A Concurrent Logic Programming Model of the Web

Technical Report TR1998/1
December, 1998

Abstract

We propose a concurrent logic programming model for the Web which maps Web page retrieval to logic programming processes and data streams. This gives us the leverage to address, in logic programming notation, Web connectivity issues such as latency and unreliability, and to encode recovery/avoidance behaviours such as time-outs, rate monitoring, and repeat requests.

We illustrate how this approach can be used to support the more abstract LogicWeb view of the Web as compositional logic programs. One benefit of the underlying concurrency is that a 'concurrent' LogicWeb can utilise AND- and OR- parallelism for search and other decision procedures.

We describe the design and implementation of the main components of our work. Examples are coded in Parlog, although most concurrent logic programming languages should be able to support this Web model.

Authors:

Andrew Davison
Department of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112, Thailand
ad@ratree.psu.ac.th

Seng Wai Loke
Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Victoria 3052, Australia
swloke@cs.mu.oz.au
http://www.cs.mu.oz.au/~swloke

A Concurrent Logic Programming Model of the Web

Andrew Davison

CSIM
Asian Institute of Technology
Pathumthani 12120, Thailand
E-mail: ad@cs.ait.ac.th

Seng Wai Loke

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

Abstract

We propose a concurrent logic programming model for the Web which maps Web page retrieval to logic programming processes and data streams. This gives us the leverage to address, in logic programming notation, Web connectivity issues such as latency and unreliability, and to encode recovery/avoidance behaviours such as time-outs, rate monitoring, and repeat requests.

This approach can be used to support the more abstract LogicWeb view of the Web as compositional logic programs. One benefit of the underlying concurrency is that a 'concurrent' LogicWeb can utilise AND- and OR- parallelism for search and other decision procedures.

We describe the design and implementation of the main components of our work. Examples are coded in Parlog, although most concurrent logic programming languages should be able to support this Web model.

1 Introduction

Different programming models for the Web impose different computational abstractions upon it: client-server, distributed objects, global hypertext, and so on. For example, our LogicWeb model considers the Web to be an open collection of logic programs which can be composed together to form new entities [10]. As with all abstractions, there are advantages and disadvantages. On the plus side is the high-level representation which emphasises structured data and logical relationships. Ironically, this viewpoint can also be considered a disadvantage since it hides the very real concerns of network latency, bandwidth, and the Web's familiar unreliability.

We propose a new model for the Web which represents the communication between a client and a server as a stream of data passing between Logic Programming (LP) processes. Essentially, we are applying the concurrent LP computational paradigm to the Web, thereby allowing issues related to Web data flow to be captured at the program level. For example, it is possible to write code that responds to different kinds of download failure (e.g. server unavailability, network outage), and exhibits various retrieval behaviours (e.g. time-outs,

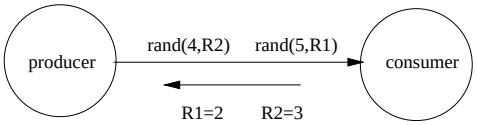

Figure 1: Producer/Consumer Example.

retries). In addition, the inherent concurrency of the formalism permits the implementation of search techniques that employ AND- and OR- parallelism.

This lower-level mapping of Web connectivity to LP processes and streams can also be utilised as the basis of more abstract Web metaphors. In particular, we explain how a 'concurrent' LogicWeb approach can be represented.

In section 2, we consider some of the important notions of concurrent LP, and the details of a specific concurrent LP language, Parlog, which we use for our examples. Section 3 introduces a new built-in, `download/4`. which is the implementation cornerstone of our model. Section 4 uses `download/4` in Parlog to code various forms of parallel Web retrieval, time-out, and retry behaviours. Section 5 discusses how LogicWeb can be supported with the help of `download/4`. Section 6 examines related work, and section 7 concludes.

The C code component of the `download/4` implementation is given in the Appendix.

2 Concurrent LP

The concurrent LP paradigm adds stream AND-parallelism, OR-parallelism, and don't care nondeterminism to logic programming. Arguably the result has little to do with the Herbrand view of computation, being more suited to representing processes and the data flows between them. Shapiro characterises the distinction as transformational systems (i.e. sequential LP) versus reactive systems (i.e. concurrent LP) [13].

Parlog is a typical concurrent LP language [4, 3] with some notable features for making programs more succinct – namely, deep guards, sequential operators, and modes on predicate arguments. The following examples use Parlog but our Web model is suitable for all the concurrent LP family of languages.

We will illustrate Parlog with a producer/consumer example. The producer process sends terms of the form `rand(No,Res)` to the consumer. `No` is an integer supplied by the producer, but `Res` will be bound by the consumer to a random integer between 0 and `No-1`. This binding is transmitted 'back' to the producer through unification, thereby utilising it as a communication mechanism. Figure 1 shows the configuration of the two processes.

The consumer is coded as:

```
mode consumer(?).
consumer([]).
consumer([rand(No,Res)|InStream]) :-
  Res is random(No),
  consumer(InStream).
```

The mode definition states that the consumer has one input argument (?). The argument is a list (employed here as an input stream) which triggers the second clause of `consumer/1` as it is incrementally bound. The consumer binds `Res` using the built-in `random/1` and then recurses. When the list is terminated (i.e. the stream is closed), the process terminates by not recursing. The ',' operator in the second clause means AND-parallel conjunction, so allowing the consumer to potentially process many input terms (messages) at once.

