LogicWeb: Enhancing the Web with Logic Programming

Andrew Davison

Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112, Thailand
Email: ad@ratree.psu.ac.th

Seng Wai Loke

Dept. of Computer Science
University of Melbourne
Parkville, Victoria 3052, Australia
Email: swloke@cs.mu.oz.au

December 23rd, 1996

Abstract

LogicWeb is a client-side logic programming tool for the World Wide Web, which allows the Web to be viewed in a more abstract way: Web pages can be rephrased as logic programming modules, and hypertext links as relationships between the modules.

This abstraction makes LogicWeb particularly suitable for coding important classes of applications, and this paper considers two in some detail: Web search, and the structuring of Web information using deductive databases.

LogicWeb illustrates that logic programming possesses many advantages for writing Web applications, including the simple representation of information (e.g. as deductive databases or as logic grammars), the ability to write meta-level descriptions (e.g. of pages and the connections between pages), and the encoding of rules and heuristics necessary for "intelligent" behaviour.

1 Introduction

The World Wide Web's popularity derives from its support for the global publication of pages which contain graphics, sound, animations, 3D images, and so on. The Web is also an excellent information source, made accessible through numerous search engines.

A drawback of basic Web pages is their limited executable behaviour, which is essentially restricted to clicking on hypertext links. This has been addressed by two classes of programming extensions, one based on server-side evaluation, the other on client-side computation.

Server-side evaluation typically involves the user in completing a form on their browser, which is submitted across the network to a Web server to be processed. The most widespread server-side evaluation mechanism is the Common Gateway Interface (CGI) which delivers form details to programs, and routes any output from the code back to the user [Davison 1995].

One disadvantage of server-side programming is the difficulty of extending the user interface. For instance, it is not possible to intercept the activation of a hypertext link or to augment the forms interface with additional GUI elements. Also, since server-side scripts are usually located on different machines from the forms which use them, communication latency can be a problem. A further drawback is the load on the server caused by multiple clients running scripts.

The other kind of Web programming evaluates code on the client-side (i.e. on the user's browser). Two well-known languages of this type are JavaScript [Reynolds and Wooldridge 1996] and Java [Lemay and Perkins 1996]. The client-side approach allows programs to utilise a wider variety of browser features than server-side scripts, thereby increasing the possible types of user interaction. For instance, Java comes with a rich set of GUI class libraries, and JavaScript can access the browser's history list of recently retrieved pages. Typically, client-side code is downloaded with the page that uses it, and so network characteristics will not affect the program's execution.

A drawback of many client-side languages is their complexity. For example, Java supports the typical features of an imperative object oriented language (although it has removed pointers). This means that the representation of structured information, as found in databases for instance, involves the manipulation of an assortment of data structures. Also, these language offer little support for meta-level information, such as descriptions of pages and the relationships between pages. These capabilities are extremely useful for a diverse class of Web applications, including search and data mining.

Security is an issue with client-side programming since it relies on code being moved from a foreign host to be executed locally. Java has a number of interesting security features, but they can make it difficult to do common tasks such as file manipulation and Web page retrieval [Lemay and Perkins 1996].

There are two main aims of our work. The first is to utilise Logic Programming (LP) as a way of viewing the Web more abstractly than just as pages connected by hypertext links. Our *LogicWeb* system allows the Web to be manipulated as a collection of LP modules, which can be interrelated using familiar LP techniques [Loke and Davison 1996]. This view is not simply a pleasing abstraction, but is essential as a framework for writing Web programs that manipulate structured information or carry out meta-level reasoning.

The second aim is to investigate the use of LP as a client-side programming tool for the Web. The client-side approach is utilised because it offers the opportunity to provide novel forms of user interaction while avoiding network problems.

LogicWeb is introduced in section 2. Section 3 examines how Web search tools can be developed using LogicWeb, with an emphasis on the representation of search heuristics. Section 4 considers how structured Web information can be encoded as lightweight deductive databases. The implementation of LogicWeb is outlined in section 5. Other approaches to using LP with the Web are described in section 6. Section 7 contains a summary of the main points of this work, and some possible directions for future research.

The material presented here is an expansion of work reported in [Loke and Davison 1996; Loke et al. 1996a; Loke et al. 1996b]. These papers can be found at `http://www.cs.mu.oz.au/~swloke/logicweb.html`.

2 What is LogicWeb?

2.1 LogicWeb Overview

A simple view of the Web is as a collection of pages connected by hypertext links. A fragment might look like Figure 1. We shall make the assumption that every page has a unique address (its URL). In fact, there are exceptions to this, such as when a page is dynamically created by a CGI script.

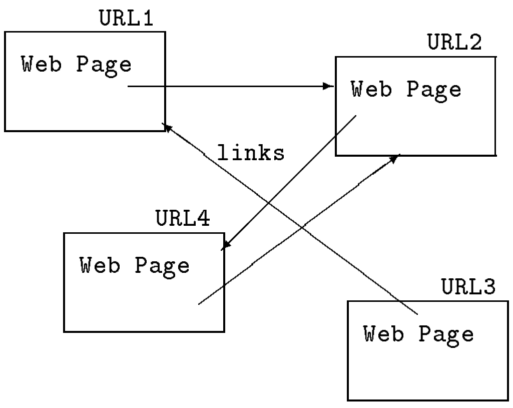

Figure 1: Web Pages connected by Hypertext links.

The LogicWeb view extends pages to become LP modules, and allows link connectivity to be augmented with LP relationships. The previous fragment could be viewed by LogicWeb as in Figure 2. Each module still contains the text of the Web pages, but now extended with LP code. How this combination is achieved is described below.

Another change of perspective is to label each module with an ID (e.g. `mod1`, `mod2` in Figure 2). The requirement is that each ID is unique, and LogicWeb achieves this by using the page's URL.

A module ID is represented by a term at the LP level, and so modules become amenable to meta-level reasoning. For instance, the fact:

```
is_a_summary_of(mod1, mod2).
```

specifies that `mod1` is a summary of `mod2`. This could be used by an on-line book previewer to direct the browser from a summary page (in `mod1`) to a more detailed page (in `mod2`), or to guide the browser from the details to the summary. Such facts (and rules) play an important role in LP-based Web search engines, as described in section 3.

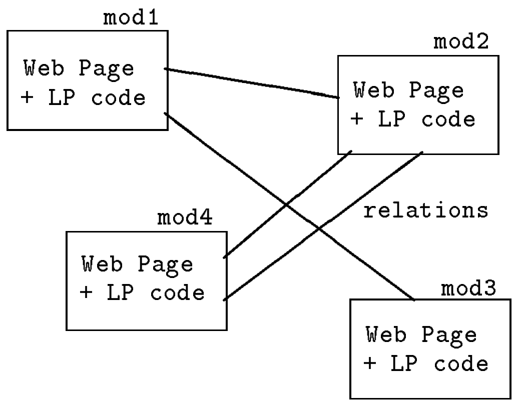

Figure 2: LogicWeb Modules connected by LP relationships.

LogicWeb is a client-side LP system, which is conceptually between the user's browser and the Web as shown in Figure 3.

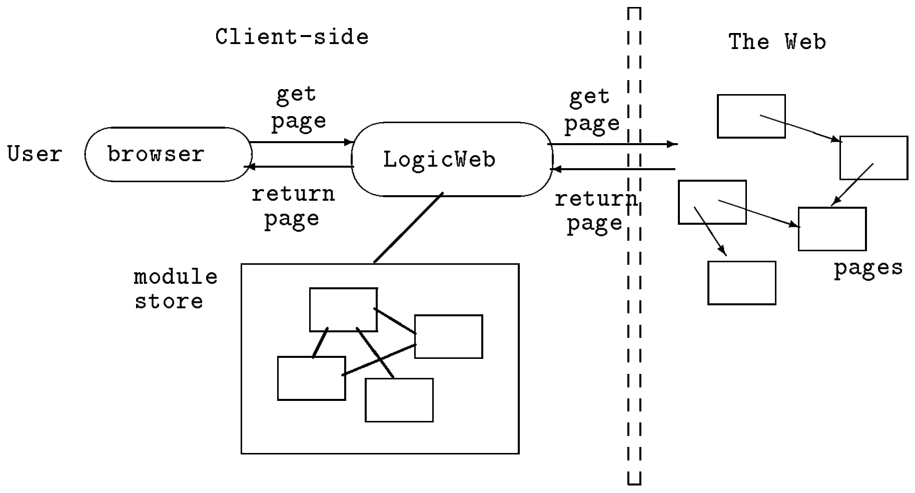

Figure 3: A Stylised Overview of the LogicWeb System.

