/jdk/1.1/`.

any one time, at most one LogicWeb application is running. The overall policy for the running application is represented by the current set of policy programs, and is determined dynamically (since it is generally not possible to determine *a priori* which programs an application will use).

8.9.2.2 Java Aglets

The Aglet workbench [108], where Java programs (called *aglets*) are transferred between hosts as mobile agents, uses a policy-based security model. A specialised language for writing aglet security policy modules is proposed which makes use of specific roles such as the aglet's manufacturer, owner, execution platform (e.g., the URL of the host), domain of the execution platform (e.g., belonging to a company), and administrator of the domains.

In contrast to a policy program which encodes a mode of resource usage, an aglet policy specifies how to assign resources to an aglet. The LogicWeb system's equivalent to an aglet policy would be a policy for assigning policy programs to LogicWeb programs. In the current implementation, the assignment of policy programs is based solely on the identity of the LogicWeb program's signatory.

8.9.3 Authentication in Two Mobile Code Systems

8.9.3.1 Agent Tcl

Authenticating Tcl programs using PGP has been used in a Tcl-based agent architecture Agent Tcl [95]. The PGP system is invoked externally from the Agent Tcl system in the same way as in the LogicWeb system keeping the Agent Tcl system simpler and flexible (the encryption mechanisms can be easily replaced). Depending on the identity of the agent's owner, an Agent Tcl program is assigned resources such as CPU time, windows, the file system, external programs, and network connections.

8.9.3.2 ActiveX

ActiveX controls [80] are programs (in native x86 code) that can be executed on the client-side from Web pages. A digital signature is attached to each ActiveX control which identifies the control's original author. The advantage of the signature is that since the author is known, an ActiveX control (whose author is trusted such as a major vendor) can have freedom in accessing resources. Hence, trusted ActiveX controls have the ability to perform more useful tasks in contrast to Java applets.

Similar to ActiveX controls, a digital signature is attached to LogicWeb programs. But unlike LogicWeb programs which is sandboxed by policy programs and meta-interpreters, a trusted ActiveX control is not restricted in any way and has access to all operating system services. Policy programs permit levels of trust to be expressed. For example, a set of policy programs ordered according to increasing privileges can be defined, whereas an ActiveX control is either not permitted to execute or permitted to run with free access to system resources. Since an ActiveX control is native code, a sandbox mechanism for ActiveX controls is more difficult to implement [159]. Moreover, meta-programming combined with the program source permits more meaningful analysis and verification of program properties than binary code verification [194].

8.10 Summary and Future Work

Security is a major concern for a system that executes downloaded code. This chapter has described a security model for protecting the client host from integrity, privacy, and denial-of-service attacks by LogicWeb programs (given in Section 8.1), and its implementation in the LogicWeb system. The security model can be viewed as dynamically extending the LogicWeb system with selected policy programs. Resource control using resource counts and loop checking are performed by meta-interpreters. The security model is sufficiently simple to be eas-

ily implemented and trusted. Authentication is used to provide varying trust levels, and the security model is specified using proof rules as part of the operational semantics of the language.

From a security viewpoint, declarative programming languages with a simple formal semantics have an advantage for mobile code over languages without formal semantics, since the formal semantics facilitates reasoning over the program's execution. For instance, as reported in [69], a language weakness of Java from the security viewpoint is its lack of formal semantics. The Java language has neither a formal semantics nor a formal description of its type system. There is no formal meaning for a Java program, and so, formal reasoning about Java and the security properties of the Java libraries written in Java is not possible. The operational semantics of the LogicWeb language permits a straightforward proof of the soundness of the model with respect to illegal system operations. The claim that formal language semantics can be used for security have also been made in [147], which advocates the use of rewriting logic as the basis for a mobile code programming language so that security properties can be formally verified, and by Volpano, in [203], who argues for the development of provably-secure remote evaluation (including downloaded code) programming languages whose semantics permit formal proof of security properties.

The security model for the LogicWeb system can be improved. As noted in [163], it may be difficult to distinguish between legitimate behaviours and denial-of-service attacks. For instance, it is easy to introduce resource bounds but not easy to state an appropriate value for the bound. If the value is too low, a useful task may be terminated prematurely, but if too high, resource wastage occurs. Perhaps a means of tackling this problem is to involve the user's judgement by providing query-the-user facilities via a meta-interpreter. Trace information supplied by a meta-interpreter can help the user judge. The user can also use such trace information to detect unwanted behaviours such as transferring data between sites using the client host's resources.

Some Prolog systems such as SICStus Prolog have a timeout facility on goal evaluation, i.e. a goal evaluation is terminated if its evaluation exceeds a given time period [106]. This can be incorporated as a hard limit after other checks:

```
time_out(solve_t(solve_ad(0 - [], demo([], empty, LogicWebGoal))),
        Time, Result)
```

The goal timeouts after `Time` milliseconds with `Result` being `timeout` or `success`.

Policy programs can be assigned based on the more general idea of credentials instead of digital signatures only. For instance, a program may be allowed certain privileges not only because of the identity of its signatory, but because it has sufficient advocates. Recent work [176] on using logic programs to specify and check electronic credentials would be helpful.

Policy programs simplify the administration of privileges to applications. A more complex policy assignment procedure involving multiple roles such as the program's manufacturer, owner, execution platform, domain of the execution platform, and domain administrators would involve composing policy programs of different parties. LW-composition operators could be explored for creating the appropriate security policies dynamically during policy assignment.

Apart from the LogicWeb system's user, policy programs used by the system can also be created by trusted parties (e.g., organisations). Authentication of policy programs then becomes necessary to ascertain their origin.

A limitation of the model is that the system must already know all the required public keys for authenticating incoming programs. An automatic distribution mechanism for PGP public keys is needed. For example, the LogicWeb system can extract required public keys from homepages given their URLs or query Internet PGP public key servers[3] [90]. But this requires trusting that the information has not been tampered with and is reliable. The use of credentials,

[3] One example of a PGP public key server is found at `http://www-swiss.ai.mit.edu/~bal/keyserver.html`.

i.e. certification by other authorities, will be required.

The LogicWeb security model is conservative in that a trusted program is not allowed to transfer its privileges to an untrusted program it is using. Besides resource usage privileges, another kind of privilege can be introduced, which is the right to transfer privileges. This will be useful since the system may not know the signatories of all the programs that a trusted program uses. The policy program must also specify which privileges are transferable, and whether the right to transfer privileges is itself transferable.

Chapter 9

Comparison With Related Work

This chapter reviews work related to LogicWeb. Section 9.1 surveys and compares work concerned with the relationship between logic programming and the Web, placing LogicWeb in the context of such work. Section 9.2 compares the LogicWeb language to other Web programming languages (not based on logic programming). Finally, Section 9.3 relates the LogicWeb model to other logic-based hypertext models.

9.1 Logic Programming Technology for the Web

The relationship between logic programming and the Web is a young area, where most research work has appeared only in the last few years as shown by the recent workshops [196, 66, 63, 78].

As mentioned in Chapter 2, the Web uses a client-server communications model. Logic programming technology has been used to build client-based and server-side systems. Section 9.1.1 considers logic programming systems which are client-based (which includes the LogicWeb system), and Section 9.1.2 examines server-side systems. Finally, Section 9.1.3 briefly describes more expressive Internet-based logic programming systems, which utilise peer-to-peer commu-

nications models.

9.1.1 Client-side Systems

Client-side code can be more closely integrated with the browser and so can offer more sophisticated user interfaces than server-side solutions. Also, once client-side code is downloaded it does not need to communicate with the server, thereby avoiding networking problems that can affect server-side applications. Only the code which is needed for the current task has to be downloaded and, since it is a copy of the original, it can be changed or combined with other code without affecting the original.

Security is a major concern with the client-side approach since foreign code is executed locally. Many client-side programming languages, such as Java, are quite complex making it difficult to ensure security, but this problem is not so significant with logic programming languages whose simpler formal semantics facilitate reasoning about their security (as shown in Chapter 8 for the LogicWeb language).

More than any other logic programming language, there has been substantial work in using Prolog for client-side programming. Section 9.1.1.1 reviews such work and discusses their feasibility for implementing LogicWeb. Logic-based languages designed for querying the Web are discussed in Section 9.1.1.2.

9.1.1.1 Prolog and Client-side Programming

Prolog is being used for client-side programming in the following ways: Prolog libraries which allow Web clients (and servers) to be built, a Prolog system implemented using the *plug-in* mechanism[1] which allows a browser to handle new MIME types, HTML extensions to incorporate Prolog code, and Java integrations

[1] Netscape Plug-In information is available at `http://home.netscape.com/comprod/products/navigator/version_2.0/plugins/index.html`.

with Prolog.

SWI-Prolog 2.1.14 extended with an application support library, HTML parser, and HTTP library has been used in the LogicWeb system. More recently, several libraries supporting Web applications have been implemented for Prolog systems such as PiLLoW/CIAO [48], a SICStus Prolog Objects-based library [79], and the ECLiPSe Prolog HTTP library [32]. These packages also contain tools for parsing the text and extracting information. Future implementations of LogicWeb could be built on these libraries. Cabeza and Hermenegildo's PiLLoW/CIAO library is the most elaborate, and it would be possible to use it to implement a version of LogicWeb. This requires a novel feature of PiLLoW, the active module, which is a form of independent Prolog process, started at the operating system level. In particular, a PiLLoW-based implementation may be a good way to make the LogicWeb system more portable than its current Mosaic-based implementation.

The core components of the LogicWeb system (written in Prolog) are quite independent of Mosaic and can be easily adapted for other implementations. One possible means of porting LogicWeb to the Netscape and Microsoft Internet Explorer browsers is the plug-in mechanism. A plug-in is a code module which is loaded into the browser (while it is running) and used to handle data of a specific MIME type. To execute LogicWeb programs, the plug-in would be an implementation of a Prolog system with libraries to access the Web. A plug-in for SICStus Prolog programs has recently been released.[2] The plug-in can be used to implement an interpreter for the LogicWeb language, but then, the implementors have identified several disadvantages with the plug-in approach: plug-ins can only support light-weight applications, they are platform specific, and a user must download the plug-in and install it before use [76]. Moreover, unlike the current LogicWeb system, when using the Prolog plug-in, LogicWeb programs must be distributed as its own MIME type, distinct from ordinary HTML pages.

[2] See `http://potato.claes.sci.eg/claes/plugin/npsp.html`.

ISO-HTML proposed by Price [170] allows Prolog code to be added to Web pages using new mark-up tags defined using the Standard Generalised Mark-up Language. Price has also developed a system residing on the server for converting ISO-HTML pages into Prolog programs (called *logic documents*). The translation emphasises the detailed representation of the internal structure of a page: every tag and attribute is represented as a fact, whereas the HTML components in LogicWeb programs are at a larger granularity than in logic documents, emphasising logical components (e.g., images, sections, and applets), yet permitting finer details to be extracted if required. A client system for processing logic documents has not been developed. There is no notion of composing programs (or pages) in Price's work.

A number of Prolog implementations in Java has been developed. A small interpreter for a subset of Prolog, called W-Prolog, has been written as a Java applet [210]. Interpreters have also been written as Java applets for two other rule-based languages: ILOG, where rules can be written in C++ [105], and the Java Experts Systems Shell [86], an implementation of the CLIPS expert system shell in Java. Like LogicWeb rules in pages, the program source is downloaded in all these three systems. In contrast, MINERVA [104] is a Prolog-based language implemented as a Java applet. The MINERVA applet downloads and executes compiled Prolog programs (in byte-code form). An advantage of executing compiled code is greater execution efficiency, but an inconvenience is that every MINERVA program has to be compiled before distribution. A compiler for Prolog which generates Java byte-codes is jProlog, which first translates Prolog into Java (the translator is written in Prolog) and then runs the Java to byte-code compiler [70].

Prolog is also interfaced with Java. Amzi! Prolog has a Java class interface to its Prolog system [5], an approach also employed by Calejo and Sousa [49] with their NanoProlog compiler, and in the JIPL system [111].

Another technique is to link Java to a Prolog system through its sockets class [85]. This approach was rejected due to firewall restrictions on non-HTTP

traffic. Linda Interactor allows Prolog-to-Java and Java-to-Prolog bidirectional communication over the net [192]. In [76], a Java applet is used as a front-end to a Prolog-based application running on the server.

There are many advantages to interfacing Prolog to Java: obtaining access to its extensive class libraries, making use of Java's close integration with the browser, and Java's portability (e.g., many browsers are already equipped with a Java interpreter). Recent additions to Java, such as remote method invocation, Sun's JavaSpaces, and the Java Beans programming mode, give Prolog programs access to forms of network programming towards the peer-to-peer model (see Section 9.1.3 below).