The producer is coded as:

```
mode producer(?, ^).
producer(0, []).
producer(No, [rand(No,Res)|OutStream]) :-
  No > 0 :
  No1 is No - 1,
  delay_write(Res),
  producer(No1, OutStream).

mode delay_write(?).
delay_write(Res) :-
  bound(Res) : write(Res) & nl.
```

The producer is called with a positive integer as its first argument. When the value is 0, the first clause closes the output list (output stream). The '^' in the mode declaration means that the second argument of `producer/2` is for output.

The second clause deals with the case when `No` is greater than 0 by checking the value in a guard test (the goal before the ':'). Only if the guard evaluates to true is the clause chosen. The execution is committed to a clause once the guard has been evaluated, and so failure somewhere in the body will cause the entire program to fail.

`delay_write/2` is called in AND-parallel with the recursive call to `producer/2` but will delay until its guard call evaluates to true. `bound/1` only succeeds when `Res` is bound and then its value is printed followed by a newline ('`&`' is sequential conjunction).

The process configuration shown in Figure 1 is created with the query:

```
?- producer(5, Str), consumer(Str).
```

The shared variable `Str` sets up the stream link between the processes, and both processes are started in AND-parallel due to the ',' conjunction. The producer will send five messages to the consumer, and receive five replies.

A common predicate for building more complex networks is `merge/3`:

```
mode merge(?, ?, ^).
merge([El|X], Y, [El|Z]) :-
  merge(X, Y, Z).
merge(X, [El|Y], [El|Z]) :-
  merge(X, Y, Z).
merge([], Y, Y).
merge(X, [], X).
```

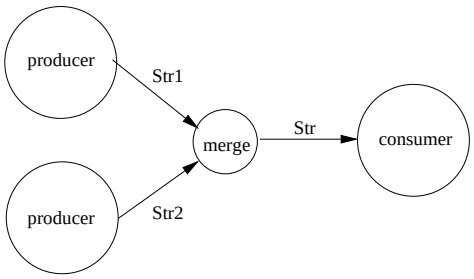

Figure 2: Two Producers and One Consumer.

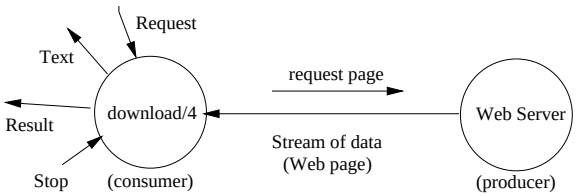

Figure 3: Downloading a Web Page.

`merge/3` combines two input streams (supplied as its first and second arguments) into a single output stream (its third argument). `merge/3` could be utilised to connect two producer processes to a consumer:

```
?- producer(10, Str1), producer(20, Str2),
   merge(Str1, Str2, Str),
   consumer(Str).
```

This configuration is shown in Figure 2.

`merge/3` utilises don't care nondeterminism when it has messages pending on both its input streams, since it can use either its first or second clause.

`merge/3` deals with producer termination with its third and fourth clauses. If one of the producers finishes then the other producer's output stream is linked directly to the consumer; in effect Figure 2 is reconfigured to look like Figure 1.

The preceding description demonstrates how concurrent LP encourages programs to be viewed as networks of processes connected by streams of data. Extensions of these basic techniques allow various other forms of interaction, such as one-to-many, broadcast, and blackboard-based communication.

3 `download/4`

Our model utilises the concurrent LP producer/consumer and stream viewpoint to represent Web page retrievals. The 'consumer' is encoded by a concurrent LP predicate `download/4`, and the 'producer' is the particular Web server. The correspondence is not entirely direct since the consumer initiates the contact with the producer, but thereafter it receives the Web page as a stream of characters. The basic situation is shown in Figure 3.

In Parlog, `download/4`'s mode declaration would be:

```
mode download(Request?, Text^, Result^, Stop?).
```

Parlog allows optional variable names to precede the '?' and '^' symbols.

`Request` is a term representing the required HTTP protocol (e.g. GET, POST [1]) and the URL of the page. For example:

```
req(get, 'http://www.cs.ait.ac.th/~ad')
```

means retrieve Andrew Davison's home page.

`Text` will output a stream of ASCII codes making up the retrieved page. `Text` may not be bound if there is an error during the request, or may return only part of the page if an error occurs during the download.

`Result` will be bound to `ok` if the download finishes successfully or `err(Message)`, where `Message` can be a variety of error values.

When `Stop` is bound by the user during a download, the retrieval will be terminated.

`download/4` can be more fully understood by considering the possible steps in its evaluation, and the bindings that its arguments have during those steps.

After `Request` is bound, `download/4` will attempt to open a connection with the page's Web server. This may result in an error, and `Result` will be bound to an `err/2` term. Several kinds of error are possible: the request's URL may not be well-formed (`err(bad_url)`), there was a problem with socket creation (`err(socket)`), DNS lookup of the server failed (`err(dns)`), or a connection could not be established with the server (`err(connect)`). After the particular error has been output in `Result`, `download/4` terminates. Alternatively, `download/4` may successfully contact the server, and the `Text` output stream will start being partially instantiated. This incremental instantiation mimics the character of the underlying network as data is read in chunks from the TCP/IP link to the server.