Figure 3 illustrates (in a stylised form) how LogicWeb supports the LP abstraction of the Web. When a Web page is retrieved, it is displayed by the browser as normally. However, the page is also converted into a module and stored on the client-side. A downloaded page/module is a *copy* of the original page on the Web, which becomes significant when state change is considered.

The presence of LogicWeb between the browser and the Web means that it can interpret the user's input. For instance, LogicWeb can convert a click on a hypertext link into a goal. This conversion allows programmers to write code which extends the meaning of a click beyond page retrieval.

LogicWeb converts all retrieved pages into modules, including those with no additional LP code. The basic transformation generates two facts for each Web page:

```
my_id("URL").
h_text("Page text").
```

`my_id/1` holds the module's ID which is its URL, while `h_text/1` contains the complete text of the Web page (including its HTML tags) as a string. `my_id/1` allows the module to refer to itself. An example is when a program must determine if its module ID is different from that of another.

LogicWeb also parses each page looking for link information so that a `link/2` predicate can be generated. For instance, the line:

```
I like <a href="http://www.prost.org/beermake.html">beer making</a>.
```

becomes:

```
link("beer making", "http://www.prost.org/beermake.html").
```

LogicWeb can also generate facts about the page structure, including a `title/1` fact and similar information about section, sub-section, and other headings.

We have still not determined what constitutes a useful set of automatically generated predicates, but such predicates can be readily created by the programmer by calling built-in parsing utilities.

LP code appears on a page inside a `<lw_code>...</lw_code>` container. For instance, the following could be part of my home page:

```
<lw_code>
interests(["Logic Programming", "AI", "Web", "OOP"]).

related("logic", "Logic Programming").
related("agents", "AI").

useful_pages("Logic Programming",
                 ["http://www-lp.doc.ic.ac.uk/",
                  "http://www.cs.mu.oz.au/~ad/alp/archive.html"]).

interested_in(X) :-
  interests(Is), member(X, Is).
interested_in(X) :-
  related(X,Y), interested_in(Y).
</lw_code>
```

Typically, such code appears inside a `<pre>...</pre>` container so that is uninterpreted by the browser.

The facts and rules in the example illustrate the kinds of Web information that can be represented. The `interests/1` and `related/2` facts give details about the author of the page in a structured form that can be readily processed. The `useful_pages/2` fact could be employed by a search engine looking for LP information, or for details on logic (by using the appropriate `related/2` fact). The `interested_in/1` rules show how page information can be inferred.

Once a LogicWeb module has been downloaded, it is queried via a forms interface, which is added to its corresponding page before it is passed to the browser. Alternatively, a LogicWeb input form may already have been added to the page by its author.

The LP language utilised by LogicWeb is a fairly elementary Edinburgh-style Prolog with some additional module operators. The main operator applies a goal to a module specified by its module ID (i.e. its URL):

```
m_id(URL)#>Goal.
```

If the module is not present in the module store (see Figure 3) then its page will be downloaded and transformed into a LogicWeb module before the query is evaluated. However, if the module is already in the store then the goal is executed immediately. Thus, the "#>" operator permits the programmer to think of Web computation as goals applied to modules, with no need for explicit Web page retrieval or parsing.

LogicWeb contains a number of module composition operators, inspired by work on structured LP [Brogi et al. 1994a; Bugliesi et al. 1994], and contextual LP [Monteiro and Porto 1989], such as the union operator:

```
lw_union(ListofModuleIDS)#>Goal.
```

This creates the clause-wise union of the modules for the duration of the goal evaluation. See section 4.3 for an example.

LogicWeb is not alone in adding more machine-processable semantic content to pages. The recently proposed HTML 3.2 standard [HTML 3.2 1996] contains tags for meta-information based around name-value pairs. HTML links can also be classified by using attributes. In addition, [Luke et al. 1996] has proposed new HTML tags for incorporating ontology-based knowledge into pages, in the form of IS-A class hierarchies and instance-instance relationships. For example, a subsection can be related to another page by a named relationship.

2.2 Examples of LogicWeb Use

Both of the examples in this section are simple search utilities; more complicated search tools will be developed in section 3.

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) :-
  m_id(CurrURL)#>interested_in(Topic),
  m_id(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.

A serious drawback of this code is that it assumes 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.

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

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

will bind P to a URL which is related to logic programming and linked to the specified page. relevant_pg/3 is defined as:

```
relevant_pg(Subject, CurrURL, URL) :-
  m_id(CurrURL)#>link(_, URL),
  m_id(URL)#>h_text(Source),
  contains(Source, Subject).
```

relevant_pg/3 selects a link without concerning itself about the anchor. The text of the retrieved page is passed to the LogicWeb built-in contains/2 to see if it contains the subject string. relevant_pg/3 only relies on the predicates automatically added to modules by LogicWeb, and so should be more robust than similar_pg/2.

A weakness of the current LogicWeb implementation is the lack of backtracking at the top-level of query evaluation: once an answer has been returned to the user it is not possible to backtrack into the computation. For that reason, the usual LogicWeb coding style for multiple answers is to embed the top-level query in a setof/3 call, such as:

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

An answer is returned to the user as a complete Web page whose formatting can be left to the LogicWeb system, or can be specified by built-in predicates as part of a program.

2.3 LogicWeb Modules

The LogicWeb module mechanism is fairly unsophisticated. A module can employ two notions of visibility, specified as facts. The visible/1 predicate states the predicates which can be utilised by a user from a form (e.g. visible(interested_in/1)). The export/1 predicate states those predicates which can be utilised by other modules (e.g. export(interested_in/1)). A combination of these allows varying degrees of visibility. For instance, an exported predicate which is not visible is similar to a protected C++ member function.

More work needs to be done on the module system, perhaps to bring it into line with the proposed ISO Prolog module standard [Hodgson 1996].

State is crucial to most Web applications, and LogicWeb currently supports one of the two possible update mechanisms. LogicWeb programs can use: assert(m_id(URL), Clause) and retract(m_id(URL), Clause) to update the modules stored on the client-side. However, these are only copies of the original Web pages. Thus, once the current session finishes, the state changes will be lost. This drawback can be coded around by the use of files on the client-side which exist beyond the session. However, this still excludes applications which might want to use the original module's state as global data or a communications link. Changing this state is complicated by security concerns: Web administrators are reluctant to allow programs on their machines to be altered over the Internet.

3 Search

3.1 Limitations of Search Engines

The Web allows readers to browse related information with ease, but the Web is too large to be searched by manual browsing alone. This has led to a proliferation of search engines which use keywords to search indexes (e.g. Lycos, AltaVista). These indexes are generated by repeated off-line traversals over the Web.

Although these engines do a good job currently, they will become increasingly less accurate as the size of the Web increases. For instance, it will become ever harder to keep the indexes up-to-date, since it may take several weeks for a new site to be discovered, or for a changed page to be revisited. Pages may also be deleted or moved, making their index entries incorrect for a long period.

A related problem will be the size of the indexes, which will become unmanageably large as the Web continues to grow.

One solution will be to limit the indexes to "important" items, such as home pages and corporate sites. This will give rise to search tools which are tuned for specific domains, a trend which is already occurring[1]. A problem with these specialised engines is that several may need to be employed before a good answer is found, especially if the search item is difficult to categorise.