An interpreter for the LogicWeb language can be implemented on top of a Prolog system built as a Java applet. One drawback with using such a Java applet is its restrictive security features as imposed by popular browsers (see Chapter 8). This drawback can be overcome by digitally signing the Java applet, which could then be given more privileges. Additional security restrictions can be imposed on LogicWeb programs via the LogicWeb interpreter instead of on the Java applet. Another drawback is the decrease in performance compared to using Prolog systems in C (e.g., SWI-Prolog used in the LogicWeb system), since there are at least two levels of interpretation (given that Java applets are interpreted).

9.1.1.2 Logic-based Web Querying Languages

A number of logic-based languages and paradigms have been developed for querying the Web. Many of these languages utilise an abstract data model of the Web (e.g., as objects, relations, or a graph). Similar to the LogicWeb language, these languages aim to provide higher level abstractions to hide the complex issues of communication and Web data parsing.

WebLog [119], and ADOOD [92] are two examples. Like the LogicWeb language, these languages represent HTTP response objects as first class entities, and utilise deduction for computation. Also, queries over the Web's hypertext

structure can be similarly expressed.

WebLog and ADOOD allow construction (i.e., grouping and formatting) of query results as first class entities. This allows different models of Web pages to be dynamically defined. For instance, an ADOOD class representing a page model with the attributes of title and image can be defined, and language constructs are present to allow instances of this class to be created or modified at run-time. The LogicWeb language does not have such explicit constructs, but a LogicWeb program can be modified by parsing the page contents and extracting components as new facts which are added to the program. For instance, HTML tables can be extracted from interesting sections of a page using the `section/3` facts.

WebLog and ADOOD do not make use of meta-level composition operators and do not deal with pages containing rules.

Another example of a logic-based language for building Web applications is W-ACE [166] which is based on ACE, a parallel logic programming system. The aim of W-ACE is to provide a complete framework for building applications on both the client and server-side. Parallelism in W-ACE allows interactions with multiple servers and permits W-ACE to be used on the server-side to serve multiple Web clients concurrently. W-ACE aims for a close coupling between client and server, where the actions of the client on pages are translated into arguments for predicate calls in the server.

W-ACE has predicates to retrieve pages and represents a page as a tree structure in a logic term. Hypertext structure involving several pages is represented as a graph where each node is a page and each edge, a link. W-ACE is not yet fully developed, but the proposed design suggests the incorporation into W-ACE ideas present in LogicWeb: including W-ACE rules within pages, the modelling of each page as a theory, the use of a basic set of facts for representing a page's HTML text, and extending ACE with meta-level program composition operators (like those described in Chapter 2). The design also speculates on the use of op-

erators from modal logic, where each "world" is a page. These operators express queries that involve reachability relations between worlds or pages. An example is a query which, given a page and a goal, determines if the goal is provable in all the pages (directly) linked from the given page. Such queries can be coded up in the LogicWeb language using context switching.

Datalog is used as a Web querying language in the D^3Web system [2]. They propose a number of new META tag name/value pairs for Web pages to hold extra semantic information. The D^3Web system treats these pages as distributed deductive databases when processing queries. As with LogicWeb, a database is derived from the analysis of a page, but most information comes from its META tag information. Aldana Montes and Yagüe del Valle argue that Datalog is a better choice for querying databases than Prolog due to its underlying set-at-a-time paradigm, its efficiency due to the restrictions placed on the forms of rules and terms, and the incremental nature of their query evaluator. As mentioned in Section 6.6.4, Datalog is a subset of Prolog where all terms in the program are variables or constants. Hence, since only simple data structures are permitted (e.g., no lists), Datalog is less expressive for general purpose programming than Prolog.

Datalog is the basis for the Hy^+ visualisation system, and its associated GraphLog language [99]. The Hy^+ system obtains its information from a Mosaic browser as Prolog facts in a similar way to the LogicWeb system: the URL, title, and any anchors are extracted from each retrieved document. These Hy^+ facts record a history of Web traversals (e.g., links activated), and group the document's URL and title together. GraphLog queries on these facts are specified graphically and are used to generate graphical views of Web traversal history and structure. Different from Hy^+, the LogicWeb system utilises its information from Mosaic to invoke specified behaviours. Also, GraphLog queries are performed over collected Hy^+ facts only, and do not download Web pages.

Another graph-oriented language is WG-log [62], which allows querying over graphs or schemas representing the content of Web sites. WG-log makes a distinction between actual information and conceptual structures. Similar schemas can be encoded in the LogicWeb language and queried, but WG-Log has the advantage of being graphical. WG-log does not yet support queries which compose schemas or take into account links between schemas of different Web sites.

9.1.2 Server-side Systems

LogicWeb applications run on the client-side but communicates with server-side software (e.g., run CGI scripts) through the HTTP protocol (e.g., as shown in Section 6.5).

In general, server-side software is ideal for controlling resources such as databases which cannot be sent over the Web for various reasons. Also, having all users communicate with a central location makes it easier to program applications requiring coordination, such as chat systems or market places.

Server-side solutions may suffer from network load problems, and from excess load on the server, which must handle all user activities.

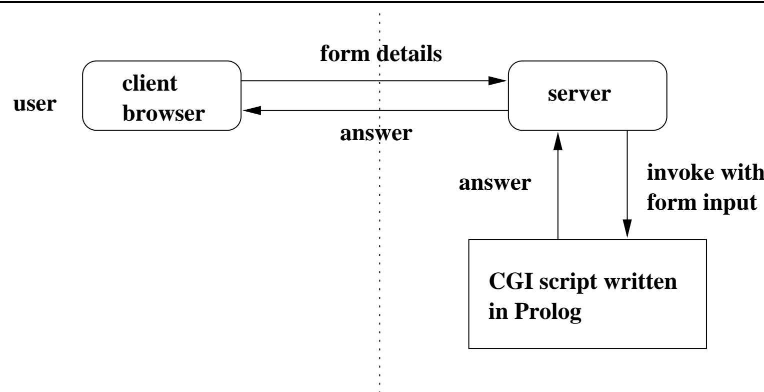
CGI is a popular server-side programming interface, allowing information from Web forms to be passed to programs. There are many libraries which enable Prolog programs to process information from CGI input, and generate suitable replies (typically, new Web pages) [6, 48, 51, 133]. The basic idea is captured by Figure 9.1.

Examples include: WebLS, a tool for building expert systems [6], Bob Carpenter's theorem prover[3], and Lee Naish's ICLP'97 submissions form[4]. An extensive set of CGI Prolog tools were written for the organisation of the program committee work in ILPS'97 [158]. These include facilities for helping the PC chair and submission reviewers.

[3]See `http://macduff.andrew.cmu.edu/cgparser/`.

[4]See `http://www.cs.mu.oz.au/~lee/iclp97/submitreg.html`.

Figure 9.1 Using Prolog CGI scripts.

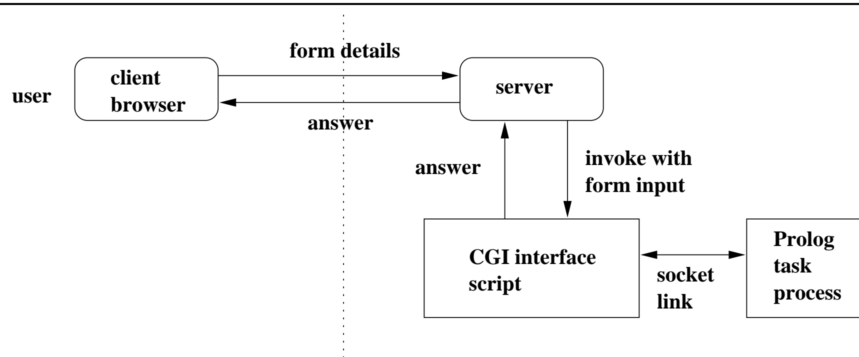


In Figure 9.1, the CGI script is newly invoked for each query from a client, which can be a problem if the script has to load very large support software. The client-server model allows a server to process several clients concurrently, which implies that several invocations of the same script may need to be running simultaneously. This may not be practical because of the size of the system, and also makes changes to shared resources more complicated.

Another server-side solution is to separate query processing into two parts: a light-weight CGI script which acts as an interface to a separate heavy-weight task process. A key feature of the task process is that it is continually running, and so only needs to be loaded once. In the context of logic programming, this process might be a Prolog system or logic database. The invoked CGI interface scripts communicate with the task process by using sockets. The overall approach is shown in Figure 9.2.

Examples of these systems include the Announce system coded in ECLiPSe Prolog which uses this technique to implement its electronic calendar of events [135], the EMRM knowledge-base of medical records which utilises the OR-parallel Aurora system to process multiple queries at once [190], and Phil Vasey's property underwriting system which uses a separate Prolog process built with LPA's ProWeb [201].

Figure 9.2 Separating the interface and task processes.



The PiLLoW/CIAO library supports a higher level communications layer between the interface and task processes based on active modules. Each invocation of the interface script communicates with the task process as if it was calling a module [48]. The authors speculate on using &-Prolog/CIAO to parallelise their Prolog engine.

Although this server-side technique solves the problem of multiple invocations of potentially large task processes, it still leaves unresolved how to support multiple queries on a shared resource. This remains an issue even when parallel languages are used.

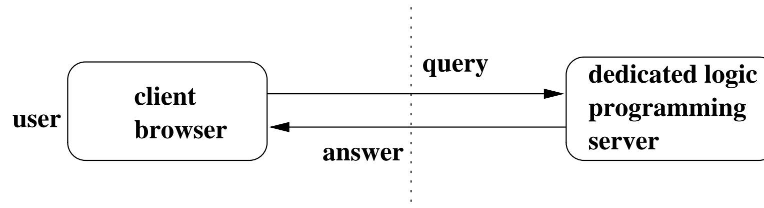
The shared resource problem was highlighted in the CGI scripts developed for ILPS'97 [158]: SICStus Prolog has a database library, but it can only be accessed by one user at a time. One solution is to connect the scripts to a separately running multi-user database; another is to extend the existing database with concurrent transaction features.

Another problem, addressed in the EMRM system, is how to deal with lengthy browser interactions, which require the task process to suspend while the user enters further details. A related difficulty, peculiar to logic programming systems, is how to deal with backtracking to a previous stage in the user interaction. The ProWeb system [133] records the pages associated with earlier stages, and can redisplay them as required. Backtracking may also make it necessary

to rollback changes to (shared) resources. These problems can occur with any multi-user logic programming application, but are compounded by the forms-based user interface supplied by CGI, and the stateless nature of the HTTP protocol. For LogicWeb applications, backtracking leaves side-effects. As mentioned in Chapter 4, LogicWeb programs downloaded in failed derivation branches remain cached. Also, LogicWeb goal evaluations can leave side effects on server hosts on backtracking. For example, if a LogicWeb goal has posted information to a server, on backtracking over this goal, the information is not recovered from the server.

A third server-side technique is to completely replace the traditional Web server by software which combines the functionality of a server with the particular task. This is illustrated in Figure 9.3.

Figure 9.3 A dedicated logic programming server.



A notable logic programming solution in this style is the ECLiPSe HTTP server library, which allows a basic server framework to be customised for different communication protocols [32]. Indeed, the major advantage of this technique is the way that the server can be specialised for specific applications and communication modes. The main drawback is the large amount of work required to implement a fully featured server with concurrency control, error handling, administrative tools, and so on.

This approach is also used in the Munich Rent Advisor, which coded its server with ECLiPSe (but without the help of the ECLiPSe HTTP server library) [87].

A different view of this server-side technique is the idea of a wrapper which

provides a uniform database interface to disparate sources, such as the semi--structured documents found on the Web. Bonnet *et al* [31] describes the use of Prolog to write a generic wrapper.

9.1.3 Peer-to-peer Systems

The Web model is based on clients and servers, which makes it difficult to code systems where the communication is between entities with equal status. In particular, it discourages the implementation of multi-agent systems where it is essential that all the participants can communicate on equal terms.

For this reason, some logic programming systems utilise other abstractions but retain the Internet as their underlying communication layer. Two such abstractions are message-passing and blackboards.

The language April [141] uses message passing and has the ability to move code between machines. April is not strictly speaking a logic programming language, but has borrowed ideas from logic programming, and its macro language can be used to support more Prolog-like behaviour [55].

April is used as the basis of a multi-agent system for extracting information from multiple, heterogeneous information sources [55]. Unlike LogicWeb's CIFI, described in Chapter 5, a variety of agents are used for carrying out component tasks, such as locating sources, phrasing queries in a suitable form for those sources, translating responses, and reformulating queries. Also, agents are moved over the network so that queries are carried out locally at the information sources.

The LogicWeb model focuses on communication between LogicWeb applications and servers, and not on communication between LogicWeb applications residing on different hosts. However, asynchronous message passing between two LogicWeb applications (running on different hosts) can be simulated by using a Web server (acting as a mediator) to store messages for clients. For example, a LogicWeb application can use the POST method to send a message to the

mediator, and another LogicWeb application can request its messages from the mediator using a GET request (at a later time). A limitation is that an application must explicitly invoke a LogicWeb goal to “receive” its messages or to determine if it has any message. The application is not interrupted by, or notified of, message arrival.