Some time later, one of two possible events will occur: there will either be a break in the connection, causing the output stream to close (i.e. the list is terminated with `[]`) and `Result` is bound to `err(connection_lost)`. Alternatively, the page wil be fully downloaded and the `Text` list will be terminated with `[]`, but `Result` will be bound to `ok`.

An important component of `download/4` is the ability for another process (or the user) to bind its `Stop` variable to `stop`. This may occur at any time after the initial call, and breaks the network connection from the client's end. Our implementation makes a simplifying assumption that a `Stop` binding will only occur after a link is established (i.e. after `Text` starts being bound). When `download/4` receives a `Stop` binding, the `Text` stream is closed and `Result` is bound to `err(stopped)`.

There are opportunities for race conditions when `download/4` has to choose between a server-initiated termination (connection loss or download completion) and the client's stop request.

3.1 Implementation Outline

Our prototype implementation is separated into two parts – the majority of the functionality is coded in C (in `isdown.c`), utilising the fork and signal features of UNIX, and a thin layer coded in the concurrent LP language (in `download/4`). We require that the language be able to spawn a UNIX process

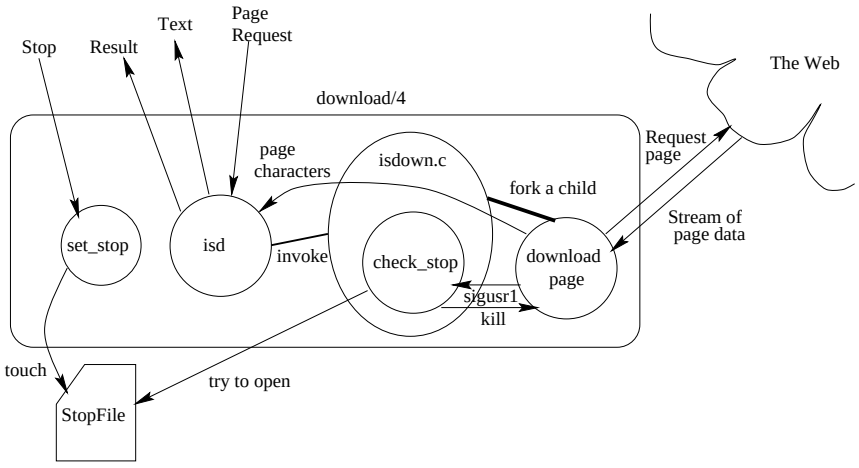

Figure 4: Implementation Details.

and incrementally read its output as an input stream. The implementation is illustrated in Figure 4.

`isdown.c` is called with two command line arguments:

```
isdown URL StopFile
```

`isdown` forks a child process to download the page at the address `URL` (at the moment, the code only supports the HTTP GET protocol). Meanwhile, the parent process periodically attempts to open the file called `StopFile`. If it ever succeeds, then the child process is killed, thereby stopping the download.

During this time, if the child finishes its retrieval, it terminates after sending a signal to the parent. This signal informs the parent to stop trying to open `StopFile`, and to exit.

As the child incrementally receives the Web page, the text is directed to standard output. An error will cause the retrieval to stop, and the child passes on the error by placing the string `"#### error-no"` onto standard output (`error-no` will have a value between 1 and 6). We assume the string is sufficiently unique to distinguish it from the preceding Web page text. If the download is successfully completed then `"#### 0"` is appended to the output.

The complete code for `isdown.c` appears in the Appendix.

The concurrent LP part of the implementation begins with `download/4`:

```
mode download(?, ^, ^, ?).
download(req(get,URL), Text, Result, Stop) :-
  make_fnm(Fnm),
  isd(URL, Fnm, Text, Result),
  set_stop(Stop, Result, Fnm).
```

It calls `make_fnm/1` to create a unique filename, which is used by `isd/4` and `set_stop/3` as the name of the stop file. `isd/4` invokes `isdown.c` and reads its output incrementally, while `set_stop/3` waits for either the `Stop` or `Result` variables to be bound.

Details of `isd/4`:

```
mode isd(?, ?, ^, ^).
isd(URL, StopFile, Text, Result) :-
  concat_atom(['isdown ', URL, " ', StopFile], Cmd),
  open(pipe(Cmd), read, Str),
  get0(Str, Ch),
  get_chars(Str, Ch, Download),
  get_result(Download, Text, Result).

mode get_chars(?, ?, ^).
get_chars(_, -1, []).
get_chars(Str, Ch, [Ch|Dld]) :-
  Ch \== -1 :
  get0(Str, NCh),
  get_chars(Str, NCh, Dld).

mode get_result(?, ^, ^).
get_result([35,35,35,35,32,Num|_], [], Result) :-
  ResNo is Num-48,        % 35 = '#'; 48 = '0'
  num_mesg(ResNo, Result).
get_result([Ch|Dld], [Ch|Text], Result) :-
  Ch \== 35 :
  get_result(Dld, Text, Result).

mode num_mesg(?, ^).
num_mesg(0, ok).
num_mesg(1, err(socket)).
num_mesg(2, err(dns)).
num_mesg(3, err(connect)).
num_mesg(4, err(bad_url)).
num_mesg(5, err(stopped)).
num_mesg(6, err(connection_lost)).
```

`isd/4` builds the `isdown` command string with the built-in `concat_atom/2` and invokes it as a process using `open/3`. Unfortunately, this latter feature is not available in any concurrent LP language we examined, and so we were forced to test this code in SWI-Prolog, version 2.1.0 [14].