Even at the moment, search engines can return poor results if the item being looked for is hard to define using keywords. For instance, looking for pages containing paper citations is difficult to specify due to the variety of ways of describing a citation. Moreover, as noted in [Luke et al. 1996], keyword searches are based only on lexical or syntactic content. This means that the search results are very sensitive to the choice of words used in the queries. If a document is indexed on a synonym of the query keyword, then the document will not be retrieved. In addition, a word often has different meanings causing redundant information to appear in the results.

3.2 LogicWeb for Searching

Searching the Web is one of the main application areas for LP. This is not due to the depth-first, backtracking behaviour of many LP languages, which is actually a disadvantage due to the numerous loops in the Web's topology. The important LP features are the ability to specify search heuristics using meta-level techniques, and to code specialised forms of search, such as breadth-first or resource-bounded. These approaches are possible in LogicWeb because module IDs and the text of Web pages are first class entities.

In section 3.3, a simple heuristic-free search tool is developed which illustrates many of the coding techniques for Web search applications. Section 3.4 describes *page type hierarchies*, which are a way of describing the structure of the sites being searched. In section 3.5, a page type hierarchy is used to add heuristics to the search tool of section 3.3. Section 3.6 discusses a bounded breadth-first search utility, and section 3.7 explains how LogicWeb can be utilised alongside existing search engines.

3.3 Heuristic-free Search

`find_ctt/3` uses a list of keywords and a starting URL to look for citations. A typical call is:

```
?- find_ctt(["Web, "Logic Programming"],
            "http://www.cs.mu.oz.au/~ad", Citation).
```

Hopefully, `Citation` will be bound to a citation containing the two keywords. `find_ctt/3` is defined as:

```
find_ctt(Keys, URL, Ctt) :-
  m_id(URL)#>h_text(Source),
```

[1]See `http://home.netscape.com/escapes/search/special_guides.html`.

```
  contains_ctt(Keys, Source),
  extract_ctt(Keys, Source, Ctt).
find_ctt(Keys, URL, Ctt) :-
  m_id(URL)#>link(_, NxtURL),
  find_ctt(Keys, NxtURL, Ctt).
```

The first clause checks whether the text of the page at URL contains all the keywords by calling `contains_ctt/2`. If the goal succeeds, the citation is extracted from the page with `extract_ctt/3`.

The second clause is called when the `contains_ctt/2` goal fails. A link is chosen from the page, and followed by recursively calling `find_ctt/3`. This strategy relies on backtracking to return to the current page if the choice is unrewarded.

`contains_ctt/2` checks every key against the page by calling the built-in `contains/2`:

```
contains_ctt([], _).
contains_ctt([Key|Ks], Source) :-
  contains(Source, Key),
  contains_ctt(Ks, Source).
```

The main weakness of this code is the way that `find_ctt/3` blindly follows links in a depth-first manner. A link may go to the top of the same page or to a distantly related page at a different site. Search heuristics, such as those based on page type hierarchies, are needed to avoid these problems.

3.4 Page Type Hierarchies

A page type hierarchy is a way of describing a *class* of Web sites (e.g. academic department Web sites, on-line newsletters, financial information pages) using relationships between categories of pages. Page type hierarchies concentrate on the organisation of semantic content at related Web sites instead of on the textual information at any given site. The benefit of this approach is that a hierarchy can be used as a general purpose "map" of any site which falls into the class that the hierarchy represents.

For instance, our study of computing department Web sites suggested that they usually contain pages of departmental information, research details, projects, project members, and researchers. These page types can be labelled as `dept`, `research`, `project`, `proj_members` and `researcher`. Also, these types are related in a fairly standard way: a `dept` page is at the top of the departmental web site, with the successive page types further down through the site hierarchy. One page is further down the hierarchy than another page if the chain of hypertext links from the starting page to that page is longer.

The ordering of the page types can be captured with `composed_of/2`:

```
composed_of(dept, research).
composed_of(research, project).
composed_of(project, proj_members).
composed_of(proj_members, researcher).
```

`composed_of/2` defines a general relationship, which will not hold in all cases. However, it is still a useful sketch of the typical hierarchy of a computing department's Web site. It can be drawn as in Figure 4.

One subtlety of Figure 4 is that the lines joining the page types (which correspond to a `composed_of/2` fact) will not usually map to a single hypertext link. For example, several link dereferences may be required to get from a department page to a research page.

Other hierarchies can be developed, which emphasize different aspects of the site, such as:

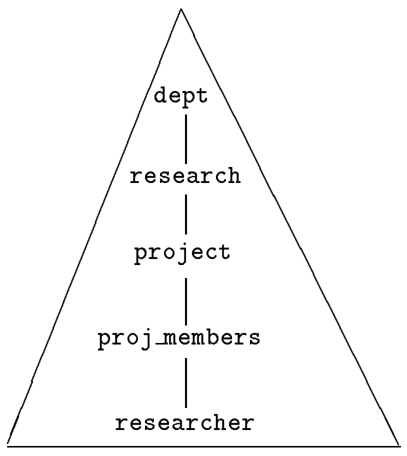

Figure 4: A Computing Department Page Type Hierarchy.

```
composed_of(X, Y) :-
  has_part(X, Y).
composed_of(X, Z) :-
  has_part(X, Y), composed_of(Y, Z).

has_part(section(prog_langs), group(decl_langs)).
has_part(group(decl_langs), project(mercury)).
has_part(group(decl_langs), project(lygon)).
has_part(section(prog_langs), project(lp_techniques)).
     :
```

The page types are the terms `section/1`, `group/1`, and `project/1`, which are parameterized with the section, group, or project names.

3.5 Heuristic Guided Search

`find_ctt/3` of section 3.1 can now be rephrased to use a page type hierarchy for CS department sites, as well as some other heuristics. A query is formulated as before:

```
?- find_ctt2(["Web, "Logic Programming"],
             "http://www.cs.mu.oz.au/~ad", Citation).
```

The new parts of `find_ctt2/3` are commented:

```
find_ctt2(Keys, URL, Ctt) :-
  likely_page(URL),             % *new*
  m_id(URL)#>h_text(Source),
  contains_ctt(Keys, Source),
  extract_ctt(Keys, Source, Ctt).
find_ctt2(Keys, URL, Ctt) :-
  m_id(URL)#>link(_, NxtURL),
  extension_of(NxtURL, URL),      % *new*
  pt_related(URL, NxtURL),        % *new*
  find_ctt2(Keys, NxtURL, Ctt).
```

The first clause uses `likely_page/1` to test whether the page belongs to a page type which is likely to contain a citation. For example, a `researcher` page is very likely to hold a citation. `likely_page/1`'s definition is:

```
likely_page(URL) :-
  m_id(URL)#>page_type(PT),
  is_ctt_pt(PT).
```

`is_ctt_pt/1` checks if the page type `PT` is one that might contain a citation. The `page_type/1` goal may be evaluated against an actual predicate in the module, or by resorting to an examination of keywords in the page's text.

The second clause of `find_ctt2/3` employs a simple syntactic check in `extension_of/2` to determine if `NxtURL` has an address derived from `URL`. For instance, `http://www.cs.mu.oz.au/~ad/papers.html` extends `http://www.cs.mu.oz.au/~ad`. Also, `pt_related/2` orders the two pages by comparing their page types to decide if the page type of `NxtURL` is nearer to the `researcher` pages where citations are most likely to be located.

```
pt_related(URL, NxtURL) :-
  m_id(URL)#>page_type(PT),
  m_id(NxtURL)#>page_type(NPT),
  composed_of(PT, NPT).
```

`composed_of/2` accesses a page type hierarchy like the ones described in section 3.2. However, it will be more complex due to the need to accept pages which are not of an interesting kind but may lead to an interesting page type. In addition, `composed_of/2` must recognise "bad" page types so that poor search paths can be eliminated. For instance, subtrees related to course content can be ignored when looking for citations.