The blackboards in Multi-BinProlog are the basis of LogiMOO, a high-level kernel for Internet collaborative work [197]. LogiMOO has been given a forms interface, where CGI scripts are used to communicate with a persistent LogiMOO server. More recent versions of BinProlog permits executing code to migrate between platforms by relocating first order continuations acting as threads [193, 194]. The intuitionistic assumption mechanism is used to invoke computations on remote BinProlog servers. LogiMOO uses local and virtual blackboards to hide the underlying network. Similar approaches are possible in other logic programming languages with Linda-style blackboards, such as SICStus Prolog. For example, ACLT, implemented in SICStus Prolog, offers multiple logical tuple spaces aimed at coordinating activities on the Web [71]. One of their extensions to the Linda model is the notion of communication events which trigger activities automatically.

As early as 1994, a commercial product based on Linda for collaborative work over the Internet, called Ubique Doors [179], used Flat Concurrent Prolog (FCP). However, it is now coded in C++.

The blackboard paradigm, or Linda model, is useful because it operates at a higher level than the networking protocols required for peer-to-peer systems. In particular, the board abstraction can be viewed in several different ways: as a kind of global knowledge-base, a flexible communications medium, or a virtual world where agents meet. Most systems support multiple boards, to further enhance the representation. The board mechanism simplifies the complex task of defining agent cooperation and negotiation protocols since it allows the coupling between agents to be less explicit, overcoming the restrictions of point-to-point

communication in simple message passing and client-server formalisms. Boards allow applications to be scaled up easily since the addition of extra agents does not require a realignment of the communication network. Also, coordination activities can be represented by agents (e.g., as mediators), thereby placing control on an equal footing with the other computational aspects of the paradigm.

The main disadvantage of the board model is the implicit nature of the connections between agents and their control or coordination logic. This can make it difficult to understand the overall behaviour of a system.

Since a CGI forms interface can be used to communicate with a LogiMOO server [197], a LogiMOO client can be implemented as a LogicWeb application. By connecting to the same LogiMOO server, two LogicWeb applications can communicate using the blackboard mechanism.

Different from the above abstractions is the use of a distributed logic programming language. Pamela is a proposed system for helping a user find and track information on the Web [77]. It will be coded using the distributed logic programming language DLP, extended with primitives to access the Web and the DESIRE modelling framework for expressing cooperation between agents. CORBA will be utilised as the object-level mediator for communication between software agents and other Internet entities.

9.2 Other Web Programming Languages

This section reviews SQL-based languages for querying the Web and languages with explicit constructs to match the Web's nondeterministic nature (as explained in Section 2.1.2.3).

9.2.1 SQL-based Web Querying Languages

WebSQL [144] and W3QL [115] are examples of Web querying languages based on the database query language SQL. These languages allow the user to specify queries which

- take into account the Web's hypertext structure;
- incorporate some of the user's knowledge for searching; and
- involve more complex analysis of page contents.

Such queries are not supported by search engines whose pages are queried as a flat corpus. These languages impose a data model on the Web and allow queries to be specified based on the data model. Both these languages adopt a simple relational model of pages which do not represent the contents of pages in detail.

WebSQL models the Web with two relations:

$$Document[url, title, text, type, length, modif]$$

which represents each HTML document's URL, title, HTML source, MIME type, length, and last modification date, and

$$Anchor[base, href, label]$$

which models each anchor (or link) in a document. $base$ is the URL of the HTML document containing the anchor, $href$ is the URL of the document referred to, and $label$ is the label of the anchor.

The data model for W3QL allows the title, base URL, HTML source, and the anchors of HTML documents to be queried.

The LogicWeb model of Web pages subsumes these page models in that a LogicWeb program contains facts allowing all of the above information (and others) to be retrieved. However, W3QL models not only pages but also Latex and PostScript files. Any type of file can be retrieved and processed as strings in

a LogicWeb program. But it would be convenient to query the components of other types of files in the same way as page components are queried. This is listed as an avenue for future work in Chapter 10.

WebSQL and W3QL queries are formulated using *path regular expressions* which specify sequences of Web link traversals. The analysis of page contents and anchor labels involves substring matching in WebSQL and regular expression pattern matching in W3QL. These queries can be reformulated in the LogicWeb language since the full expressive power of Prolog is present. As examples, two WebSQL queries from [11] are re-expressed in the LogicWeb language. These two queries show that link traversals specifiable by path regular expressions are specifiable in the LogicWeb language.

Example 1 (query 5 in [11]). The following query first queries an index server (AltaVista is chosen) to find pages that mention the keywords "employment job opportunities", and then, from each of these pages, paths of length 1, 2, or 3 (i.e., a sequence of 1, 2, or 3 links) are followed to find a page that contains the phrase "software engineer".

```
% URL1 is a page containing ``software engineer'' reachable in a
% path of length 1, 2, or 3 from the AltaVista result page.
se_page(URL1) :-
  lw(get, "http://www.altavista.yellowpages.com.au/cgi-bin/query?
           mss=simple&pg=q&q=employment+job+opportunities")#>
                link(_, URL),
  contains_se(3, URL, URL1).

contains_se(_, URL, URL) :-
  lw(get, URL)#>h_text(Src),
  contains(Src, "software engineer").
contains_se(Depth, URL, URL2) :-
  Depth > 1,
  lw(get, URL)#>link(_, URL1),
  Depth1 is Depth - 1,
  contains_se(Depth1, URL1, URL2).
```

Note that the code does not detect loops formed by pages linking back to previously visited pages.

Example 2 (a generalisation of query 10 in [11]). The following query returns the URL of a broken link which is reachable from a given page.

```
broken_url(URL, BURL) :-
  reachable(URL, BURL),
  not(lw(get, BURL)#>true).       % the link is broken

% URL1 is reachable from URL if
% URL1 is contained in URL or URL1 is reachable from a link in URL.
reachable(URL, URL1) :-
  lw(get, URL)#>link(_, URL1).
reachable(URL, URL2) :-
  lw(get, URL)#>link(_, URL1),
  reachable(URL1, URL2).
```

A broken link is detected by the failure of the goal `lw(get, BURL)#>true`. In both cases, `setof/3` can be used to find all such pages (though infinite looping is possible if a link goes back to a previous page).

The semantics of WebSQL and W3QL assume a static model of the Web, focusing on the structure of the part of the Web being queried such as the node content structure and hypertext structure. The operational semantics in Chapter 3 recognises that the Web is not static and that changes on server hosts and in the network could mean that required programs are not obtainable. The condition on the result of $add_programs$ in rule (3.10) ensures that required programs are available before goal evaluation continues.

WebOQL [10], another Web querying language with SQL-based syntax, employs a tree-structured model of HTML documents similar to that used in W-ACE, but whose representation is not a logic term but expressions made up of tree labels and operators over trees. The tree representation gives better support for analysis and extraction of document components than the relational models. The main aim of the language is to allow restructuring of documents: forming a

new document using selected components extracted from an existing document. In contrast to the tree-structured page models which focus on HTML structure, the LogicWeb page model emphasises types of components such as links, sections, and images, and facilitates extracting and searching over components of the same type.

A drawback of the above query languages is their limited expressiveness compared to traditional programming languages. A full programming language with constructs for Web interaction has greater expressive power and generality.

9.2.2 Languages Modelling the Web's Nondeterministic Nature

The above SQL-based languages assume that the Web is a static database and do not take into account unreliability and nondeterminism in communication. Two recently proposed languages aim to provide constructs to match the nondeterministic nature of the Web: the High-level Internet Programming with Persistent Objects (HIPPO) language [57], and Cardelli *et al*'s service combinators [50].

HIPPO views the Web as a large database mutable by multiple agents (and not just by the agent querying the database). A key feature of HIPPO is the ability to express nondeterminism using the `alt` construct. For example, if `a` and `b` are two statements to retrieve pages from two different sites, then `a alt b` means return the result of either operation. It is an implementation choice to determine if this nondeterminism results in concurrent or alternative (and single-threaded) computations. Such nondeterminism is also expressible in the LogicWeb language, with single-threaded semantics. The HIPPO project also plans to add typed data to the Web and support their querying using HIPPO. The HIPPO language is still under development.

Cardelli *et al*'s service combinators allow the behaviour of a human browsing the Web to be represented as a program consisting of a sequence of commands. There are commands to retrieve pages, concurrently execute two commands, time-out on a page request, terminate a command if the rate of data trans-

fer drops below a threshold, or repeat commands. The service combinators do not in themselves form a full programming language but can be usefully embedded in a language. The service combinators have been implemented as functions and operators in a scripting language called WebL [114]. Timing-out and measuring data transfer rate would be useful in the LogicWeb language. Extending the LogicWeb language to incorporate these features is discussed in the next chapter.

9.3 Logic-based Hypertext Models

The use of first-order logic to capture the components of hypertext have been used by Bieber in [27]. Simple notions of hypertext node, hypertext link, and link traversal are first formalised in first-order logic. For example, the following example adapted from [27] shows how two nodes and a link between them are represented:

node(1, ['The Age', 'August 8, 2001'])

node(2, ['The New York Times', 'August 11, 1999'])

link(1, 1, 2, display, full_window)

This formalisation is then used as a basis for generalising the node and link notions, allowing the generalisations to be captured and reasoned with formally. Nodes are extended from fixed pieces of text to dynamically generated text, and link activation do not simply produce the destination node but can execute attached procedures.

First-order logic is used to model hypertext in [91], with emphasis on formally capturing functionalities such as the filtering of information during linking, and the versioning, aggregation, and generalisation of nodes.

Both the above models aim to provide a clear and rigorous logic-based understanding of hypertext abstractions (e.g., node and link) and a framework to extend their functionality. In a similar way, LogicWeb provides a logic programming interpretation of the Web's hypertext notions of node (or page) and link, and extend their functionality. In LogicWeb, the modelling of link activation using rules provides a basis for extending link semantics (as shown in Chapter 7), and the modelling of pages as programs provides a conceptual understanding of "query-able pages". LogicWeb also introduces the idea of composing Web pages which is not present in the other two hypertext models. In [91], an aggregation groups nodes together under the same name but do not specify semantically rich pairwise combination of nodes (as in the case of LW-composition operators).

Chapter 10

Conclusion

This thesis has presented *LogicWeb*, a model of the Web as a collection of interrelated logic programs, and explored its implementation and applications. The central notion in LogicWeb is the LogicWeb program which is formed from HTTP response objects.

This thesis has demonstrated that LogicWeb is a feasible, versatile, and effective integration of logic programming technology with the Web. LogicWeb enhances Web documents and links with logic programming based interactive behaviours using chiefly client-side computations, and enables the construction of applications on the Web with the help of abstractions from compositional logic programming. A new Web programming paradigm has been developed based on the idea of LogicWeb programs.

Chapter 3 showed how Web documents are translated into LogicWeb programs, how LogicWeb programs form applications on the Web, and how Prolog is extended with the use of LogicWeb programs to form the LogicWeb language. The operational semantics of LogicWeb programs was given by extending the operational semantics of an ordinary compositional logic programming language (described in Chapter 2). Computation involving LogicWeb programs combines deduction with Web interaction, returning a set of downloaded LogicWeb programs as a side-effect of goal derivation. The operational semantics

models this side-effect and shows how LogicWeb programs downloaded at one point in a derivation are used for subsequent goal derivations without further network access. The unpredictable characteristic of the oracle function models the unpredictability in downloading the programs.

Implementation of LogicWeb was explained in Chapter 4. A system integrating a Prolog system with the Mosaic browser was described which permits LogicWeb programs to be queried through the user interface mechanisms of forms and links. For processing queries, an interpreter for LogicWeb programs was developed based on the operational semantics of Chapter 3. Alternative methods for implementing LogicWeb were outlined in Chapter 9.

The LogicWeb applications investigated in Chapters 5, 6, and 7 demonstrate a variety of Web programming tasks that logic programming is very useful for. These tasks naturally involve symbolic processing (e.g., knowledge modelling and manipulation). Particular applications developed included the rule-based agent CIFI, a querying facility for citation databases on the Web, guided tours on the Web, and links which on activation query structured data to determine their destinations and perform computations to cope with HTTP request failures.

Logic programming can contribute towards Web search tool construction. As CIFI exemplifies, when the knowledge required for a Web searching task is known in detail and can be precisely stated (e.g., the search target can be recognised using heuristics, and it is known which sites to use, and what to look for on pages), search can be automated by engineering the knowledge into a tool, and logic programming is a convenient formalism for such knowledge engineering. Moreover, as logic programming is widely used for declaratively coding graph search algorithms, the LogicWeb language would be useful for declaratively coding Web search algorithms, with minimal effort required for details of networking, document caching, and document parsing.