`get_chars/3` incrementally reads in `isdown`'s output and passes it to `get_result/3` which pulls off the terminating result value and converts it to a more informative `Result` term.

`set_stop/3` is:

```
mode set_stop(?, ?, ?).
set_stop(_, ok, _).
set_stop(_, err(_), _).
set_stop(stop, _, StopFile) :-
  concat_atom(['touch ', StopFile], Cmd),
  shell(Cmd).
```

`set_stop/3` suspends until its first or second argument is bound. If its first argument is bound (which is the `Stop` variable) then it creates the stop file. If its second argument is bound (which is the `Result` variable from `isd/4`) then `isd/4` has finished and `set_stop/3` should also terminate.

4 Modelling Web Interactions

We consider how `download/4` can be used to build various Web interaction behaviours.

The AND-parallel download of two pages is:

```
?- download(req(get,'http://www.cs.ait.ac.th/~ad'), T1, R1, _),
   download(req(get,'http://www.cs.mu.oz.au/~swloke'), T2, R2, _).
```

Often it is useful to specify a download that tries several alternative sites (for example when searching). A predicate for OR-parallel search is:

```
mode or_get(?, ^).
or_get([URL|Ds], T) :-
  download(req(get,URL), Text, ok, _) : T=Text.
or_get([_|Ds], T) :-
  or_get(Ds, Text) : T=Text.
```

`or_get/2` uses an important feature of Parlog – the deep guard, which is a guard containing user-defined predicates. `or_get/2`'s behaviour is to call `download/4` in OR-parallel for each URL. When any one of the guarded downloads is successful then the other guard evaluations will be terminated automatically. This coding technique can be rephrased using only AND-parallelism, which is necessary for languages with only flat guards.

An example call to `or_get/2` to try downloading Andrew Davison's page from two different sites:

```
?- or_get(['http://fivedots.coe.psu.ac.th/~ad',
           'http://www.cs.ait.ac.th/~ad'], Text).
```

It is frequently useful to limit the amount of time that a download should take, especially when the network is very loaded. This mechanism could be used with predicates like `or_get/2` to try alternative URLs if the current download is too slow.

A definition for a retrieval predicate with a time-out facility:

```
mode timeout(?, ?, ^, ^, ?).
timeout(_Time, Request, Text, Result, Stop) :-
  download(Request, Text, Result, Stop) : true.
timeout(Time, _, _, err(timeout), _) :-
  sleep(Time) : true.
```

Deep guards are again utilised to set up an OR-parallel evaluation, this time of `download/4` and `sleep/1` (a built-in which succeeds after suspending for a specified number of seconds). If the time-out expires before `download/4` has finished then `download/4` is terminated and `Result` is bound to `err(timeout)`.

Web users do not give up easily, and will often reattempt a download if it fails the first time (or even several times). `repeat/5` repeatedly calls `download/4` up to a specified number of times until the retrieval is successful or the limit is reached.

```
mode repeat(?, ?, ^, ^, ?).
repeat(Limit, Request, Text, Result, Stop) :-
  repeat1(0, Limit, Request, Text, Result, Stop).

mode repeat1(?, ?, ?, ^, ^, ?).
repeat1(Limit, Limit, _, _, err(limit), _).
repeat1(Count, Limit, Request, Text, ok, Stop) :-
  Count < Limit,
  download(Request, Text, ok, Stop) : true;
repeat1(Count, Limit, Request, Text, Result, Stop) :-
  Count < Limit :
  Count1 is Count + 1,
  repeat1(Count1, Limit, Request, Text, Result, Stop).
```

`repeat1/6` uses the sequential-OR operator (';') between its second and third clauses so that the download is tried first. If nothing is obtained then the third clause recurses after incrementing the count argument.

Another Web problem is dealing with slow downloads: a common behaviour is to 'give up' on a retrieval when its arrival rate drops below some acceptable value, and then perhaps switch to another site.

`rate/5` monitors the arrival rate for a page and terminates the download if the speed drops below the specified minimum.

```
mode rate(?, ?, ^, ^, ?).
rate(Minimum, Request, Text, Result, Stop) :-
  download(Request, Text, R1, S1),
  time(Time),
  rate_mon(Text, Minimum, Time, 0, R2, S2),
  combine(R1, S1, R2, S2, Result, Stop).

mode rate_mon(?, ?, ?, ?, ^, ?).
rate_mon([], _, _, _, done, _).       % download done
rate_mon(_, _, _, _, stopped, stop).  % download stopped
rate_mon(Text, Min, Time, Len, R, S) :-
  calc_len(Text, Text1, Len, Len1),
  test_len(Text1, Min, Time, Len1, R, S).

mode calc_len(?, ^, ?, ^).
calc_len([], [], Len, Len).       % end of text
calc_len(Var, Var, Len, Len) :-   % current end of text
  var(Var) : true.
calc_len([Ch|Chs], Var, Len, Len2) :-
  Len1 is Len + 1,
  calc_len(Chs, Var, Len1, Len2).

mode test_len(?, ?, ?, ?, ^, ?).
test_len(_Text, Min, Time, Len, err(too_slow), _) :-
  time(NowTime),
  Rate is Len/(NowTime-Time),
  Rate < Min : true;              % rate too slow
```

```
test_len(Text, Min, Time, Len, R, S) :-
  rate_mon(Text, Min, Time, Len, R, S).

mode combine(R1?, S1^, R2?, S2^, Result^, Stop?).
combine(_, S1, err(too_slow), _, Result, _) :-   % rate error
  S1 = stop, Result = err(too_slow).
combine(R1, _, _, S2, Result, _) :-       % download result
  ground(R1) :
  S2 = stop, Result = R1.
combine(_, S1, _, S2, Result, stop) :-    % stop from outside
  S1 = stop, S2 = stop, Result = err(stopped).
```