3.6 Resource-bounded Breadth-first Search

The following bounded breadth-first search program shows how other types of search strategies can be readily encoded in LogicWeb. This code will also be used as a basis for talking about search accuracy, and how the unpredictable nature of the Web can be accommodated.

The top-level query is:

```
?- 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) :-
  m_id(URL)#>h_text(Text),
  score_page(Keys, Text, Score),
  act_score(Score, URL, 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(Score, CurrURL, ToVisit, Keys, PScore, Max, Ps, FPs) :-
  Score > PScore,
  setof(URL, [Label]^m_id(CurrURL)#>link(Label,URL), URLs), % get links
  append(ToVisit, URLs, ToVisit1),                          % store
  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).
```

`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. If the `append/3` call was:

```
  append(URLs, ToVisit, ToVisit1)
```

then a depth-first search would be carried out.

The second clause of `act_score/8` discards the page since its score is too low, and then continues with the collection.

`collect/6` illustrates how various search strategies can be easily programmed because module IDs and the text of Web pages are first class entities.

The accuracy of the search can be increased by utilising structured information in the visited pages. For instance, if we assume that Web pages generally contain `interests/1`, `related/2`, and the other predicates in the section 2.1 example , then the search performance can be improved.

For example, a new score predicate could utilise `interested_in/1`:

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

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

Another extension would be to add the URLs in `useful_pages/2` to the `ToVisit` list if they were related to any of the search keys.

When a page cannot be downloaded (e.g., when the server is down), the "`#>`" goal in the third clause of `collect/6` will fail. This can be avoided by replacing the goal with:

```
mod_text(URL, Text)
```

which is defined as:

```
mod_text(URL, Text) :-
  m_id(URL)#>h_text(Text), !.
mod_text(_, "").
```

A slightly more sophisticated version could cater for page retrieval failure by trying to retrieve the page from a mirror site:

```
mod_text(URL, _, Text) :-
  m_id(URL)#>h_text(Text), !.
mod_text(_, MirrorURL, Text) :-
  m_id(MirrorURL)#>h_text(Text), !.
mod_text(_, _, "").
```

This illustrates how LP non-determinism can reflect the non-deterministic nature of the Web, where network failure, load effects, and servers are unpredictable [Connor 1996].

3.7 LogicWeb and Existing Search Engines

Even though LogicWeb is very useful for coding search applications, better results are obtained if it is used in conjunction with conventional search engines. In the CiFi system [Loke et al. 1996a; Han et al. 1996], a range of engines are used to find good starting points for a LogicWeb-based search for a citation. Such starting points include the author's home page and the author's departmental home page. These pages are easy to define using keywords, and are likely to be among the first few answers returned by conventional search engines. In addition, CiFi employs several search engines that are specialised for computer science information (e.g. the New Zealand Digital Library, and the CS bibliography collections at the University of Karlsruhe).

Once a starting point has been determined, a LogicWeb search program takes over to browse through the pages beneath it. A heuristic-free search would almost certainly go into an infinite loop, or head off to an unrelated site. Instead, LogicWeb uses a variety of search heuristics. Some are based on predicates like `link/2` and `related/2` described in section 2.1, which are generated automatically by LogicWeb or extracted by the search program through parsing. CiFi also uses a page type hierarchy for a standard computing department Web site to guide its search.

One advantage of using a combination of search engines and browsing is its resilience to change in the Web, since home pages and departmental pages rarely change and are readily available through the conventional search engines. Also, the LogicWeb search component can cope with changes to pages so long as the underlying structure of the site is not altered too drastically. Combining several engines is necessary since no one engine contains all the useful citation information.

4 Lightweight Deductive Databases

LogicWeb allows the Web to be viewed as a distributed collection of deductive databases, where each database is represented by a LogicWeb module (or Web page). This has a number of advantages, the main one being that it offers a way of adding structured information to the Web. Such information can be searched, combined, and extracted using familiar techniques from deductive databases. Also, LogicWeb databases can be reused in various ways by the application of its composition operators, although the interfaces of the databases must be carefully designed. These databases are lightweight in the sense that they lack the functionality of full database systems, such as transaction processing, and query optimisation.

In section 4.1, we develop a simple set of lightweight deductive databases for research interests. This code is modified in section 4.2 to be more distributed without requiring major changes to the query mechanism. In section 4.3, we discuss how LogicWeb's union operator can improve the reusability of the code. Section 4.4 considers an important category of Web databases: those which cannot be downloaded over the Web, but can be queried on their servers.

More details on lightweight deductive databases can be found in [Loke et al. 1996b].

4.1 Finding Research Interests

We imagine that institutions store details of their academic interests in lightweight deductive databases. Each database contains facts of the form:

```
rs_ints( name(First, Last), net_info(Login, HomePageURL), [Interest,...]).
```

For instance, the database at the University of Melbourne might be:

```
% research information at Uni. Melb.
my_id("http://www.cs.mu.oz.au/ri.html").

rs_ints( name(andrew, davison),
     net_info("ad@cs.mu.oz.au", "http://www.cs.mu.oz.au/~ad"),
     ["Logic Programming", "AI", "Web", "OOP"]).
      :
```

The database at Imperial College in London might be:

```
% research information at Imperial.
my_id("http://www.doc.ic.ac.uk/ri.html").

rs_ints( name(keith, clark),
     net_info("klc@doc.ic.ac.uk", "http://www-lp.doc.ic.ac.uk/~klc"),
     ["Logic Programming", "Agents"]).
      :
```

In addition, we assume a central database at `http://www.res.info/rinfo.html` which lists the URLs of all the institute databases. It contains facts of the form:

```
institute(InstituteName, URL).
```

Thus, it might hold:

```
% institute info database
my_id("http://www.res.info/rinfo.html").

institute("Melbourne", "http://www.cs.mu.oz.au/ri.html").
institute("Imperial", "http://www.doc.ic.ac.uk/ri.html").
      :
```

An `acad_interest/2` predicate to find someone interested in a given topic can be expressed as:

```
acad_interest(Topic, Name) :-
  m_id("http://www.res.info/rinfo.html")#>institute(_, URL),
  m_id(URL)#>rs_ints(Name, _, Interests),
  member(Topic, Interests).
```

The rule chooses an institution, and uses its URL to obtain the research interests of someone. If the specified topic is one of the person's interests then his/her name is returned, otherwise backtracking will take place to look for other individuals, either at the same institution or elsewhere.

A problem with this code is that it will eventually load every institute database onto the client-side. Fortunately, LogicWeb contains operators to discard modules, so that memory usage can be kept under control.

The following `contacts/2` predicate returns the login IDs of all the people at a given institution:

```
contacts(Institute, Ls) :-
  m_id("http://www.res.info/rinfo.html")#>institute(Institute, URL),
  setof(Login, [N,U,I]^m_id(URL)#>rs_ints(N,net_info(Login,U),I), Ls).
```

4.2 A More Distributed Version

The institute databases can be subdivided so that the home page of each academic contains their research details. This has the advantage that the information can be maintained by the academics themselves.

The structure of the central database does not change, but each institute's database now contains facts of the form:

```
researcher(URL).
```

For instance, the database for the University of Melbourne becomes:

```
% research information at Uni. Melb.
my_id("http://www.cs.mu.oz.au/ri.html").

researcher("http://www.cs.mu.oz.au/~ad").
researcher("http://www.cs.mu.oz.au/~swloke").
      :
```

Each academic will now have a database in their home page which may contain a range of information in addition to their research interests. However, care must be taken that the old research details interface is maintained. For instance, Andrew Davison's home page may hold:

```
% research information for Andrew Davison.
my_id("http://www.cs.mu.oz.au/~ad").

rs_ints(name(F, L), net_info(Login, URL), Interests) :-
  name(F, L), login(Login),
  my_id(URL), interests(Interests).

name(andrew, davison).
login("ad@cs.mu.oz.au").
interests(["Logic Programming", "AI", "Web", "OOP"]).
      :
```

The increased distribution of information will have little effect on the predicates of the previous section. For example, `acad_interest/2` would change to:

```
acad_interest(Topic, Name) :-
  m_id("http://www.res.info/rinfo.html")#>institute(_, URL),
  m_id(URL)#>researcher(RURL),    % *new*
  m_id(RURL)#>rs_ints(Name, _, Interests),
  member(Topic, Interests).
```

The extra level of distribution is reflected in the extra module call.