The data modelling and high-level symbolic processing capabilities of logic programming are crucial for building lightweight Web databases which are read-

able yet formal and amenable to sophisticated querying and composition. These capabilities and dynamic database manipulation are also significant for addressing Web linking problems through querying and manipulating structured metadata.

In Chapter 8, security issues concerned with executing downloaded LogicWeb programs were dealt with, and the system was extended to implement a policy-based security model for the safe execution of LogicWeb programs. The use of policy programs was specified by extending the operational semantics from Chapter 3. The modified operational semantics was used to provide a simple proof of the soundness of the model with respect to a safety property.

From the viewpoint of mobile code security, logic programming has a number of advantages, as the security model in Chapter 8 exemplifies. First, logic programs are amenable to formal reasoning and their formal semantics can be extended to accommodate security features forming the basis of a security model. Second, extensive control over the execution of logic programs can be achieved using meta-interpreters. Third, the absence of pointers and automatic memory management prevent misuse of memory. Fourth, the use of logic programs as policy programs in Chapter 8 enables the declarative specification of resource usage and control.

Chapter 9 placed LogicWeb in the context of other work on logic programming and the Web, Web programming languages, and logic-based hypertext models.

Amidst the large established base of imperative code and expertise, logic programming languages continue to improve in a variety of ways such as in expressive power (e.g., *Lygon* [211]), efficiency in execution (e.g., *Mercury* [181]), and program structuring abstractions (e.g., [101]). Logic programming languages (and other declarative programming languages) should be exploited for Web programming, and this thesis has proposed the LogicWeb approach towards this end.

10.1 Language Extensions

A goal of language design should be simplicity [206]. The current design of the LogicWeb language focuses on providing the ability to manipulate LogicWeb programs from disparate sources. Mastery of the current language features will already enable a variety of applications to be built such as those described in this thesis. Nevertheless, the language can be extended in a variety of directions increasing its functionality and generality, though at the cost of greater complexity. Seven features which can be added are outlined below. A design and implementation challenge is to have all these features co-exist in the language, which potentially will have a complicated semantics.

10.1.1 Extending the Semantics of Downloading

In the current implementation of the LogicWeb language, there are no mechanisms for handling network delays, low data transfer rates, or requests via proxy servers. Such mechanisms would allow greater programmer control over interactions with the Web, similar to Cardelli's service combinators (see Section 9.2.2). For example, the programmer should be able to limit the amount of time spent on a HTTP request, and a request may need to be routed through a proxy server.

One possible way to implement these mechanisms is to extend the LogicWeb program identifier to the form:

```
lw(Type, URL, Control, Status)#>Goal
```

`Control` is a list of terms specifying control information such as a time-out period on the HTTP request, a proxy Web server through which the request must be channelled, or a threshold data transfer rate. If the data transfer rate drops below this threshold value, then the download is interrupted. `Status` is a term reporting a successful download or a list of terms indicating reasons for a download failure. A crucial change here from the earlier semantics of LogicWeb goals is that download failure will not fail the LogicWeb goal, i.e. the LogicWeb goal

will succeed leaving variables in `Goal` unbound. However, the earlier semantics can be coded up by taking advantage of unification, instantiating `Status` with `download_success` in the LogicWeb goal:

```
lw(Type, URL, Control, download_success)#>Goal
```

The control information is required for downloading a program. As an example, the following goal time-outs if there is no response from the server after 5 seconds, and issues the HTTP request through a proxy at location `URL1`:

```
?- lw(get, "URL0", [timeout(5), proxy("URL1")], Status)#>goal.
```

10.1.2 Other Header Fields in HTTP Requests

The LogicWeb program identifier used in this thesis is formed from the minimum amount of information required in a HTTP request. This information is adequate for the applications presented and keeps the identifiers simple.

The HTTP protocol allows clients to submit other header fields in a HTTP request such as passwords and usernames (often used by Web servers for restricting access to information), and MIME types that the client will accept. Similar to the control information mentioned above, such information can be included in the program identifier when downloading a page. For example, the following goal extends the goal in Section 10.1.1 by submitting the username `guest` with password `general`:

```
?- lw(get, "URL0",
      [timeout(5), proxy("URL1"), pass("guest", "general")],
      Status)#>goal.
```

This extension leads to lengthy program identifiers. Subsequent referrals to a downloaded program do not require the additional header information, but the programmer has to remember that a program has been downloaded.

10.1.3 Lazy Download

The semantics in Chapter 3 can be said to download programs *eagerly*. Context switching as specified in Chapter 3 downloads all programs in a LW-composition before goal evaluation begins, including programs that are never used in the evaluation of the specified goal. One optimisation is to download programs *lazily*, i.e. a program is downloaded only when an attempt is made to retrieve its clauses.

For example, in the following LogicWeb goal

```
?- (lw(get, "URL0") + lw(get, "URL1"))#>p(X).
```

suppose that `p(X)` succeeds binding `X` using only the clauses from `lw(get, "URL0")`, and no further solutions for `X` are required. Then, with eager downloading, time and resources are wasted in downloading `lw(get, "URL1")`, whose clauses are not used. Since the evaluation of `p(X)` uses the clauses from `lw(get, "URL0")` first, this wastage is prevented by first downloading `lw(get, "URL0")` (as soon as its clauses are required), and then evaluating `p(X)` without needing to download `lw(get, "URL1")`.

Using the semantics given in Chapter 3, the above LogicWeb goal expresses the programmer's intention to "download the two programs, and evaluate `p(X)` in the LW-union of the two programs", and if one of the programs cannot be downloaded, this intention cannot be satisfied.

Lazy download offers an alternative semantics for the LogicWeb goal. The LogicWeb goal now means "downloading programs only when their clauses are needed, evaluate `p(X)` returning a solution for `X` as soon as one can be obtained", i.e. a solution can be returned using only programs in a sub-expression of the original program expression.

Goals that use lazy download are more robust returning as many solutions as possible and can prevent downloading programs whose clauses are never used. In the above example, if downloading `lw(get, "URL1")` fails, then eager down-

load returns no solution. Goals with lazy download semantics must be distinguished from goals using eager download semantics (e.g., by using a different operator).

Lazy download semantics might be implemented as follows. Program 4.2 is modified so that `download/2` is called when clauses are selected and not by `establish_context/3`. `::/2` is replaced by `select_clause/2` in the fourth clause of `demo/2` in Program 4.2:

```
demo(E, A) :- select_clause(E, (A :- B)), demo(E, B).
```

`select_clause/2` defines how clauses are selected from a program expression in a similar way as `select_clause/3` in Program 8.1 (without the list of policy program identifiers). But for selecting a clause from a single program, the following rule is defined:

```
select_clause(lw(T, U), (A :- B)) :-
  download(T, U),       % attempt a download
  lw(T, U)::(A :- B).
```

This means that no attempt is made to download a program until a clause is required from it, i.e. when `select_clause/2` is called with the program's identifier as its first argument.

10.1.4 Concurrency

The current LogicWeb implementation is single-threaded. The exploitation of concurrency can potentially improve the performance of applications in a number of ways. For example, the programs required in a LW-composition can be downloaded concurrently. Furthermore, while a program is downloading, other goals can be evaluated. In the following query, the conjuncts can be evaluated concurrently:

```
?- lw(get, "URL0")#>goal0, lw(get, "URL1")#>goal1, goal2.
```

The programs are downloaded at the same time and the goals are evaluated independently and concurrently. LogicWeb extensions to a concurrent logic programming language such as Parlog [54] can be investigated. The addition of meta-level operators for composing programs to the concurrent language W-ACE has already been proposed (as noted in Section 9.1.1.2).

10.1.5 Operations On the Program Store

The operational semantics in Chapter 3 captures the monotonic extension of the program store through successfully downloaded programs. However, deletions (and replacement) of programs is not represented. One promising formalism for representing such updates is linear logic which, roughly speaking, contains connectives to delete (consume) and add (produce) clauses (resources). In contrast to `assert/1` and `retract/1` in Prolog which are not part of classical logic, operations to update resources are part of linear logic. Linear logic has been used to provide a logical interpretation of database updates [121]. Based on such work, linear logic based operators can be investigated for updating the program store. Other formalisms for state update in logic programming (e.g., [3]) may also be explored.

10.1.6 Application-specific LW-composition Operators

Other LW-composition operators can be explored besides those utilised in the LogicWeb language. Adding more operators increases expressive power but also adds complexity to the language. Dialects of the LogicWeb language can be created which are tailored for particular applications, or the language can be compartmentalised. For example, LogicWeb counterparts of union (e.g., LW-union), overriding union, and tuple inheritance would form a suite of operators modelling (three different notions of) inheritance between LogicWeb programs. As mentioned in Chapter 2, overriding union has been used to model inheritance in

object-oriented programming, and tuple inheritance expresses a form of inheritance between two programs which cannot be captured using union and overriding union (as shown in [152]).

10.1.7 Multiple Page Models

The page model employed in this thesis allows the convenient retrieval of particular HTML components from pages. This page model is satisfactory for many applications, but it is desirable to have other models of the page at hand. For example, the current page model does not explicitly capture context information such as the section to which a given link belongs, or the paragraph containing a reference to an image. But the representation of a page as a single term in [48], the detailed representation of HTML tags and attributes in [170], and the tree-structured representations in W-ACE (mentioned in Section 9.1.1.2) and Web-OQL (mentioned in Section 9.2.1) make some context information explicit.

One way to allow more flexible querying of pages is to utilise multiple page models for each page, each model optimised for a particular kind of query. Another method is to allow a LogicWeb program a means of specifying its own page model, such as the tags to parse and the facts to generate. For example, a tag such as

<PAGE_MODEL= *URL*>

can be used where *URL* refers to a program which can be loaded into the system for parsing the page. A basic page model will still be employed but additional components can be generated via this specification.

The Web contains a large variety of document types (e.g., PostScript, Latex, Microsoft Word, Virtual Reality Modelling Language, music mark-up languages). Models are needed for representing and querying other types of documents besides HTML.

10.2 Using Standardised Mark-up

The recent HTML 4.0 specification [171] recommends tags and attributes that allow HTML documents to include scripts in highly active and interactive ways. For instance, it is possible to include in a document

- a script that will execute whenever the document is downloaded modifying the document's contents dynamically,
- a script to process form inputs, and
- a script that is associated with a link and will execute when the link is activated.

The scripting language must be specified together with the embedded script. This mark-up provides a standard way in which LogicWeb rules can be included within documents, i.e. non-standard tags such as "`<LW_CODE>...</LW_CODE>`" can be omitted. Pages with rules embedded using these HTML 4.0 tags can be mapped to LogicWeb programs in a way similar to the mapping of HTTP response objects to facts described in Chapter 3.

A Web browser could be built which will not only support JavaScript, VB-Script, or Tcl scripts, but also LogicWeb rules. The browser should allow its user to build link behaviours into it, as is possible for the LogicWeb system, where the user can modify the predicate `sys_link_action/2` which is invoked by the system in response to link activations (see Section 7.6).

10.3 Applications

Section 6.6.2 briefly outlined the use of lightweight deductive databases in an Intranet setting. Other applications should be explored.

One LogicWeb application that looks promising is distributed software engineering, where timely communication between project members on remote sites

is crucial. As noted in [14], the Web can support the required communication. Sedlock *et al* [177] recommends writing Prolog programs in the literate programming style, together with their documentation and specifications, in HTML files. The programs are extracted when needed. This not only benefits the programmer, but also other team members who can then browse the code. The paper describes an implementation in Prolog of a software project management system where all the code is written in HTML files. However, all the source code resides on a single server.

Sedlock *et al*'s recommendation could be taken further with LogicWeb as the basis for a distributed programming platform. A scenario can be envisioned where programmers on remote sites cooperate in building software consisting of distributed LogicWeb programs, each of which is readily browsable and executable from other LogicWeb programs. More sophisticated module mechanisms can already be built from LogicWeb programs. For example, the operators for combining LogicWeb programs can be used to model several notions of encapsulation and visibility in the way done in [39]. Linking to versions of programs can be supported as shown in Chapter 7.

Another use of LogicWeb would be to bring more traditional rule-based applications to the Web such as expert systems. There are already expert systems with a Web interface and use mainly server-side evaluation. For example, users interact with the WebLS system via CGI scripts [178]. In the architecture proposed in [76], a Java-applet is used as the front-end to a Prolog expert system running on the server-side. LogicWeb offers a different architecture where the rule-bases, the reasoning component, and the user interface can be implemented as separate LogicWeb programs, and where all processing is client-based. This reduces the load on the server host, stores state information on the client host relieving the server of the need to maintain state information for multiple clients, provides each user with a copy of the expert system (allowing local alterations and extensions), and encourages software engineering principles.

Bibliography

[1] K. Akama. Inheritance Hierarchy Mechanism in Prolog. In *Proceedings of the 5th Conference on Logic Programming (Lecture Notes in Computer Science No. 264)*, pages 12 – 21. Springer-Verlag, June 1986.