`rate_mon/6`'s usual behaviour is to calculate the arrival rate by calling `calc_len/4` to get the length of the text already downloaded, and then use `test_len/6` to check if the rate has dropped below the minimum permitted. However, `rate_mon/6` can also be terminated when the download has finished or been stopped.

`calc_len/4` uses the fact that the retrieved text is represented as a partially instantiated list ending in a variable or `[]`. It recurses down to the current end of the list counting the characters it sees. This number is added to the previous length of the text to obtain the current length.

`combine/6` is a standard predicate in concurrent LP programs for monitoring the `Stop` and `Result` parameters of two processes (`download/4` and `rate_mon/6` in this case). If one of the processes produces a result then `combine/6` stops the other process. It also passes the final result to `rate/5` inside its `Result` output, and monitors `rate/5`'s `Stop` variable.

The underlying approach in our examples was to develop a new predicate for each kind of interaction: `timeout/5` for time-outs, `repeat/5` for repeated retries, `rate/5` for data transfer rate measurement, and so on.

It is easy to combine these interaction behaviours. For example, a predicate employing rate monitoring *and* a time-out would essentially be the same as `rate/5` but with a call to `timeout/5` instead of `download/4`.

5 Concurrent LogicWeb

As mentioned above, LogicWeb views the Web as a collection of logic programs which can be composed together using operators such as union, intersection, and encapsulation [10]. This programming style makes it much easier to implement structured data representations on top of the Web, including light-weight databases and concept nets [7, 9]. The Web link mechanism can be augmented with logical relationships, which are useful when programming search engines or Web guided tours [6, 8].

One of the key components of LogicWeb is the *context operator*:

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

This executes `Goal` against the logic program specified by the Web request method and the URL. Low-level issues such as page retrieval, parsing, and conversion into logic program format are hidden. In addition, if the program

required by `Goal` is already present on the client-side (because it was previously downloaded) then it is not retrieved again.

This abstraction away from network issues, such as latency, bandwidth, and connection failure, is very useful for many kinds of program. However, it also makes some types of code harder to write. For example, search agents would often like to be able to identify why a page cannot be retrieved.

A `#>` goal can fail, so it is possible to distinguish download failure from goal failure by executing:

```
lw(RequestMethod, URL)#>true.
```

but this still does not supply any information about the kind of download failure, such as a bad URL or connection loss.

A more informative context operator can be implemented using `download/4`. The `#>` call becomes:

```
lw(RequestMethod?, URL?, Store?, NewStore^, Result^)#>Goal
```

`Store` is the LogicWeb program store which holds the programs which have been previously obtained. If a further download is required for the evaluation of `Goal` then the store will be extended with the new program, and output in `NewStore`. If no download is necessary, or the retrieval fails, then `NewStore` takes on `Store`'s value. `Result` returns `download/4`'s result information.

The reason for the `Store`/`NewStore` pair is the presence of AND-parallelism, which means that two (or more) `#>` goals could be evaluated concurrently, with the resulting problem of reconciling concurrent changes to the program store. We have chosen to sequentialise the updates by requiring the store to be passed explicitly between `#>` goals. Therefore, the parallel execution of goals in two different programs (pages) would be:

```
?- lw(get,'http://www.cs.ait.ac.th/~ad',
                        St, St1, R1)#>interests(AIs),
   lw(get,'http://www.cs.mu.oz.au/~swloke',
                        St1, St2, R2)#>interests(SIs),
   intersect(AIs, SIs, Is).
```

The query collects the interests from the two programs and calculates their intersection. Since the two pages are independent, the two downloads could be carried out in AND-parallel, as the ',' conjunction specifies. However, the sharing of the program store between the `#>` goals sequentialises them.

The implementation of the concurrent context operator:

```
mode #>(?, ?).
lw(Method, URL, St, NSt, Result)#>Goal :-
  download_prog( req(Method,URL), St, NSt, Result),
  try_goal(Result, NSt, lw(Method,URL), Goal).

mode download_prog(?, ?, ^, ^).
download_prog(Request, St, St, ok) :-
  member( prog(Request,_), St) : true;      % already in store
download_prog(Request, St, NSt, Result) :-
  download(Request, Text, Result, _Stop),   % stop not used
```

```
  extract_prog(Result, Request, Text, St, NSt).

mode extract_prog(?, ?, ?, ?, ^).
extract_prog(err(_), _, _, St, St).
extract_prog(ok, Request, Text, St, [prog(Request,Clauses)|St]) :-
  make_prog(Text, Clauses).   % convert text into LP clauses

mode try_goal(?, ?, ?, ?).
try_goal(err(_), _, _, _).
try_goal(ok, St, ProgID, Goal) :-
  demo(St, ProgID, Goal).
```

The `#>` predicate tries to download the program required for `Goal` by using `download_prog/4`. If the retrieval is error-free then the goal is evaluated by `demo/3`, otherwise the goal is ignored.