4.3 Reusability Using Union

A drawback of `acad_interest/2` is that it contains the chain of URLs which need to be followed to find researcher information. This means that any changes to the chain requires a change to `acad_interest/2`, as occurred in the last section.

`acad_interest/2` would be more reusable if it did not contain this chaining information. For instance, assume the existence of the module:

```
my_id("http://www.res.info/acadi.html").

acad_interest(Topic, Name) :-
  rs_ints(Name, _, Interests),
  member(Topic, Interests).
```

How can this be used, since it makes no reference to the modules where `rs_ints/3` is defined? The answer is to combine `acadi.html` with the relevant modules by using LogicWeb's union operator, `lw_union/1`.

For example, the URLs of researchers from the University of Melbourne can be collected using:

```
melb_people(RUs) :-
  m_id("http://www.res.info/rinfo.html")#>institute("Melbourne", URL),
  setof(m_id(RURL), m_id(URL)#>researcher(RURL), RUs).
```

The Melbourne researchers interested in logic programming can then be expressed as:

```
?- melb_people(RUs),
   lw_union([m_id("http://www.res.info/acadi.html")|RUs])#>
                              acad_interest("Logic Programming", Name).
```

The `lw_union/1` version of "#>" creates a clause-wise union of the modules specified in the list, and so `acad_interest/2` will utilise the `rs_ints/3` facts of the Melbourne researchers.

An advantage of this approach is that `acad_interest/2` can be used for searches over other subsets of researchers without modification. For instance, Imperial College people interested in artificial intelligence can be found using:

```
ic_people(RUs) :-
  m_id("http://www.res.info/rinfo.html")#>institute("Imperial", URL),
  setof(m_id(RURL), m_id(URL)#>researcher(RURL), RUs).

?- ic_people(RUs),
   lw_union([m_id("http://www.res.info/acadi.html")|RUs])#>
                              acad_interest("AI", Name).
```

Several other LogicWeb composition operators are discussed in [Loke et al. 1996b].

4.4 Server-side Databases

A possible disadvantage of LogicWeb for database manipulation is its use of client-side processing, which means that a database must be downloaded to the user's browser 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.

In this section, we discuss LogicWeb's mechanism for accessing such server-side databases. This allows LogicWeb to be utilised with existing databases (and search engines, as briefly discussed in section 3.7), and to be used as a front-end to these facilities.

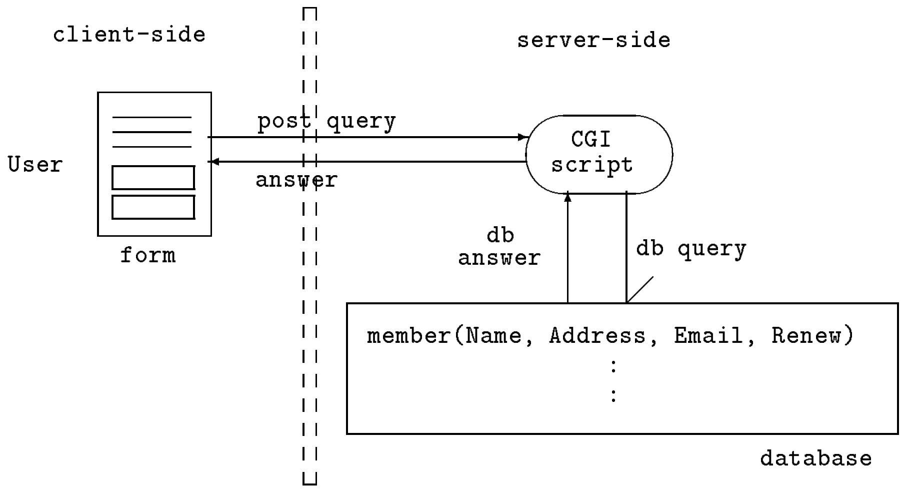

Figure 5: A Server-side Database and its Interface.

A typical server-side database and its interface is represented in Figure 5. 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. We shall assume that the script is located at `http://www.cs.mu.oz.au/cgi-bin/db-query` in the following discussion. The form details are encoded as a POST method using the HTTP protocol [Davison 1995]. Essentially, each field of the form is converted into a string of the type `''field-name=field-value''`. 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 5, the database is assumed to contain Prolog facts of the form:

```
member(Name, Address, Email, Renewal-Date).
```

For example:

```
member(name("Andrew Davison"), address("Univ. of Melbourne"),
       email("ad@cs.mu.oz.au"), renew(november, 1996)).
```

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.

Having outlined a likely server-side database, how can LogicWeb interact with it? It uses a variant of the "#>" operator:

```
m_id(post(URL_of_CGI_script, List_of_Fields))#>Goal.
```

For the scenario outlined above, a possible query would be:

```
m_id( post("http://www.cs.mu.oz.au/cgi-bin/db-query",
              [field("Name", "none"), field("Address", "Univ. of Melbourne"),
               field("Email", "none"), field("Renew", "none")]) )#>
                                 member(Name, _, Email, renew(_, 1997)).
```

The `post/2` term can be viewed as a *specification* of the module against which the `member/4` goal will be evaluated. In this case, the retrieved module will contain all the members from the University of Melbourne, and the goal will extract the name and e-mail address of someone who should renew during 1997 (through backtracking all the Melbourne people in this situation can be collected).

This abstraction moves away from the notion of a POST message being sent to a server-side database, and utilises the familiar LogicWeb model of queries applied to retrieved modules.

Similar mechanisms are also available for specifying modules created using the GET and HEAD methods in the HTTP protocol.

5 Implementation

LogicWeb is implemented using the Common Client Interface (CCI) in the NCSA XMosaic Browser [NCSA 1996]. Figure 6 shows the general structure of the system together with the sequence of steps taken when a user clicks on a hypertext link.

The LogicWeb system has two components: `WWWMain` and a Prolog engine. `WWWMain` is about 400 lines of C, and converts CCI output into a suitable format for the Prolog part, and also creates temporary local files. The Prolog system is mostly written in SWI-Prolog, and is about 1600 lines long. Most of its code is for the LogicWeb meta-interpreter, but there are also utilities for parsing and communicating with the Web. Some of the low-level and/or speed critical features (such as string matching) are coded as C functions with Prolog interfaces.

When the user clicks on a link (step 1), Mosaic gets the page from the Web (steps 2 and 3). The page is not displayed but passed through the CCI to `WWWMain` (step 4). `WWWMain` saves the page to a temporary local file (step 5) and sends a "page downloaded" message via a pipe to the Prolog engine process (step 6). The Web page is read in by the engine (step 7) and converted to a LogicWeb module which is stored inside the meta-interpreter. Often the temporary page is modified to include a forms interface for entering LogicWeb queries (step 8). When any modifications are complete, the URL of the page is sent to Mosaic via the CCI (step 9), and a "done" message is 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 7 illustrates the other main form of user interaction with LogicWeb: the processing of a query.