[2] J.F. Aldana Montes and M.I. Yagüe del Valle. Querying the Web with Higher Expressive Power: D^3Web. 1997. Available at <http://apolo.lcc.uma.es/personal/yague/seg/iclp97.html>.

[3] V. Alexiev. Mutable Object State for Object-Oriented Logic Programming. Technical Report TR 93-15, Department of Computing Science, University of Alberta, August 1993. Available at <ftp://ftp.cs.ualberta.ca/pub/TechReports/1993/TR93-15/>.

[4] Amzi! Prolog. HyperBase. July 1992. Product out of distribution. Information available by e-mail to <info@amzi.com>.

[5] Amzi! Prolog. Amzi! Prolog + Logic Server – Embed Logic-Bases in C/C++, Java, Visual Basic, Delphi, the Web and More. 1997. Available at <http://www.amzi.com/catprls.htm>.

[6] Amzi! Prolog. WebLS - Embed Intelligent Components on Web Pages. 1997. Documentation and system available at <http://www.amzi.com/catwebls.htm>.

[7] D. Aquilino, P. Asirelli, C. Renso, and F. Turini. Applying Restriction Constraints to Deductive Databases. *"Non-determinism in Deductive Databases", Annals of Mathematics and Artificial Intelligence*, 19(1 & 2):3 – 25, 1997. Available at <file://repl.iei.pi.cnr.it/pub/asirelli/AMAI-97.ps>.

[8] D. Aquilino, C. Renso, and F. Turini. Towards Declarative GIS Analysis. In *Proceedings of the 4th ACM-GIS Workshop*, Rockville, Maryland, U.S.A., November 1996. Available at <http://www.di.unipi.it/~renso/papers/gis-analysis.ps>.

[9] R. Armstrong, D. Freitag, T. Joachims, and T. Mitchell. WebWatcher: A Learning Apprentice for the World Wide Web. In *On-line Working Notes of the AAAI Spring Symposium Series on Information Gathering from Distributed, Heterogeneous Environments*, 1995. Available at <http://www.isi.edu/sims/knoblock/sss95/mitchell.ps>.

[10] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, U.S.A., February 1998. Available at <ftp://ftp.db.toronto.edu/pub/papers/weboql.ps.gz>.

[11] G. Arocena, A. Mendelzon, and G.A. Mihaila. Applications of a Web Query Language. In *Proceedings of the 6th International World Wide Web Conference*, Santa Clara, U.S.A., April 1997. Available at <http://proceedings.www6conf.org/HyperNews/get/PAPER267.html>.

[12] P. Asirelli, C. Renso, and F. Turini. Language Extensions for Semantic Integration of Deductive Databases. In *Proceedings of the International Workshop on Logic In Databases 1996 (Lecture Notes in*

Computer Science No. 1154). Springer-Verlag, 1996. Available at <file://rep1.iei.pi.cnr.it/pub/asirelli/LID-96.ps>.

[13] H. Bacha. MetaProlog Design and Implementation. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1371 – 1387, Seattle, Washington, August 1988. MIT Press.

[14] M. Baentsch, G. Molter, and P. Sturm. WebMake: Integrating Distributed Software Development in a Structure-enhanced Web. *Proceedings of the 3rd International World Wide Web Conference, Computer Networks and ISDN Systems*, 27(6), April 1995. Available at <http://www.igd.fhg.de/www/www95/proceedings/papers/51/WebMake/WebMake.html>.

[15] M. Balabanovic and Y. Shoham. Learning Information Retrieval Agents: Experiments with Automated Web Browsing. In *On-line Working Notes of the AAAI Spring Symposium Series on Information Gathering from Distributed, Heterogeneous Environments*, 1995. Available at <http://www.isi.edu/sims/knoblock/sss95/balabanovic.ps>.

[16] M. Baldoni, L. Giordano, and A. Martelli. A Multimodal Logic to Define Modules in Logic Programming. In D. Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 473 – 487. MIT Press, 1993.

[17] S. Ball. SurfIt! - A WWW Browser. In *Proceedings of the 4th Annual USENIX Tcl/Tk Workshop*, pages 161 – 171, Monterey, California, U.S.A., July 1996. Available at <http://demos.anu.edu.au/steve/papers/Tcl-Workshop-96/ball.html>.

[18] J.A. Bank. Java Security. December 1995. Available at <http://www-swiss.ai.mit.edu/~jbank/javapaper/javapaper.html>.

[19] C. Barcaroli, L. Iocchi, M. Lenzerini, and D. Nardi. Knowledge-based Access to the Network. In *Proceedings of the Workshop on "Artificial Intelligence-based Tools to Help WWW Users" at the 5th International World Wide Web Conference*, May 1996. Available at <http://www.info.unicaen.fr/~serge/3wia/workshop/papers/paper6.html>.

[20] J. Barklund. Metaprogramming in Logic. In A. Kent and J.G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 33, pages 205 – 227. Marcel Dekker, New York, 1995. Also available as UPMAIL Technical Report No. 80 at <ftp://ftp.csd.uu.se/pub/papers/reports/0080.ps.gz>.

[21] T. Berners-Lee. Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World Wide Web. May 1994. Available at <http://www.w3.org/Addressing/URL/URI_Overview.html>.

[22] T. Berners-lee. Generic Resources. 1996. Available at <http://www.w3.org/DesignIssues/Generic.html>.

[23] T. Berners-Lee. World-wide Computer. *Communications of the ACM*, 40(2):57 – 58, February 1997.

[24] T. Berners-Lee, R. Cailliau, A. Luotonen, H.F. Nielsen, and A. Secret. The World Wide Web. *Communications of the ACM*, 37(8):76 – 82, August 1994.

[25] T. Berners-Lee, R. Fielding, and H. Frystyk. HyperText Transfer Protocol version 1.0 Specification (RFC 1945). Available from

<http://www.w3.org/pub/WWW/Protocols/Specs.html> and at <http://ds.internic.net/rfc/rfc1945.txt>.

[26] P. Beynon-Davies, D. Tudhope, C. Taylor, and C. Jones. A Semantic Database Approach to Knowledge-based Hypermedia Systems. *Information and Software Technology*, 36(6):323 – 329, 1994.

[27] M.P. Bieber and S.O. Kimbrough. On the Logic of Generalised Hypertext. *Decision Support Systems*, 11:241 – 257, 1994.

[28] H. Bloomfield. *Links in Hypertext: An Investigation into How They Can Provide Information on Inter-node Relationships*. PhD thesis, Queen Mary and Westfield College, University of London, 1994.

[29] R. Bol. *Loop Checking in Logic Programming*. PhD thesis, CWI, Amsterdam, The Netherlands, October 1991.

[30] H. Boley. Knowledge Bases in the World Wide Web: A Challenge for Logic Programming. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 139 – 147, Bonn, Germany, September 1996. Available at <http://clement.info.umoncton.ca/~lpnet/lp-internet/lpwww-e/lpwww-e.html>.

[31] Ph. Bonnet and S. Bressan. Extraction and Integration of Data from Semi-structured Documents into Business Applications. In *Proceedings of the 10th Symposium and Exhibition on Industrial Applications of Prolog*, Kobe, Japan, October 1997. Available at <http://sirac.inrialpes.fr/~pbonnet/inap97-f.ps.gz>.

[32] Ph. Bonnet, S. Bressan, L. Leth, and B. Thomsen. Towards ECLiPSe Agents on the Internet. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 1 – 9, Bonn, Germany, September 1996. Available at <`http://clement.info.umoncton.ca/~lpnet/lp-internet/eclipse/ea.html`>.

[33] K. A. Bowen. Meta-Level Programming and Knowledge Representation. *New Generation Computing*, 3(4):359 – 383, 1985.

[34] K. A. Bowen and R. A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153 – 172. Academic Press, 1982.

[35] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, Universita di Pisa-Genova-Udine, 1993.

[36] A. Brogi, E. Lamma, and P. Mello. A General Framework for Structuring Logic Programs. Technical report, Dipartimento di Matematica e Informatica, Universita'di Udine. C.N.R., TR 04-90-RR, Maggio 1990.

[37] A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In *Proceedings of the 1st and 2nd Russian Conference on Logic Programming Framework (Lecture Notes in Artificial Intelligence No. 592)*, pages 102 – 113. Springer-Verlag, 1990/1991.

[38] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for Modularising Logic Programming. In *Proceedings of the 3rd. International Workshop, Meta-Programming in Logic, META 92 (Lecture Notes in Computer Science No. 649)*, pages 105 – 119. Springer-Verlag, June 1992.

[39] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361 – 1398, 1994.

[40] A. Brogi, C. Renso, and F. Turini. Amalgamating Language and Meta-Language for Composing Logic Programs. In *Proceedings of GULP-PRODE 94 Joint Conference on Declarative Programming*, pages 580 – 592, Peniscola, Spain, 1994. Available at <`ftp://ftp.di.unipi.it/papers/turini/PRODE94.ps.gz`>.

[41] A. Brogi, C. Renso, and F. Turini. Dynamic Composition of Parameterised Logic Programs. In *Proceedings of the Workshop on Logic-based Composition of Software (in conjunction with the 14th International Conference on Logic Programming)*, July 1997. Available at <`http://www.scs.leeds.ac.uk/hill/locos/final/brogi.ps.gz`>.

[42] A. Brogi and F. Turini. Meta-logic for Program Composition: Semantic Issues. In K.R. Apt and F.Turini, editors, *Meta-Logics and Logic Programming*, chapter 4, pages 83 – 110. MIT Press, 1995.

[43] L. Brown. Mobile Code Security. Technical report, Australian Defence Force Academy, Canberra, Australia. TR - CS07/96, September 1996. Presented to AUUG96, Melbourne, Australia and available at <`http://www.adfa.oz.au/~lpb/TR/mcode96.html`>.

[44] L. Brown. Custom Security Policies in SSErl (DRAFT). April 1997. Available at <`http://www.adfa.oz.au/~lpb/TR/ssp97/sserl_pol97.html`>.

[45] L. Brown. Introducing SERCs Safer Erlang (DRAFT). April 1997. Available at <`http://www.adfa.oz.au/~lpb/TR/ssp97/sserl97a.html`>.

[46] M. Bugliesi. A Declarative View of Inheritance in Logic Programming. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 113 – 127. MIT Press, 1992.

[47] M. Bugliesi, E. Lamma, and P. Mello. Modularity In Logic Programming. *Journal of Logic Programming*, 19 & 20:443 – 502, May 1994.

[48] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for Internet/WWW Programming Using Computational Logic Systems. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 43 – 62, Bonn, Germany, September 1996. Available from <http://clement.info.umoncton.ca/~lpnet/lp-internet/pillow/lpnet3.html>.

[49] M. Calejo and J.P. Sousa. Embedding Prolog in the Java Environment. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 108 – 116, July 1997. Available at <http://www.clip.dia.fi.upm.es/lpnet/proceedings97/calego/calego.html>.

[50] L. Cardelli and R. Davies. Service Combinators for Web Computing. 1997. Available at <http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers/ServiceCombinators.A4.ps>.

[51] B. Carpenter. A Prolog Based CGI Handler. 1996. Available at <http://macduff.andrew.cmu.edu/cgparser/prolog_cgi.html>.

[52] T. Chikayama. ESP - Extended Self-Contained Prolog - as a Preliminary Kernel Language of 5th Generation Computers. *New Generation Computing*, 1(1):11 – 24, 1983.

[53] P. Ciancarini and G. Levi. What is Logic Programming Good for in Software Engineering. Technical Report UBLCS-93-9, Laboratory for Computer Science, University of Bologna, April 1993.

[54] K.L. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1 – 49, January 1986.

[55] K.L. Clark and V.S. Lazarou. Distributed Information Retrieval Using a Multi-Agent System and the Role of Logic Programming. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 82 – 99, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/lazarou.ps>.

[56] D. Connolly. W3C: On Mobile Code. December 1996. Available at <http://www.w3.org/MobileCode/>.

[57] R. Connor, K. Sibson, and P. Manghi. HIPPO: High-level Internet Programming with Persistent Objects. Available at <http://www.dcs.gla.ac.uk/~hippo/>.

[58] P. Coscia, P. Franceschi, G. Levi, G. Sardu, and L. Torre. Meta-Level Definition and Compilation of Inference Engines in the Epsilon Logic Programming Environment. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 359 – 373. MIT Press, 1988.

[59] M. Covington. *Natural Language Processing For Prolog Programmers*. Prentice-Hall, Inc., 1994.

[60] M.L. Creech. Author-Oriented Link Management. In *Proceedings of the 5th International World Wide Web Conference*, April 1997. Available at <http://www5conf.inria.fr/fich_html/papers/P11/Overview.html>.