`download_prog/4` starts by checking if the store already contains the necessary program by searching for the term `prog(Request,_)`. If it is present then a new download is unnecessary, otherwise one is carried out. If the retrieval is successful, `extract_prog/5` converts the resulting text into logic program clauses and stores them.

One benefit of coding `#>` explicitly is that it makes the design decisions clearer. For instance, if a program has already been downloaded then it is never retrieved again, and there is no way of stopping a `#>` download. Of course, these decisions can be modified.

5.1 Parallel Search

The utilisation of the LogicWeb model in a concurrent LP framework means that parallel algorithms can be used to search and analyse Web pages.

In the following example, we assume that the logic program generated for a Web page contains the fact `links/1`, which holds a list of all the link anchor and URL pairs on the page:

```
  links([ link(Anchor1,URL1), ...]).
```

This differs from the sequential LogicWeb translation where each link anchor and URL pair is stored in its own fact. A list representation permits parallel search to be programmed more directly.

`search_pages/4` uses deep guards to search in OR-parallel starting from a given URL until a page is found which contains the specified phrase. A typical query would be:

```
?- search_pages('http://www.cs.ait.ac.th/~ad',
                                          Store, "Parlog", URL).
```

`Store` holds the LogicWeb program store.

`search_pages/4` is defined as:

```
mode search_pages(?, ?, ?, ^).
search_pages(URL, Store, Phrase, Address) :-
  relevant_page(URL, Store, Phrase) : Address = URL.
search_pages(URL, Store, Phrase, Address) :-
```

```
  lw(get, URL, Store, NStore, ok)#>links(Ls),
  visit_links(Ls, NStore, Phrase, A) : Address = A.

mode relevant_page(?, ?, ?).
relevant_page(URL, St, Phrase) :-
  lw(get, URL, St, _, ok)#>h_text(Text),
  contains(Text, Phrase).

mode visit_links(?, ?, ?, ^).
visit_links([link(_,URL)|_], St, Phrase, Addr) :-
  search_pages(URL, St, Phrase, A) : Addr = A.
visit_links([_|Ls], St, Phrase, Addr) :-
  visit_links(Ls, St, Phrase, A) : Addr = A.
```

`relevant_page/3` uses a call to `h_text/1`, a predicate generated by LogicWeb for every downloaded page to hold the text of that page. `relevant_page/3` also uses the built-in `contains/2` to check if `Phrase` is present in the page's text.

One useful feature of `search_pages/4` is that the updates to the program store are not retained at the end of the search. Only the URL of the matching page is returned.

A poor aspect of `search_pages/4` and its subsidiary predicates is the lack of loop checking – it is quite likely that many of the OR-parallel searches will revisit pages already seen. This could be avoided by passing around a list of visited URLs but this would linearise the code to a large degree.

5.2 Atomic Test-and-Set

The use of explicit store variables in the context operator cannot be avoided in Parlog, but in more powerful concurrent LP languages there is an alternative. If the language possesses an atomic test-and-set primitive (e.g. as in FCP(|,:) and cc(|,↓) [13, 12]) then the store (and its updates) can be hidden from the user.

The basic coding change is to represent the store as a partially instantiated list, ending in a variable. When a new program is added to the store, the variable is atomically unified with a list holding the program term and a new variable.

This technique requires a change to `extract_prog/5` which we previously used to update the store. In FCP(|,:) notation, it becomes:

```
extract_prog(err(_), _, _, St).
extract_prog(ok, Request, Text, St) <-
  make_prog(Text, Clauses),
  add_prog(prog(Request,Clauses), St).

add_prog(Prog, Store) <-
  true : Store = [Prog|_] | true.     % atomic test-and-set
add_prog(Prog, Store) <-
  Store = [_|Store1] | add_prog(Prog, Store1).
```

`add_prog/2` tries to add `prog(Request,Clauses)` to the list, but will be unable to do so until it reaches its variable end. The actual assignment is achieved in

the first clause of `add_prog/2`. It cannot be interleaved with another update and so there is no danger of the modification being corrupted. Consequently, it is unnecessary to return the modified program store as an extra argument of `extract_prog` or `#>`. Indeed, the program store can be completely hidden from the programmer, as in the sequential version of LogicWeb. The resulting `#>` predicate is:

```
lw(RequestMethod, URL, Result)#>Goal
```

If two `#>` goals try to update the store at the same time, one will update it before the other, but the order is undefined.

5.3 Composition Operators

Hiding the program store simplifies LogicWeb goals involving composition operators. For instance, the following goal is written in a similar way to its sequential counterpart:

```
(lw(get, URL1, _) + lw(get, URL2, _))#>Goal
```

`+` is LogicWeb union which forms the set-theoretic union of its arguments. Parallelism can be exploited within the expression on the left hand side of `#>`: the pages `URL1` and `URL2` are downloaded concurrently.

5.4 Web Interactions in LogicWeb Goals

If the underlying concurrent language supports other behaviours, as typified by `timeout/5` and `rate/5`, then these could also be made accessible at the LogicWeb level.

One way of doing that is to extend the `lw()` term to include a user-specified list of interaction parameters. These would be employed at the lower-level to set up and call the necessary predicates.