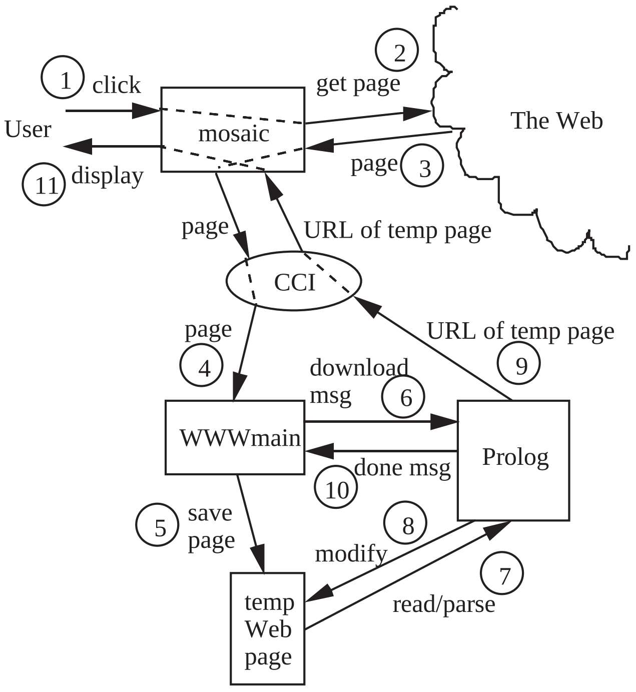

Figure 6: The LogicWeb System and the steps followed after a user clicks on a link.

The query is input via a form (step 1) 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 module 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). At the same time, the Prolog system 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 module 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 module is extracted before the goal is evaluated.

The operational semantics of the meta-interpreter can be specified easily by using a variant of the `demo/2` predicate introduced in [Kowalski 1979; Bowen and Kowalski 1982]. Assuming that `demo/2` has the form:

```
demo(ListofModules, Goals)
```

then the processing of a `m_id(URL)#>Goal` can be described with the code fragment:

```
demo(Ms, [m_id(URL)#>Goal|Gs]) :-
  lookup(URL, Ms, Module),
  apply(Module, Goal),
  demo(Ms, Gs).
demo(Ms, [m_id(URL)#>Goal|Gs]) :-
  web_load(URL, Module),
  apply(Module, Goal),
```

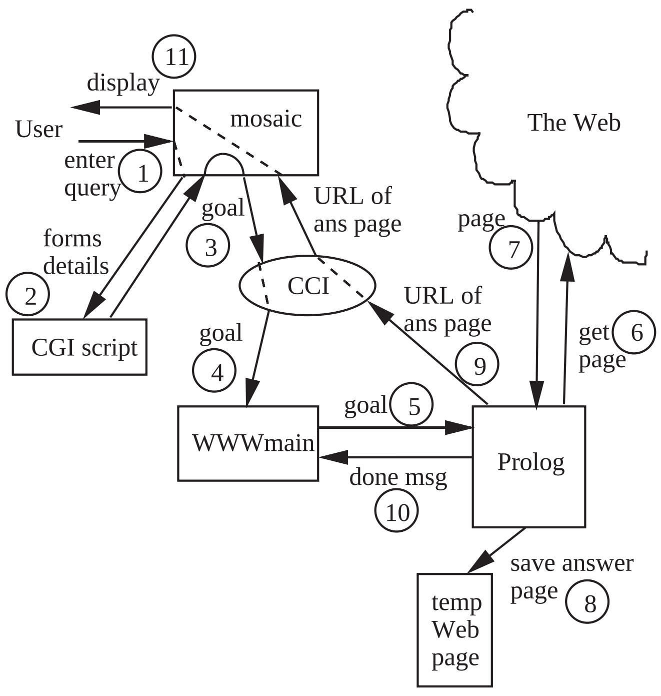

Figure 7: The LogicWeb System and the steps followed after a user enters a query.

```
  demo([Module|Ms], Gs).
```

`lookup/3` is a simple list search predicate which uses the URL (the module's ID) as a search key, and either returns the corresponding module or fails. If it is successful then `Goal` is evaluated against `Module` using `apply/2`, and `demo/2` continues. If `lookup/3` fails, then `web_load/2` in the second clause accesses the Web for the page with the address URL and converts it to a module. Goal is applied to this module, and `demo/2` recurses with the module added to its module list.

At a more abstract level, `m_id(_)#>G` corresponds to the context switch operation found in contextual LP [Monteiro and Porto 1989]. The goal is proved in the specified module regardless of the current context:

$$\frac{M \vdash G}{Ms \vdash m\_id(M)\# > G}$$

It also corresponds to the operation [M]G in [Baldoni et al. 1993], where [M] is a modal operator.

6 Related Work

The Web uses a client-server communications model, and we start by considering LP systems which are client-based (which includes LogicWeb), and then examine server-side solutions. Finally, we briefly describe more expressive Internet-based LP systems, which utilise a peer-to-peer communications model.

6.1 Client-side Systems

WebLog can refer to aspects of Web page structure (the title, links, etc) using LP goals that utilise the page's URL as an identifier [Lakshmanan et al. 1996]. However, WebLog does not treat pages as modules, and pages cannot contain arbitrary LP code, or be composed together.

Several LP libraries allow pages to be downloaded from the Web [Bonnet et al. 1996; Cabeza and Hermenegildo 1996]. These packages also contain tools for parsing the text and extracting information. With these tools, it would be relatively easy to support a fragment of LogicWeb. However, these packages do not contain important browser capabilities, such as being able to display pages, capture clicks on hypertext links, or accept queries from Web forms. Aside from these libraries, any Prolog system with a TCP/IP sockets library, or the ability to invoke a utility like `telnet`, can retrieve pages off the Web.

Java is a popular client-side programming language, and is being used with Prolog in various ways. An interesting interpreter for a subset of Prolog, called W-Prolog, has been written in Java [Winikoff 1996]. Amzi! Prolog has a Java class interface to its Prolog system [Amzi! Prolog 1996]. MINERVA is a compiler for Prolog which generates Java byte-codes [IF Computer 1996], and a similar approach is used in the jProlog addition to BinProlog [Tarau 1996]. Another technique is to link Java to a Prolog engine through its sockets class [Ferguson 1996]. Interestingly, he rejected this approach due to firewall restrictions on non-HTTP traffic.

A drawback with using Java is its restrictive security features. For example, it is quite difficult to store information between sessions since file creation is usually prohibited. Also, Java does not normally allow pages to be downloaded from arbitrary Web sites [Lemay and Perkins 1996].

6.2 Server-side Systems

There are several server-side approaches for using LP in the Web.

As mentioned earlier, CGI is a popular server-side programming interface, which allows information from Web forms to be passed to programs. There are several libraries for writing Prolog programs which can process information from CGI input, and generate suitable replies (typically, new Web pages) [Amzi! Prolog 1996; Cabeza et al. 1996; Cabeza and Hermenegildo 1996; Carpenter 1996; Naish 1995]. The basic idea is captured by Figure 8.

Examples include: WebLS, a tool for building help systems [Sehmi and Kroening 1996], Bob Carpenter's theorem prover (`http://macduff.andrew.cmu.edu/cgparser/`), and Lee Naish's ICLP'97 submissions form (`http://www.cs.mu.oz.au/~lee/iclp97/submitreg.html`).

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. Much of this overhead should be avoidable by the use of shared dynamically linked libraries, and by the utilisation of compilers which generate fast object code and small executables. Also, it is far from clear whether the poor performance of a particular Prolog CGI script is due to its coding in Prolog, or because of network and machine overheads, and/or the slowness of CGI.

A related issue is that 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 LP, this process would be a Prolog engine or logic database. The invoked CGI

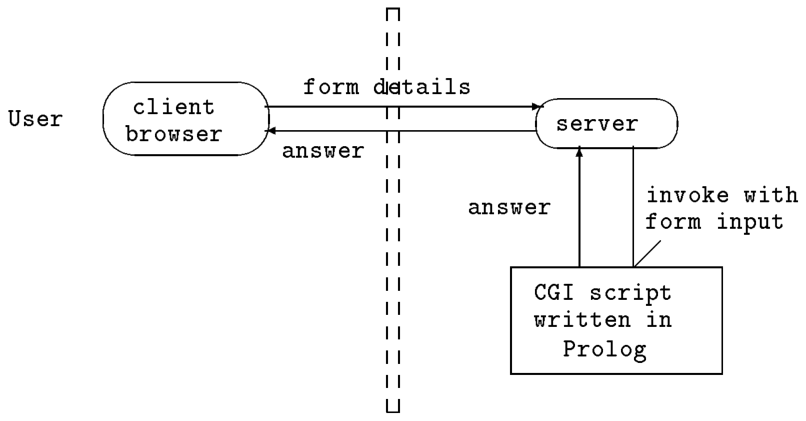

Figure 8: Using Prolog CGI Scripts.

interface scripts communicate with the task process by using sockets. The overall approach is shown in Figure 9.