[61] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. Analyzing Mobile Code Languages. In *Mobile Object Systems: Towards the Programmable Internet (Lecture Notes in Computer Science No. 1222)*, pages 93–110. Springer-Verlag, April 1997.

[62] E. Damiani and L. Tanca. Structuring and Querying the Web Through Graph-Oriented Languages. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with 14th International Conference on Logic Programming)*, pages 129 – 148, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/edamiani.ps>.

[63] J. Davila (Web page author), editor. *On-line Abstracts for "Logic Programming and the Internet, Opportunities and Challenges, A Compulog Net Workshop at Imperial College"*, London, U.K., December 1996. Available at <http://www-lp.doc.ic.ac.uk/lp-internet/lp-int-abs.html>.

[64] A. Davison. A Survey of Logic Programming-based Object Oriented Languages. In P. Wegner, A. Yonezawa, and G. Agha, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 42 – 106. MIT Press, 1993.

[65] A. Davison and S.W. Loke. An Introduction to LogicWeb. In *Proceedings of the 20th Electrical Engineering Conference of Thailand*, pages 262 – 267 (Volume 1), Bangkok, Thailand, November 1997. Technology Media Company Ltd.

[66] K. De Bosschere, M. Hermenegildo, and P. Tarau, editors. *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, Leuven, Belgium, September 1997. Available at <`http://clement.info.umoncton.ca/~lpnet/proceedings97/`>.

[67] P.M.E. De Bra and R.D.J. Post. Information Retrieval in the World-Wide Web: Making Client-based Searching Feasible. In *Proceedings of the 1st World Wide Web Conference, Computer Networks and ISDN Systems*, pages 183 – 192. Elsevier Science, 1994.

[68] P.M.E. De Bra and R.D.J. Post. Searching for Arbitrary Information in the WWW: the Fish-Search for Mosaic. In *Proceedings of the 2nd International World Wide Web Conference*, 1994. Available at <`http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/debra/article.html`>.

[69] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190

– 200, Oakland, California, U.S.A., May 1996. Available at <http://www.cs.princeton.edu/sip/pub/secure96.html>.

[70] B. Demoen and P. Tarau. jProlog. 1997. System available at <http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/>.

[71] E. Denti, A. Natali, and A. Omicini. Merging Logic Programming into Web-based Technology: A Coordination-based Approach. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 117 – 128, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/denti.ps>.

[72] E. Denti and A. Omicini. Open Logic Theory Composition in a Software Engineering Perspective. In L. Sterling and P. Ciancarini, editors, *Proceedings of the Post-Conference Workshop on Applications of Logic Programming to Software Engineering (in conjunction with the International Conference on Logic Programming)*, pages 139 – 148, Genova, Italy, June 1994.

[73] S. DeRose. Expanding the Notion of Links. In *Proceedings of Hypertext '89*, pages 249 – 257. ACM Press, 1989.

[74] S. Dobson and V. Burrill. Towards Improving Automation in the World Wide Web. Available at <http://www.scit.wlv.ac.uk/ndisd/dobson.ps>.

[75] S.A. Dobson and V.A. Burrill. Lightweight Databases. *Proceedings of the 3rd International World Wide Web Conference, Computer Networks and ISDN Systems*, 27(6), April 1995. Available at <http://www.igd.fhg.de/www/www95/proceedings/papers/54/darm.html>.

[76] S.R. El-Beltagy, M. Rafea, and A. Rafea. Practical Development of Internet Prolog Applications using a Java Front End. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 100 – 107, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/beltagy/beltagy.html>.

[77] A. Eliëns, P. de Bra, J. Treur, F. Brazier, and H. van Vliet. Position paper: Web Agent Support Programming. In *Proceedings of the Workshop on Logic Programming and the Web at the 6th International World Wide Web Conference*, Santa Clara, U.S.A., 1997. Available at <http://www.cs.vu.nl/~eliens/WWW6/papers/wasp/>.

[78] A. Eliëns (Web page author), editor. *Proceedings of the Workshop on Logic Programming and the Web at the 6th International World Wide Web Conference*, Santa Clara, U.S.A., April 1997. Available at <http://www.cs.vu.nl/~eliens/WWW6/index.html>.

[79] J. Eriksson, F. Espinoza, N. Finne, F. Holmgren, S. Janson, N. Kaltea, and O. Olsson. An Internet Software Platform Based on SICStus Prolog. In *Proceedings of the Workshop on Logic Programming and the Web at the 6th World Wide Web Conference*, Santa Clara, U.S.A., April 1997. Available at <http://www.cs.vu.nl/~eliens/WWW6/papers/joakime/>.

[80] W. Ernst. *Presenting ActiveX*. Sams.net, 1996.

[81] O. Etzioni and D. Weld. Intelligent Agents on the Internet: Fact, Fiction, and Forecast. *IEEE Expert*, 10(4):44 – 49, August 1995.

[82] J. Euzenat. Knowledge Bases as Web Page Backbones. In *Proceedings of the Workshop on "Artificial Intelligence-based tools to help WWW users" at*

the 5th International World Wide Web Conference, May 1996. Available at <http://www.info.unicaen.fr/~serge/3wia/workshop/papers/paper10.html>.

[83] A. Fall. *Reasoning with Taxonomies*. PhD thesis, School of Computing Science, Simon Fraser University, 1996.

[84] A. Farquhar, A. Dappert, R. Fikes, and W. Pratt. Integrating Information Sources Using Context Logic. In *On-line Working Notes of the AAAI Spring Symposium Series on Information Gathering from Distributed, Heterogeneous Environments*, January 1995. Available at <http://www.isi.edu/sims/knoblock/sss95/farquhar.ps>.

[85] D. Ferguson. Linking a Prolog Program into an HTTPD. 7 November 1996. Posting to <comp.lang.prolog>.

[86] E.J. Friedman-Hill. JESS, The Java Expert System Shell (Version 3.2). November 1997. Available at <http://herzberg.ca.sandia.gov/jess/README.html>.

[87] T. Frühwirth and S. Abdennadher. The Munich Rent Advisor. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 11 – 27, Bonn, Germany, September 1996. Available at <http://clement.info.umoncton.ca/~lpnet/lp-internet/lpnet5/lpnet5.html>.

[88] D.M. Gabbay and U. Reyle. N-Prolog: An Extension of Prolog with Hypothetical Implications. *Journal of Logic Programming*, 1:179 – 210, 1984.

[89] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[90] S. Garfinkel. *PGP: Pretty Good Privacy*. O' Reilly and Associates, Inc., 1995.

[91] P.K. Garg. Abstraction Mechanisms in Hypertext. *Communications of the ACM*, 31(7), July 1988.

[92] F. Giannotti, G. Manco, and D. Pedreschi. A Deductive Data Model for Representing and Querying Semistructured Data. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 55–67, July 1997. Available at <`http://clement.info.umoncton.ca/~lpnet/proceedings97/manco.ps`>.

[93] L. Giordano and A. Martelli. Structuring Logic Programs: A Modal Approach. *Journal of Logic Programming*, 21(2):59 – 94, 1994.

[94] L. Giordano, A. Martelli, and G. Rossi. Structured Prolog: A Language for Structured Logic Programming. In *Software - Concepts and Tools*, number 15, pages 125 – 145. Springer-Verlag, 1994.

[95] R.S. Gray. Agent Tcl: A Flexible and Secure Mobile-agent System. In M. Diekhans and M. Roseman, editors, *Proceedings of the 4th Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, U.S.A., July 1996. Available at <`http://www.cs.dartmouth.edu/~agent/papers/tcl96.ps.Z`>.

[96] K. Hammond, R. Burke, and C. Martin. A Case-Based Approach to Knowledge Navigation. In *Proceedings of the AAAI Workshop on Indexing and Reuse in Multimedia Systems*, pages 46 – 57. AAAI, 1994.

[97] Y. Han, S.W. Loke, and L. Sterling. Agents for Citation Finding on the World Wide Web. In *Proceedings of the 2nd International Conference on Prac-*

tical Applications of Intelligent Agents and Multi-Agent Technology, pages 303 – 317. The Practical Application Company Ltd, April 1997.

[98] J. Harland and R. Kotagiri. An Aditi Implementation of a Flights Database. *Applications of Logic Databases*, 1994. Available at <http://www.cs.rmit.edu.au/~jah/publications/book94.ps.gz>.

[99] M.Z. Hasan, A.O. Mendelzon, and D. Vista. Visual Web Surfing with Hy^+. In *Proceedings of the 1995 IBM CASCON Conference*, pages 524 – 535, November 1997. Available at <http://www.cs.ubc.ca/doc/rr1/proceedings/cascon95/htm/francais/abs/hasanmv.htm>.

[100] F.J. Hauck. Supporting Hierarchical Guided Tours in the World Wide Web. In *Proceedings of the 5th International World Wide Web Conference*, May 1996. Available at <http://www5conf.inria.fr/fich_html/papers/P30/Overview.html>.

[101] P. Hill (Web page author), editor. *Proceedings of LOCOS'97: Workshop on Logic-based Composition of Software at the 14th International Conference on Logic Programming*, 1997. Available at <http://www.scs.leeds.ac.uk/hill/locos/accepted.html>.

[102] T. Hoppe, C. Kindermann, O.K. Paulus, and R. Tolksdorf. The MIHMA Project: a Web Information Service Based on Description Logics. In *Proceedings of the Workshop on "Artificial Intelligence-based tools to help WWW users" at the 5th International World Wide Web Conference*, May 1996. Available at <http://www.info.unicaen.fr/∼serge/3wia/workshop/papers/paper25.html>.

[103] R. Hurwitz and J.C. Mallery. The Open Meeting: A Web-Based System for Conferencing and Collaboration. In *Proceedings of the 4th International World Wide Web Conference*, December 1995. Available at <http://www.ai.mit.edu/projects/iiip/doc/open-meeting/paper.html>.

[104] IF Computer. Minerva. 1997. Available at <http://www.ifcomputer.com/MINERVA/>.

[105] ILOG. ILOG: Optimization and Visualization Software Components for Strategic Advantage. Available at <http://www.ilog.fr/>.

[106] Intelligent Systems Laboratory, Swedish Institute of Computer Science. SICStus Prolog User's Manual. October 1996. Available at <ftp://ftp.sics.se/archive/sicstus3/sicstus.ps.gz>.

[107] T. Joachims, T. Mitchell, D. Freitag, and R. Armstrong. WebWatcher: Machine Learning and Hypertext. *Fachgruppentreffen Maschinelles Lernen, Dortmund, Germany*, August 1995. Available at <http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/webwatcher/mltagung-e.ps.Z>.

[108] G. Karjoth, D. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, 1(4), July/August 1997. Available at <http://www.computer.org/internet/ic1997/w4toc.htm>.

[109] H. Kauffmann and A. Grumbach. MULTILOG: Multiple Worlds in Logic Programming. In B. Du Boulay, D. Hogg, and L. Steels, editors, *Advances in Artificial Intelligence*, pages 233 – 247. Elsevier Science, 1987.

[110] R. Khare and A. Rifkin. XML: A Door to Automated Web Applications. *IEEE Internet Computing*, 1(4), July/August 1997. Available at <http://www.computer.org/internet/ic1997/w4toc.htm>.

[111] N. Kino. JIPL. 1997. Available at <http://Prolog.isac.co.jp/jipl/index_e.html>.

[112] T. Kirk. Knowledge Based Access to Information on the World Wide Web. In *Proceedings of the Workshop on "Artificial Intelligence-based tools to help WWW users" at the 5th International World Wide Web Conference*, May 1996. Available at <http://www.info.unicaen.fr/~serge/3wia/workshop/papers/paper20.html>.

[113] T. Kirk, A.Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *On-line Working Notes of the AAAI Spring Symposium Series on Information Gathering from Distributed, Heterogeneous Environments*, January 1995. Available at <http://www.isi.edu/sims/knoblock/sss95/kirk.ps>.

[114] T. Kistler and J. Marais. WebL - A Programming Language for the Web. Technical Report 1997-029, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, December 1997. Available at <http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/SRC-1997-029-html/>.

[115] D. Konopnicki. Information Gathering in the World Wide Web: The W3QL Query Language and the W3QS System. Master's thesis, Computer Science Department, Technion (Israel Institute of Technology), 1996.

[116] R. Kotagiri and J. Harland. An Introduction to Deductive Database Languages and Systems. *VLDB Journal*, 3(2):107 – 122, April 1994.

[117] R.A. Kowalski. Using Meta-Logic to Reconcile Reactive with Rational Agents. In K. Apt and F. Turini, editors, *Meta-Logic and Logic Programming*, pages 227 – 242. MIT Press, 1995. Available at

<http://www-lp.doc.ic.ac.uk:80/UserPages/staff/rak/agents.ps.gz>.

[118] C.S. Kwok. A Survey of Structuring Mechanisms for Logic Programs. Technical report, Department of Computing, Imperial College, London, November 1988.