For example, the following LogicWeb goal sets a time-out value of five seconds, a minimum data transfer rate of 100 KB/s, and permits up to three retrieval attempts (at five second intervals) for downloading the page:

```
lw(get, URL, inter([time(5), rate(100), atts(3)], Result))#>Goal
```

6 Related Work

The research by Cardelli and Davies on service combinators for Web computing is closely related to our proposal [2]. They describe a concurrent model with the aim of reproducing human Web browsing behaviour, such as responses to download failure or slow transmission rates. Their language contains constructs for retrieving a page as a string, representing time-outs and repeating failed downloads. Their approach is also used in the Web data accessing component of WebL, an object-oriented scripting language [5].

There are some important differences between the service combinator view of the Web and our approach, the main one being that the combinator language contains no representation for the data stream between the client and server. For example, Cardelli and Davies' download operator either returns a

complete page string or fails. Also their language does not supply a way for the programmer to access the failure type of a download or to stop a retrieval in mid-execution (possible with `download/4` via the `Result` and `Stop` variables). The consequence of their design decisions is that it is much harder to use combinators to build new behaviours based on an examination of the downloading data stream. For instance, we can implement rate-based monitoring, while the combinator language must contain it as a predefined operator.

The only concurrent LP language with explicit support for Web computation is W-ACE, a constraint-based language which borrows its parallel features from ACE [11]. Their paper discusses the use of AND- and OR- parallelism for agent search in general terms, but is more concerned with the representation of Web pages as structured terms.

7 Conclusion

We have described a concurrent LP-based model of the Web which uses processes and data streams as an abstraction for Web connectivity. This is very useful for addressing issues such as latency and recovery from different types of failure.

Our view of the Web can be readily utilised as a building block for more complex behaviours. In particular, we discussed two designs for a 'concurrent' LogicWeb with different context operators.

We outlined the design and implementation of a `download/4` predicate which is the key element of our approach. It comprises a fairly simple UNIX-based component and a thin concurrent LP layer. Unfortunately, we were unable to find a language that could implement a 'pipe reading' function for incrementally reading the output stream of the UNIX process. We do not believe that adding such functionality would be onerous. We tested our approach using the pipe facilities in SWI-Prolog.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk, *HyperText Transfer Protocol (HTTP/1.0) Specification*, RFC 1945.

[2] L. Cardelli and R. Davies, *Service Combinators for Web Computing*, SRC Research Report 148, Digital Systems Research Center, Palo Alto, California, USA, June, 1997. Available at `ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-148.html`

[3] K.L. Clark and S. Gregory, *Parlog: Parallel Programming in Logic*, ACM TOPLAS 8(1), 1986, 1–49.

[4] S. Gregory, *Parallel Logic Programming in Parlog*, Addison-Wesley, 1987.

[5] T. Kistler and H. Marais, *WebL – A Programming Language for the Web* SRC Research Report, Digital Systems Research Center, Palo Alto, California, USA, 1998. Available at `http://www.elsevier.nl/cas/tree/store/comnet/free/www7/1832/com1832.htm` .

[6] S.W. Loke, A. Davison, and L. Sterling, *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, 1996.

[7] S.W. Loke, A. Davison, and L. Sterling, *Lightweight Deductive Databases on the World-Wide Web*, Proc. of the 1st Workshop on Logic Programming Tools for Internet Applications, JICSLP'96, Bonn, Germany, September, 91–106.

[8] S.W. Loke and A. Davison, *A Logic Programming Approach to Generating Web-based Guided Tours*, PAP'97: 5th Int. Conf. and Exhibition on The Practical Application of Prolog, London, UK, April, 1997.

[9] S.W. Loke and A. Davison, *A Two-level World Wide Web Model with Logic Programming Links*, Second Int. Workshop on LP Tools for Internet Applications, ICLP'97, Leuven, Belgium, 1997.

[10] S.W. Loke and A. Davison, *LogicWeb: Enhancing the Web with Logic Programming*, The Journal of Logic Programming, 36, 1998, 195–240.

[11] E. Pontelli and G. Gupta, *W-ACE: A Logic Language for Intelligent Internet Programming*, ICTAI'97, Proc. of the IEEE 9th Int. Conf. on Tools with AI, 1997, 2–10.

[12] V.A. Saraswat, *Concurrent Constraint Programming Languages*, Ph.D Thesis, Carnegie-Mellon Univ. 1988.

[13] E. Shapiro, *The Family of Concurrent Logic Programming Languages*, ACM Computing Surveys, Vol. 21, No. 3, September 1989, 413–510.

[14] J. Wielemaker, *SWI-Prolog Reference Manual*, Dept. of Social Science Informatics (SWI), Amsterdam, The Netherlands, 1998. Available at `ftp://swi.psy.uva.nl/pub/SWI-Prolog/`.