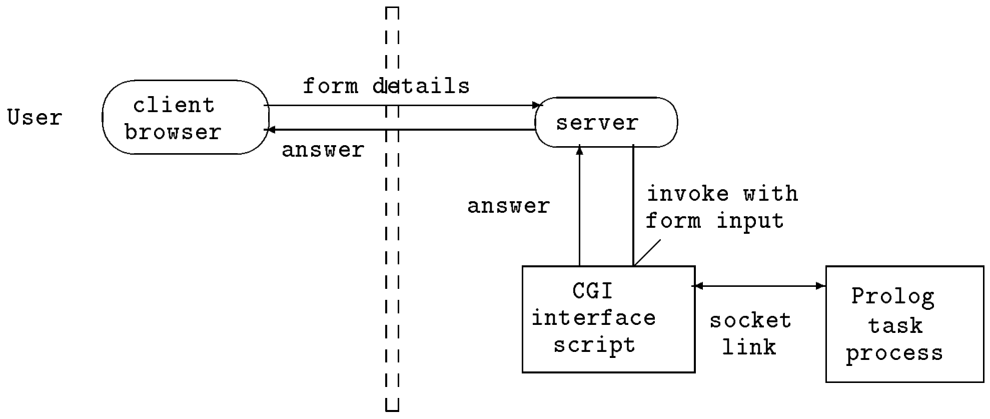

Figure 9: Separating the Interface and Task Processes.

The Announce system uses this technique to implement its electronic calendar of events [Lüttringhaus-Kappel and Schulz 1996]. The task process is coded in ECLiPSe.

This approach is also used in the EMRM knowledge base of medical records, but it utilises the OR-parallel Aurora system to process multiple queries at once [Szeredi at al. 1996].

Don Ferguson has implemented a financial database access system using Quintus Prolog and its TCP library (see `http://edgarscan.tc.pw.com/`). The user interface is a Java applet which communicates with the CGI interface script.

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 [Cabeza et al. 1996]. 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. 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 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 10.

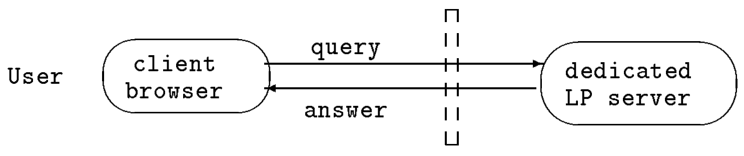

Figure 10: A Dedicated LP Server.

A notable LP solution in this style is the ECLiPSe HTTP server library, which allows a basic server framework to be customized for different communication protocols [Bonnet et al. 1996]. 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) [Frühwirth and Abdennadher 1996].

6.3 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 LP systems utilise the Internet as their underlying communication layer. Two languages in this category are Distributed Oz [Haridi and Van Roy 1996] and April [McCabe and Clark 1995]. Both use message passing and have the ability to move code between machines. April is not strictly speaking a LP language, but has borrowed ideas from LP, and its macro language can be used to support more Prolog-like behaviour [Clark et al. 1996].

SICStus Prolog and its objects package are being used to develop an Internet-based trading application called MarketSpace [Eriksson et al. 1996]. However, the authors note the need for a language that supports richer notions of concurrency, and features such as persistence.

The blackboards in Multi-BinProlog are the basis of LogiMoo, a high-level kernel for Internet collaborative work [Tarau and De Bosschere 1996]. It uses local and virtual blackboards to hide the underlying network. Similar approaches may be possible in other LP languages with Linda-style blackboards, such as SICStus Prolog [SICStus 1996] and &-Prolog/CIAO

[Hermenegildo and Greene 1991]. There is a commercial product along these lines, called Ubique Doors, which initially used FCP [Shapiro 1994]. However, it is now coded in C++.

7 Summary

There are three key advantages of using LP for Web programming.

- LP allows the Web to be viewed more abstractly. LogicWeb encourages Web pages and hypertext links to be reinterpreted as modules and relationships. The practical result is that programmers do not need to concern themselves with lower-level issues like page retrieval and parsing.
- LP is particularly suitable for coding important classes of Web applications. We have currently identified three domains: information structures, search, and parsing. The ease of coding structured information (e.g. as in a database), and manipulating it, is in stark contrast to the effort required to do the same in imperative or object-oriented languages. Simple search applications using backtracking can be coded in a few lines, and more robust versions can utilise heuristics coded with familiar LP techniques. Parsing is essential to most applications, and plentiful techniques are available (e.g. DCGs and other logic grammars).
- LP supports meta-programming. Meta-level reasoning is essential for making a program "intelligent" in the sense that it can reason about its own actions, and respond to changes in its environment (the Web) [Kowalski 1996]. Meta-programming facilitates the manipulation and composition of LogicWeb modules using an approach similar to that described in [Brogi et al. 1994b], except that the modules come from the Web. These techniques are possible because module IDs and Web pages are represented as first class entities in LogicWeb.

  Meta-programming makes interpreters easier to build, offering the potential for LogicWeb modules to store information in domain-specific languages (e.g. as HTML meta tags, VRML).

  Meta-level capabilities are also important for the implementation of LogicWeb's inference engine, and for the specification of its operational semantics. The simplicity of this encoding makes it easier to specify security restrictions.

Future Work. LogicWeb must be able to model the changing nature of the Web, and so we shall be exploring the use of temporal operators to capture the notion of repeatedly updating and differentiating between versions of a module.

Since LogicWeb interprets a click upon a hypertext link as a goal, it is straightforward to enhance the meaning of the link. A large body of work exists in the hypertext literature on various forms of link behaviour, which might be usefully reinterpreted in the context of LogicWeb [CACM 1995].

Our work on lightweight deductive databases can be extended in a number of directions, including support for server-side updates, and the investigation of composition operators such as intersection and inheritance.

Further work is required to make LogicWeb more secure. The meta-interpreter implementation of LogicWeb's inference engine means that it is relatively easy to control which goals are evaluated. A more problematic aspect is resource control, where code should be prevented from going into an infinite loop, and from downloading modules until memory is exhausted. Resource bounded inferencing, strong typing, and partial evaluation may be of use.

LogicWeb's reliance on NCSA XMosaic makes the system difficult to distribute. We may recode the system as a Netscape plug-in to make it more portable.

The operational semantics of LogicWeb has only been partially defined as yet. However, we hope to draw upon the work on the semantics of structured LP [Brogi et al. 1994a; Brogi et al. 1994b; Bugliesi et al. 1994] in order to develop a fuller semantics.

Acknowledgements. We are grateful to Leon Sterling for valuable comments on a previous version of this paper.

References

Amzi! Prolog. 1996. “Internet and Web Tools”, `http://www.amzi.com/internet.htm`.

Baldoni, M., Giordano, L., and Martelli, A. 1993. “A Multimodal Logic to Define Modules in Logic Programming”, In *Proc. of the Int. Symp. on Logic Programming*, D. Miller (ed.), MIT Press, pp.473-487.

Bonnet, Ph., Bressan, S., Leth, L., and Thomsen, B. 1996. “Towards ECLiPSe Agents on the Internet”, In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.1-9. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/eclipse/ea.html`.

Bowen, K. and Kowalski, R.A. 1982. “Amalgamating Language and Meta-language”, In *Logic Programming*, K.L. Clark and S. Tarnlund (eds.), Academic Press, pp.153-172.