[119] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. A Declarative Language for Querying and Restructuring the World Wide Web. In *Proceedings of the Post-ICDE IEEE Workshop on Research Issues in Data Engineering (RIDE-NDS'96)*, February 1996. Available at <http://www.cs.concordia.ca/~grad/subbu/publications.html>.

[120] B.A. LaMacchia. Internet Fish. Technical Report AITR-1579, PhD Thesis at the Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), June 1996. Available at <http://www.farcaster.com/thesis/ifish-tr.ps.gz>.

[121] D. Lee and C.P. Tsang. Solving the Database Update Problem Using Linear Logic. In *Proceedings of the 7th Australasian Database Conference*, pages 131 – 138, 1996.

[122] H. Lieberman. Letizia: An Agent That Assists Web Browsing. In *Proceedings of the International Joint Conference on Artificial Intelligence, Montreal*, 1995. Available at <http://lieber.www.media.mit.edu/people/lieber/Lieberary/Letizia/Letizia.html>.

[123] A. Littleford. Artificial Intelligence and Hypermedia. In E. Berk and J. Devlin, editors, *Software Engineering Series, Hypertext/Hypermedia Handbook*, pages 357 – 378. McGraw-Hill Publishing Company, Inc., 1991.

[124] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[125] S.W. Loke. LogicWeb: Using Logic Programming to Extend the World Wide Web. In *Proceedings of the 2nd Joint AUUG and Asia Pacific World Wide Web Conference*, pages 220 – 226, Melbourne, Australia, September 1996.

[126] S.W. Loke and A. Davison. Logic Programming with the World Wide Web. In *Proceedings of the 7th ACM Conference on Hypertext*, pages 235 – 245. ACM Press, March 1996. Available at <http://www.cs.unc.edu/~barman/HT96/P14/lpwww.html>.

[127] S.W. Loke and A. Davison. A Two-level World Wide Web Model with Logic Programming Links. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 41 – 54, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/loke.ps>.

[128] S.W. Loke and A. Davison. A Logic Programming Approach to Generating Web-based Guided Tours. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Prolog*, pages 191 – 203, London, U.K., April 1997.

[129] S.W. Loke and A. Davison. LogicWeb: Enhancing the Web with Logic Programming. *Journal of Logic Programming*, 36(3):195 – 240, September 1998.

[130] S.W. Loke, A. Davison, and L. Sterling. Lightweight Deductive Databases on the World Wide Web. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 91 – 106, Bonn, Germany, September 1996. Available at

<http://clement.info.umoncton.ca/˜lpnet/lp-internet/lwddbs/lwddbs.html>.

[131] S.W. Loke, A. Davison, and L. Sterling. CIFI: An Intelligent Agent for Citation Finding on the World Wide Web. In *Topics in Artificial Intelligence, Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence (Lecture Notes in Artificial Intelligence No. 1114)*, pages 580 – 592, 1996.

[132] S.W. Loke, L. Sterling, E. Sonenberg, and H. Kim. ARIS: A Shell for Information Agents that Exploit Web Site Structure. In *Proceedings of the 3rd Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology* (to appear), April 1998.

[133] LPA. LPA ProWeb Server. 1997. Available at <http://www.lpa.co.uk/>.

[134] S. Luke, L. Spector, and D. Rager. Ontology-Based Knowledge Discovery on the World Wide Web. In *Proceedings of the Workshop on Internet-based Information Systems at AAAI '96*, Portland, Oregon, U.S.A., 1996. Available at <http://www.cs.umd.edu/projects/plus/SHOE/aaai-paper.html>.

[135] S. Lüttringhaus-Kappel and D. Schulz. A Calendar of Events - Architecture and Experiences. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 29 – 41, Bonn, Germany, September 1996. Available at <http://clement.info.umoncton.ca/˜lpnet/lp-internet/announce/announce.html>.

[136] D. Maier, L. Delcambre, L. Anderson, and R. Reddy. Modeling Explicit Semantics Over a Universe of Information. In *Proceedings of the*

3rd GCA International HyTime Conference, August 1996. Available at <http://www.hightext.com/IHC96/lmd7.htm>.

[137] M. Maloney. Hypertext Links in HTML. Available at <http://www.sq.com/papers/Relationships.html>.

[138] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1006 – 1023. MIT Press, 1988.

[139] J. Mayfield. Two-level Models of Hypertext. In C. Nicholas and J. Mayfield, editors, *Intelligent Hypertext: Advanced Techniques for the World Wide Web (Lecture Notes in Computer Science No. 1326)*, pages 90 – 108. Springer-Verlag, 1997.

[140] F.G. McCabe. *Logic And Objects*. Prentice Hall, 1992.

[141] F.G. McCabe and K.L. Clark. April - Agent Process Interaction Language. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents (Lecture Notes in Artificial Intelligence No. 890)*. Springer-Verlag, 1995.

[142] D.L. McGuiness, H. Manning, and T.W. Beattie. Knowledge Assisted Search. In *Proceedings of the Workshop on the Future of AI and the Internet at IJCAI-97*. 1997.

[143] M. Meier. ProTcXl - the Prolog Interface to Tcl/Tk and Xlib. Web site at <http://www.ecrc.de/eclipse/html/protcl/protcl.html>.

[144] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54 – 67, 1997. Available at <http://www.cs.toronto.edu/~mendel/papers.html>.

[145] A. Mendelzon and T. Milo. Formal Models of the Web. In *Proceedings of the Principles of Database Systems, PODS '97*, May 1997. Available from <http://www.cs.toronto.edu/~mendel/papers.html>.

[146] D. Merritt. *Building Expert Systems in Prolog*. Springer-Verlag, 1989.

[147] J. Meseguer and C. Talcott. Rewriting Logic and Secure Mobility. In *Proceedings of Foundations for Secure Mobile Code Workshop*, 1997. Available at <http://www.cs.nps.navy.mil/research/languages/statements/meseguer.ps>.

[148] D. Miller. A Theory of Modules for Logic Programming. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 106 – 114. IEEE Computer Society Press, 1986.

[149] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(1 & 2):79 – 108, January/March 1989.

[150] A.E. Monge and C.P. Elkan. The WebFind Tool for Finding Scientific Papers over the World Wide Web. In *Proceedings of the 3rd International Congress on Computer Science Research*, Tijuana, Baja California, Mexico, November 1996. Available at <http://www.cs.ucsd.edu/users/amonge/Papers/ciicc96.ps>.

[151] L. Monteiro and A. Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutter, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115 – 147. MIT Press, 1993.

[152] J.J. Moreno-Navarro. Tuple Inheritance: A New Kind of Inheritance for (Constraint) Logic Programming (Extended Abstract). In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, page 829. MIT Press, 1995. Full paper is available at <http://gedeon.ls.fi.upm.es/~jjmoreno/pap_bib.html#inh>.

[153] Motiv Systems, Ltd. The Java White Paper: The HotJava World Wide Web Browser. 1996. Available at <http://java.motiv.co.uk/intro/javawhitepaper_7.html>.

[154] NCSA. NCSA Mosaic for the X Window System. Available at <http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/>.

[155] Netscape Communications Corporation. JavaScript Resources. Available at <http://home.netscape.com/comprod/products/navigator/version_2.0/script/script_info/index.html>.

[156] C.K. Nicholas and L.H. Rosenberg. Canto: A Hypertext Data Model. *Electronic Publishing*, 6(1):1–23, March 1993.

[157] D. Nicol, C. Smeaton, and A.F. Slater. Footsteps: Trail-blazing the Web. *Proceedings of the 3rd International World Wide Web Conference, Computer Networks and ISDN Systems*, 27(6), April 1995. Available at <http://www.igd.fhg.de/www/www95/proceedings/papers/60/footsteps.html>.

[158] U. Nilsson and L. Naish. The Program Committee Virtual Workbench. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 1 – 12, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/nilsson.ps>.

[159] Object Management Group. Comparing ActiveX and CORBA/IIOP. Available at <http://www.omg.org/news/activex.htm>.

[160] R. O'Keefe. Towards An Algebra for Constructing Logic Programs. In J. Cohen and J. Conery, editors, *Proceedings of the IEEE Symposium on Logic Programming*, pages 152 – 160. IEEE Computer Society Press, 1985.

[161] R.A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[162] D.E. O'Leary. AI and Navigation on the Internet and Intranet. *IEEE Expert*, pages 8 – 10, April 1996.

[163] J.K. Ousterhout. The Safe-Tcl Security Model (DRAFT). March 1997. Available at <`http://www.sunlabs.com/people/john.ousterhout/safeTcl.html`>.

[164] M. Perkowitz, R.B. Doorenbos, O. Etzioni, and D.S. Weld. Learning to Understand Information on the Internet: An Example -Based Approach. *Journal of Intelligent Information Systems*, 8(2), March/April 1997. Available at <`http://www.cs.washington.edu/homes/map/ila.html`>.

[165] J.E. Pitkow and R.K. Jones. Supporting the Web: A Distributed Hyperlink Database System. In *Proceedings of the 5th International World Wide Web Conference*, April 1997. Available at <`http://www5conf.inria.fr/fich_html/papers/P10/Overview.html`>.

[166] E. Pontelli and G. Gupta. W-ACE: A Logic Language for Intelligent Internet Programming. In *Proceedings of the IEEE 9th International Conference on Tools with Artificial Intelligence*, pages 2 – 10, 1997.

[167] H. Prendinger. Logic Programming Methods for Searching the Web (Preliminary Report). In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 68 – 81, July 1997. Available at

<http://clement.info.umoncton.ca/~lpnet/proceedings97/prendinger.ps>.

[168] V. Prevelakis. *A Model for the Organisation and Dynamic Reconfiguration of Information Networks*. PhD thesis, University of Geneva, 1995.

[169] V. Prevelakis. A Framework for the Organisation and Dynamic Reconfiguration of the World Wide Web. In *Proceedings of the 5th Hellenic Conference on Informatics*, 1996. Available at <http://senanet.com/epy/18/www/epy.htm>.

[170] R. Price. No More "Me Too" - Different Approaches to Logic Documents. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 28 – 40, July 1997. Available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/price.ps>.

[171] D. Raggett, A. Le Hors, and I. Jacobs. HyperText Markup Language 4.0 Reference Specification. December 1997. Available at <http://www.w3.org/TR/REC-html40/>.

[172] G. Rossi. Programs As Data. *The Computer Journal*, 36(3):217 – 226, 1993.

[173] F. Rouaix. A Web Navigator with Applets in Caml. In *Proceedings of the 5th International World Wide Web Conference*. May 1996. Available at <http://pauillac.inria.fr/~rouaix/mmm/papers/Overview.html>.

[174] D. Rus and D. Subramanian. Information Retrieval, Information Structure, and Information Agents. In C. Nicholas and J. Mayfield, editors, *Intelligent*

Hypertext: Advanced Techniques for the World Wide Web (Lecture Notes in Computer Science No. 1326), pages 145 – 182. Springer-Verlag, 1997. Available at <`http://cs.cornell.edu/Info/People/rus/papers/Information-agents.ps`>.

[175] E. Sandewall. Towards a World-wide Data Base. In *Proceedings of the 5th International World Wide Web Conference*, May 1996. Available at <`http://www5conf.inria.fr/fich_html/papers/P54/Overview.html`>.

[176] K.E. Seamons, W. Winsborough, and M. Winslett. Internet Credential Acceptance Policies. In K. De Bosschere, M. Hermenegildo, and P. Tarau, editors, *Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (in conjunction with the 14th International Conference on Logic Programming)*, pages 13 – 27, July 1997. Available at <`http://clement.info.umoncton.ca/~lpnet/proceedings97/winsborough.ps`>.

[177] D. Sedlock and J. Jörg. Managing Software Projects with Prolog and the WWW. In *Proceedings of the 4th International Conference on the Practical Applications of Prolog*, London, U.K., 1996. Available at <`http://www.franken.de/users/nicklas/das/papers/pap96/pap96.html`>.

[178] A. Sehmi and M. Kroening. WebLS: A Custom Prolog Rule Engine for Providing Web-Based Tech Support. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 107 – 123, Bonn, Germany, September 1996. Available from

<http://clement.info.umoncton.ca/~lpnet/lp-internet/amzi/lspap.html>.

[179] E. Shapiro. Enhancing the WWW with Co-Presence. In *Presentation at the Workshop on "Wide-Area Collaboration and Cooperative Computing" of the 2nd International World Wide Web Conference*, 1994. Available at <http://www.ai.mit.edu/projects/iiip/colab/shapiro-abstract.html>.

[180] Y. Shoham. *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann, 1994.

[181] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1-3):17 – 64, October-December 1996.

[182] E. Spertus. ParaSite: Mining Structural Information on the Web. In *Proceedings of the 6th International World Wide Web Conference*, Santa Clara, U.S.A., 1997. Available at < http://www6.nttlabs.com/HyperNews/get/PAPER206.html>.