Appendix

A complete listing of `isdown.c`, the C component of the `download/4` predicate.

```
/* isdown.c */
/* Andrew Davison, Nov. 1998 (ad@cs.ait.ac.th)
   Seng Wai Loke (swloke@cs.mu.oz.au)
*/
/* Retrieve the text of a Web page using its URL, but
   this is interrupted if the stop-file is found
   to exist. The page is output to stdout and is
   followed by "#### result-code"

   Result codes:
      0   : page downloaded okay
      1   : socket creation failure
      2   : DNS lookup error
      3   : connection error
      4   : URL incorrectly formed
      5   : downloaded interrupted by user
```

```
      6   : something wrong with input from socket

   Usage:
     isdown http://www.cs.ait.ac.th/~ad/index.html stopfnm
*/
/*  Compilation on SunOS:
      \gcc -o isdown isdown.c -lnsl -lsocket
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>      /* for read(), write(), close() */
#include <sys/wait.h>
#include <signal.h>

#define SIZE 512         /* max length of a string */

void extract_parts(char url[], char host[], char fnm[]);
int open_socket(char *host);
void send_GET(int sd, char *fnm);
void show_reply(int sd);
void show_result(int result);
void check_stop(char *stopfile, pid_t pid, int sd);
void child_done(int signo);

int main(int argc, char *argv[])
{
  int sd;
  pid_t pid;
  char host[SIZE], fnm[SIZE];

  signal(SIGUSR1, child_done);   /* child signals to parent */

  if (argc != 3) {
    fprintf(stderr, "Usage: isdown URL StopFile\n");
    exit(1);
  }

  extract_parts(argv[1], host, fnm);
  sd = open_socket(host);

  if ((pid = fork()) == 0) {   /* child */
    send_GET(sd, fnm);         /* get the page */
    show_reply(sd);            /* print it to stdout */
    close(sd);
    kill(getppid(), SIGUSR1);  /* tell the parent to exit */
  }
  else   /* parent */
    while(1) {             /* interrupted by child signal */
      check_stop(argv[2], pid, sd);   /* stop the child? */
```

```
      sleep(1);
    }
  return 0;
}

void extract_parts(char url[], char host[], char fnm[])
/*  url should be of the form:
        http://host[/fnm]
    host and fnm are extracted  from url; fnm may be empty
*/
{
  int i = 0, j, k;

  while ((url[i] != '\0') && (url[i] != '/'))
    i++;
  if (url[i] == '\0') {
    fprintf(stderr, "Illegal URL format\n");
    show_result(4);      /* URL incorrectly formed error */
    exit(1);
  }

  i = i + 2;    /* skip both '/'s */
  j = 0;
  while ((url[i] != '\0') && (url[i] != '/'))
    host[j++] = url[i++];      /* build host */
  host[j] = '\0';

  if (url[i] == '\0')          /* no file part */
    fnm[0] = '\0';
  else {
    i++;     /* skip the '/' */
    k = 0;
    while (url[i] != '\0')
      fnm[k++] = url[i++];     /* build fnm */
    fnm[k] = '\0';
  }
}

int open_socket(char *host)
/* Create a socket. Assign host's details to name.
   Create a stream using name, and connect the socket to it.
*/
{
  struct sockaddr_in name;
  struct hostent *hptr;
  int sd;

  if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    show_result(1);    /* socket creation error */
    exit(1);
  }

  /* initialise the name struct to all zeros */
```

```
  memset(&name, 0, sizeof(name));

  name.sin_family = AF_INET;
  name.sin_port = htons(80);         /* host is a HTTP server */

  if ((hptr = gethostbyname(host)) == NULL) {
    perror("gethostbyname");
    show_result(2);    /* DNS lookup error */
    exit(1);
  }

  /* copy the IP address of host into name */
  memcpy(&name.sin_addr.s_addr, hptr->h_addr, hptr->h_length);

  fprintf(stderr, "Trying to contact %s...\n", host);
  if (connect(sd, (struct sockaddr *) &name,
                                sizeof(name)) < 0) {
    close(sd);
    perror("connect");
    show_result(3);    /* connection error */
    exit(1);
  }

  return sd;
}

void send_GET(int sd, char *fnm)
/* Send a GET request for fnm to the server */
{
  char request[SIZE];

  fprintf(stderr, "Sending a GET request...\n");
  sprintf(request, "GET /%s HTTP/1.0\n\n", fnm);
  write(sd, request, sizeof(request));
}

void show_reply(int sd)
/* Print the reply to the GET request */
{
  char buf[SIZE];
  int res;

  fprintf(stderr, "Waiting for a response\n");
  while ((res = read(sd, buf, SIZE)) > 0)  {
    write(1, buf, res);         /* write to stdout */
    /* sleep(1); */   /* testing -- to slow download */
  }
  if (res < 0)   /* read error */
    res = 6;     /* socket read error result code */
  show_result(res);
}

void show_result(int result)
/* Finish the output with the result code details */
```

```
{
  char buf[SIZE];
  fprintf(stderr, "Download result: %d\n", result);
  sprintf(buf, "\n#### %d\n", result);
  write(1, buf, strlen(buf));
}

void check_stop(char *stopfile, pid_t pid, int sd)
/* If stopfile exists (i.e. can be opened) then kill
   the child which is downloading the Web page.
*/
{
  FILE *fp;
  if ((fp = fopen(stopfile, "r")) != NULL) {
    fprintf(stderr, "stop file detected\n");
    kill(pid, SIGKILL);      /* kill child process */
    close(sd);
    fclose(fp);
    remove(stopfile);
    show_result(5);    /* download interruption code */
    exit(1);
  }
}

void child_done(int signo)
/* Executed when the child sends a SIGUSR1 signal to the
   parent to inform it that the page downloading has
   finished. This allows the parent to terminate as well.
*/
{
  fprintf(stderr, "Child has finished\n");
  exit(0);
}
```