Brogi, A., Mancarella, P., Pedreschi, D., amd Tutini, F. 1994a. “Modular Logic Programming”, *ACM Trans. on Programming Languages and Systems*, Vol. 16, No. 4, pp.1361-1398.

Brogi, A., Renso, C., and Turini, F. 1994b. “Amalgamating Language and Meta-Language for Composing Logic Programs”, In *Proc. of GULP-PRODE 94 Joint Conference on Declarative Programming*, Peniscola, Spain.
Also at `ftp://ftp.di.unipi.it/papers/turini/PRODE94.ps.gz`.

Bugliesi, M., Lamma, E., and Mello, P. 1994. “Modularity in Logic Programming”, *Journal of Logic Programming*, Vol. 19 & 20, May, pp.443-502.

Cabeza, D. and Hermenegildo, M. 1996. “`html.pl`: An HTML Package for (C)LP Systems”, Spain, March. Available from `http://www.clip.dia.fi.upm.es/miscdocs/`.

Cabeza, D., Hermenegildo, M., and Varma, S. 1996. “The PiLLoW/CIAO Library for INTERNET/WWW Programming”, In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.43-62. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/pillow/lpnet3.html`.

CACM 1995. Special Issue on “Designing Hypermedia Applications”, *Comm. of the ACM*, Vol. 38, No. 8, August.

Carpenter, B. 1996. “A Prolog-Based CGI Handler”,
`http://macduff.andrew.cmu.edu/cgparser/prolog_cgi.html`.

Clark, K.L., Skarmeas, N., and McCabe, F.G. 1996. “Agents as Clonable Objects with Knowledge-base State”, In *Proc. of the 2nd Int. Conf. on MultiAgent Systems.* Also at `http://www-lp.doc.ic.ac.uk/~klc/mob_agents.html`.

Connor, R. 1996. “An Overview of the Aims of the Hippo Project”. Available from `http://grappa.dcs.st-and.ac.uk/HIPPO/overview.ps`.

Davison, A. 1995. "Programming with HTML Forms", *Dr. Dobb's Journal*, Vol. 20, No. 6, June, p.70-75.

Eriksson, J., Finne, N., and Janson, S. 1996. "Information and Interaction in MarketSpace and their Implementation in Prolog", In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.125-137. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/ms/marketspace.html`.

Ferguson, D. 1996. "Linking a Prolog Program into an HTTPD", Posting to `comp.lang.prolog`, November 7th.

Frühwirth, T. and Abdennadher, S. 1996. "The Munich Rent Advisor", In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.11-27. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/lpnet5/lpnet5.html`.

Han, Y., Loke, S.W., and Sterling, L. 1996. "Agents for Citation Finding on the World Wide Web", Dept. of Computer Science, Univ. of Melbourne, Tech. Report 96/40. Available from `http://www.cs.mu.oz.au/tr_db/mu_96_40.ps.gz`.

Haridi, S., and Van Roy, P. 1996. "An Overview of the Design of Distributed Oz", In *Proc. of the Multi-Paradigm Logic Programming Workshop*, M.M.T Chakravarty, Y. Guo, and Y. Ida (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.13-24.

Hermenegildo, M. and Greene, K. 1991. "The &-Prolog System: Exploiting Independent And-Parallelism", *New Generation Computing*, Vol. 9, No. 3,4, pp.233-257.

Hodgson, J. 1996. "Programming Language Prolog Part 2, Modules", Committee Draft, May 31st. Available from `http://www.sju.edu/~jhodgson/x3j17.html`.

HTML 3.2 Reference Specification. 1996. W3C Proposed Recommendation. `http://www.w3.org/pub/WWW/TR/PR-html32-961105`.

IF Computer. 1996. `MINERVA` Documentation, `http://www.ifcomputer.com/StrategicWeb/MINERVA/home_en.html`.

Kowalski, R.A. 1979. *Logic for Problem Solving*, Elsevier, New York.

Kowalski, R.A. 1996. "Using Meta-Logic to Reconcile Reactive with Rational Agents", *PAAM 96: The Practical Applications of Intelligent Agents and Multi-Agent Technology*, London, April, pp.361-374.

Lakshmanan, L.V.S., Sadri, F., and Subramanian, I.N. 1996. "A Declarative Approach to Querying and Restructuring the World-Wide-Web", *Post-ICDE Workshop on Research Issues in Data Engineering (RIDE'96)*, New Orleans, February. Available as `ftp://ftp.cs.concordia.ca/pub/laks/papers/ride96.ps.gz`

Lemay, L. and Perkins, C. 1996. *Teach Yourself Java in 21 Days*, Sams.net Publishing.

Loke, S.W. and Davison, A. 1996. "Logic Programming with the World-Wide Web", In *Proc. of the 7th. ACM Conf. on Hypertext*, ACM Press, March, pp.235-245.

Loke, S.W., Davison, A., Sterling, L. 1996a. "CiFi: An Intelligent Agent for Citation Finding on the World Wide Web", *PRICAI'96: 4th Pacific Rim Int. Conf. on Artificial Intelligence*, Cairns, Australia, August.

Loke, S.W., Davison, A., and Sterling, L. 1996b. "Lightweight Deductive Databases on the World-Wide Web", In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.91-106. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/lwddbs/lwddbs.html`.

Luke, S., Spector, L., and Rager, D. 1996. "Ontology-Based Knowledge Discovery on the World Wide Web", In *Proc. of the Workshop on Internet-based Information Systems, AAAI-*

96, Portland, Oregon, USA.
Also at `http://www.cs.umd.edu/projects/plus/SHOE/aaai-paper.html`.

Lüttringhaus-Kappel, S. and Schulz, D. 1996. "A Calendar of Events – Architecture and Experiences" In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.29-41. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/announce/announce.html`.

McCabe, F.G. and Clark, K.L. 1995. "April – Agent Process Interaction Language", In *Intelligent Agents*, M. Wolldridge and N. Jennings (eds.), LNAI, Vol. 890, Springer-Verlag.

Monteiro, L. and Porto, A. 1989, "Contextual Logic Programming", In *Proc. of the 6th Int. Conf. on Logic Programming*, G. Levi and M. Martelli (eds.), Lisbon, Portugal, The MIT Press, pp. 284-299.

Naish, L. 1995. "HTML Forms Interface to NU-Prolog",
`http://www.cs.mu.oz.au/~lee/src/forms/index.html`.

NCSA. 1996. `NCSA XMosaic` and `CCI` Documentation,
`http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/` and
`http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-spec.html`.

Reynolds, M.C. and Wooldridge, A. 1996. *Special Edition Using JavaScript*, QUE.

Sehmi, A. and Kroening, M. 1996. "WebLS: A Custom Prolog Rule Engine for Providing Web-Based Tech Support", In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.107-123. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/amzi/lspap.html`.

Shapiro, E. 1994. "Enhancing the WWW with Co-Presence", In *Proc. of the 2nd Int. Conf. on the WWW*.

SICStus. 1996. `SICStus Prolog` Documentation, `http://www.sics.se/sicstus.html`.

Szeredi, P., Molnár, K., and Scott, R. 1996. "Serving Multiple HTML Clients from a Prolog Application", In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.81-90. Also at `http://www.cs.mu.oz.au/~ad/lp-internet/iqsoft/multiple.html`.

Tarau, P. 1996. `BinProlog 5.25` Documentation, System available from
`http://clement.info.umoncton.ca/~tarau`.

Tarau, P. and De Bosschere, K. 1996. Virtual World Brokerage with BinProlog and Netscape", In *Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications*, P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo (eds.), JICSLP'96: Joint Int. Conf. on Logic Programming, Bonn, Germany, September, pp.63-80. Also at
`http://www.cs.mu.oz.au/~ad/lp-internet/lpnet10/art.html`.

Winikoff, M. 1996. `W-Prolog 1.0` Documentation, System available from
`http://www.cs.mu.oz.au/~winikoff/wp`.