[183] Spyglass, Inc. Software Development Interface. 1996. Available at <http://www.spyglass.com/products/smosaic/sdi/sdi_spec.html>.

[184] L. Sterling and R.D. Beer. Metainterpreters for Expert System Construction. *Journal of Logic Programming*, 6(1 & 2):163–178, January/March 1989.

[185] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[186] L. Sterling and Y. Yalcinalp. Logic Programming and Software Engineering - Implications for Software Design. *Knowledge Engineering Review*, 11(4):317– 332, December 1996. Also available as Technical Report 96/31

at <http://www.cs.mu.oz.au/publications/tr_db/mu_96_31.ps.gz>.

[187] Sun Microsystems, Inc. Java Home Page. Available at <http:/java.sun.com/>.

[188] Sun Microsystems, Inc. Secure Computing with Java: Now and the Future (Executive Summary). 1997. Available at <http://www.javasoft.com/marketing/collateral/security.html>.

[189] R. Swick. Resource Description Framework (RDF). December 1997. Available at <http://www.w3.org/RDF/>.

[190] P. Szeredi, K. Molnár, and R. Scott. Serving Multiple HTML Clients from a Prolog Application. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 81 – 90, Bonn, Germany, September 1996. Available at <http://clement.info.umoncton.ca/~lpnet/lp-internet/iqsoft/multiple.html>.

[191] P. Tarau. BinProlog 3.30, User Guide. Technical report, Departement d'Informatique, Universitá de Moncton, Moncton,Canada, February 1995.

[192] P. Tarau. Linda Interactor. 1997. Distribution available at <http://clement.info.umoncton.ca/~tarau/LindaInteractor.tar.gz>.

[193] P. Tarau and V. Dahl. Mobile Threads through First Order Continuations. 1997. Available at <http://clement.info.umoncton.ca/html/tmob/html.html>.

[194] P. Tarau, V. Dahl, and K. de Bosschere. A Logic Programming Infrastructure for Internet Programming, Mobile Code, and Agents. 1997. Available at <http://clement.info.umoncton.ca/html/mobile/html.html>.

[195] P. Tarau, V. Dahl, and K. De Bosschere. Remote Execution, Mobile Code and Agents in BinProlog. In *Proceedings of the Workshop on Logic Programming and the Web at the 6th International World Wide Web Conference*, Santa Clara, U.S.A., 1997. Available at <http://www.cs.vu.nl/~eliens/WWW6/papers/tarau/>.

[196] P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors. *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, Bonn, Germany, September 1996. Available at <http://clement.info.umoncton.ca/~lpnet>.

[197] P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications (in conjunction with the Joint International Conference and Symposium on Logic Programming)*, pages 63 – 80. Bonn, Germany, September 1996. Available at <http://clement.info.umoncton.ca/~lpnet/lp-internet/lpnet10/art.html>.

[198] R.H. Trigg. Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment. *ACM Transactions on Office Information Systems*, 6(4):398 – 414, October 1988.

[199] T. Uustalu. Combining Object-Oriented and Logic Paradigms: A Modal Logic Programming Approach. In O. Lehrmann Madsen, editor, *Proceed-*

ings of the 6th European Conference on Object-Oriented Programming '92 (Lecture Notes in Computer Science No. 615), pages 98 – 113. Springer-Verlag, July 1992.

[200] J. van Rossum, G. van Rossum, and K. Manheimer. Python Language Home Page. Available at <`http://www.python.org/`>.

[201] P. Vasey. Underwriting on the Web Using Prolog. In *PAP'97 tutorial at the 5th International Conference and Exhibition on the Practical Application of Prolog*, 1997.

[202] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet (Lecture Notes in Computer Science No. 1222)*. Springer-Verlag, April 1997.

[203] D. Volpano. Provably-Secure Programming Languages for Remote Evaluation. In *Proceedings of Foundations for Secure Mobile Code Workshop*, 1997. Available at <`http://www.cs.nps.navy.mil/research/languages/statements/necula.ps`>.

[204] G. Wagner. Artificial Agents and Logic Programming. In *Panel statement at the workshop on Logic Programming and Multi-agent Systems at the 14th International Conference on Logic Programming*, July 1997. Available at <`http://www.informatik.uni-leipzig.de/~gwagner/lpmas.ps.gz`>.

[205] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. Extensible Security Architectures for Java. Technical Report TR 546-97, Department of Computer Science, Princeton University, April 1997.

[206] D.A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall International Ltd., 1990.

[207] webMethods, Inc. webMethods - Web Interface Definition Language. Available at <http://www.webmethods.com/technology/widl_description.html>.

[208] S. Weibel. A Proposed Convention for Embedding Metadata in HTML. Available at <http://www.oclc.org:5046/~weibel/html-meta.html>.

[209] J. Wielemaker. SWI-Prolog Reference Manual. Technical report, University of Amsterdam, Department of Social Sciences Informatics (SWI), April 1995.

[210] M. Winikoff. W-Prolog 1.0. 1996. Documentation and system available at <http://www.cs.mu.oz.au/~winikoff/wp/>.

[211] M. Winikoff. *Logic Programming with Linear Logic*. PhD thesis, Department of Computer Science, The University of Melbourne, 1997.

[212] C. Wong. *Web Client Programming*. O' Reilly and Associates, Inc., 1997.

[213] A. Wyatt. *SportsFinder: An Information Extraction Agent to Return Sports Results*. Honour's thesis, Department of Computer Science, The University of Melbourne, 1997.

[214] S. Yokoi. A Prolog Based Object-oriented Language SPOOL and its Compiler. In *Proceedings of the 5th Conference on Logic Programming (Lecture Notes in Computer Science No. 264)*, pages 116 – 125. Springer-Verlag, June 1986.

Appendix A

Prolog Facts Storing the Title, Body, Sections, and Links to Images and Applets on a Page

HTML contains tags to mark-up text as the title, body, and sections of a page, and to embed images and applets. These components are extracted on demand as noted in Chapter 4 and are stored in the following five types of facts:

- **title**. `title(Title)` stores the text between the tags `<TITLE>` and `</TITLE>`. The goal `lw(get, URL)#>title(Title)` retrieves the title of the page at `URL`. In the first call to `title/1`, the page is parsed and a `title/1` fact is asserted into the program. Any subsequent calls to `title/1` retrieves information from the `title/1` fact, without re-parsing the page.

- **body**. `body(Text)` stores the text between the tags `<BODY>` and `</BODY>`. Like `title/1`, this fact is generated the first time a call to `body/1` is made.

- **sections**. `section(Title, Text, Level)` facts store the title, text, and level of the page's sections. The title is the text between the tags `<Hn>` and `</Hn>`. Level is n which is an integer from 1 to 6 indicating the section level, and text is the portion of text between the end tag of this section and the start tag of the next (or end of file).

  In the first call to `section/3`, all `section/3` facts are generated regardless of the arguments in the call. Then, variables in the goal are instantiated returning information in the same call. Subsequent calls to `section/3` retrieve information from the created `section/3` facts. The same goes for the `image/2` and `applet/5` facts described below.

- **images**. `image(ALTText, URL)` stores the value of the `ALT` attribute (i.e., the text to use in place of the image), and the URL of the image. For example, the marked-up text:

  ```
  <IMG SRC="http://www.prost.org/beer.gif" ALT="beer picture">
  ```

  becomes:

  ```
  image("beer picture", "http://www.prost.org/beer.gif").
  ```

- **applets**. `applet(CODEBASE, CODE, ALT, NAME, Description)` stores the base URL of the applet, the applet class name, the alternative text, the applet name, and the applet description. For example, the tags which include an applet within a page:

  ```
  <APPLET CODEBASE="http://base.applet.url/" CODE="applet.class"
          ALT="should be an applet here" NAME="applet-name"
          WIDTH=10 HEIGHT=10 ALIGN=middle>
  <PARAM NAME="audio" VALUE="on">
  Here is my applet.
  </APPLET>
  ```

  is translated into the fact:

```
applet("http://base.applet.url/", "applet.class",
       "should be an applet here", "applet-name",
       "Here is my applet.").
```

Other information such as parameters to the applet and size of the applet display area for use by the browser are not stored.

Appendix B

Application Support Library

This appendix contains a complete list of the 9 predicates (in 6 categories) which are built into the LogicWeb language to facilitate construction of the applications described in this thesis. These predicates are used for communicating with Mosaic, constructing HTML documents, fast string matching, comparing dates, checking if a LogicWeb program exists in the program store, and deleting programs in the program store. The predicates mentioned here are additional to the SWI-Prolog built-in predicates which LogicWeb programs can use (see Chapter 4).

B.1 Displaying Information on the Mosaic Browser

`display_page(`*`BaseURL, PageComponents`*`)`: this predicate is used for sending information to the browser. It uses the CCI library to communicate with Mosaic. *`BaseURL`* is the base URL of the displayed page and is added to the page by the system via the mark-up:

```
<BASE HREF= BaseURL>
```

`BaseURL` is displayed on Mosaic's URL window.

`PageComponents` is a list of terms of the form `data(String)` or `form(URL, FormElements)`. `String` is a SWI-Prolog string. `data/1` terms carry text to be included in a HTML page. A `form/2` term with `URL` as an argument defines a form acting as the interface to the LogicWeb program identified by `lw(get, URL)`. `FormElements` is a list of terms each of which represents a component of a form. These terms are illustrated below via examples:

- `text(Data)`: the terms `text("hello "), text("there")` are converted into the following text:

  ```
  hello there
  ```

- `textnl(Data)`: the terms `textnl("hello "), text("there")` are converted into the following text:

  ```
  hello
  there
  ```

- `input(Type, Name, Value, Checked, Size, Maxlength)`: the term

  ```
  input("text", "key", "none", "", "5", "6")
  ```

  is converted into the following HTML mark-up:

  ```
  <INPUT TYPE="text" NAME="key" Value="none" SIZE=5 MAXLENGTH=6>
  ```

- `select(Name, Size, Multiple, OptionList)`: the term

  ```
  select("choices", "10", "", ["choice1", "choice2", "choice3"])
  ```

  is converted into the following mark-up:

```
<SELECT NAME="choices" SIZE=10>
<OPTION SELECTED> choice1
<OPTION> choice2
<OPTION> choice3
</SELECT>
```

- `textarea(`*`Name`*`, `*`Rows`*`, `*`Cols`*`, `*`Contents`*`)`: the term

  ```
  textarea("text", 10, 4, "stuff")
  ```

  is converted into the following mark-up:

  ```
  <TEXTAREA NAME="text" ROWS=10 COLS=4>stuff</TEXTAREA>
  ```

The argument of the reserved predicate `interface/1` (introduced in Section 4.4.1) is a list of terms each of which is in one of the above forms.

B.2 Constructing HTML Components

The following predicates are used for constructing HTML pages. The input and output arguments to the predicates are lists of ASCII codes:

`build_head(`*`Title, Head`*`)`: a goal such as `build_head("Hello", Head)` instantiates `Head` with the list

```
<HEAD>
<TITLE>Hello</TITLE>
</HEAD>
```

`build_code(`*`Code, CodePart`*`)`: a goal such as `build_code("a :- b.", CodePart)` instantiates `CodePart` with the list

```
<LW_CODE>
a :- b.
</LW_CODE>
```

build_body(*Title*, *PageData*, *Body*): a goal such as build_body("Title", "Data", Body) instantiates Body with the list

```
<BODY>
<H1>Title</H1>
Data
</BODY>
```

build_whole(*Head*, *Body*, *Whole*): *Whole* is instantiated with the concatenation of the lists in *Head* and *Body*:

```
<HTML>
 Head
 Body
</HTML>
```

B.3 Fast String Matching

contains(*Text*, *Pattern*): this predicate determines if *Text* contains *Pattern*. This predicate takes arguments which can be lists of ASCII codes, SWI-Prolog strings, or atoms. contains/2 is a foreign predicate implemented in C for fast string matching.

B.4 Comparing Dates

getime(*GMTTime*, *LWTime*): this predicate compares an atom *GMTTime* in the typical format of times returned by HTTP servers (e.g., "Thu Jun 27 04:44:42 GMT 1996" or "Tue, 11 Feb 1997 02:41:00 GMT") with an atom *LWTime* in the form "1996;06;27;14:44" specifying the year, month, day, hours, and minutes, and succeeds if the first time is greater (or later) than or equivalent to the second.

B.5 Determining If a Program Exists in the System

`program_exists(`*`LWProgramID`*`)`: this predicate takes a LogicWeb program identifier *`LWProgramID`* as an argument and determines if the program is in the program store, succeeding if so, and failing, otherwise.

B.6 Deleting Programs

`delete_programs/0`: this predicate deletes (using `retract/1` repeatedly) all the LogicWeb programs from the SWI-Prolog database except the LogicWeb program manager (see Section 4.4.4). The LogicWeb program manager is identified by the filename “lwmanager.html”.