Agent-integrated concurrent web programming

Theodore W. Hong*,† and Keith L. Clark

Dept. of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, U.K.

SUMMARY

The web is becoming a rich computation environment supporting an ecology of programs that operate on web information resources, which tend to follow certain structural conventions and be accessed over the network. Webstream is a language designed to address the special characteristics of web data and simplify the development of web applications. It is embedded within the April agent programming language and encapsulates web documents as streams of messages passing through sequences of concurrent April processes, permitting downstream operations to begin in advance while upstream operations are still being completed, in a manner similar to lazy evaluation. Streams can be pipelined through arbitrary combinations of filters which perform transformations such as parsing HTML/XML tags, extracting subset streams based on function closures or regular expressions, and combining or splitting streams in various ways. These facilities make it easy to build a wide variety of web applications such as metasearch engines or web crawlers. The language is particularly suitable for building intelligent information server agents that can monitor and collect data from the web and answer high-level queries or subscriptions from other agents about data that they are interested in. It provides more features than other web-specific languages and is more concise than libraries for general-purpose languages.

KEY WORDS: web programming, agents, information extraction, web intelligence

1. INTRODUCTION

Although the web is often regarded as simply a large database, it is also increasingly becoming a rich computation environment that supports a diverse software ecology of programs whose native domain of operation is the reading, processing, and writing of web information resources. Such programs include web crawlers, personal information agents, comparison-shopping agents,

*Correspondence to: Theodore Hong, Dept. of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, U.K.
†E-mail: t.hong@doc.ic.ac.uk

and more. Web resources themselves have grown in scope from simple static collections of files to dynamic database queries, gateways to other applications like email or multimedia streams, and interfaces to server-side computations.

Computation on the web is similar in many ways to computation on files or objects: web resources can be read or queried and can point to one another. However, web resources also have special characteristics of their own: in particular, they typically follow particular structural conventions and are accessed remotely over the network. The most obvious feature of web resources is that most of them are tagged HTML or XML documents. Hence natural operations on web data include parsing tags, filtering tags, and searching for sequences. Network access raises additional issues of latency, caching, and reliability. Web objects can frequently take a long time to retrieve and may be duplicated on multiple mirror sites with varying bandwidths to the user. Hosts are often unavailable and linked resources can disappear without warning.

Webstream is a web programming language designed to address these issues and simplify the development of web applications, particularly web-aware information agents. It is implemented as a macro extension to the April agent programming language[8] and can be used either on its own or to add web capabilities to April agents. Programming in Webstream uses a paradigm of streams and pipelines which permits a high degree of concurrency to maximize throughput in the face of network latency. Webstream also provides a rich set of components to parse and process HTML/XML documents, including sequence and pattern matching facilities. Furthermore, data from multiple sources can be easily merged, split, and rearranged in a variety of ways. These features make it easy to build a wide variety of web applications with only a small amount of code, including web crawlers, link checkers, extraction wrappers, and metasearch engines, as well as to add web capabilities to intelligent agent applications.

In Section 2, we give a general overview of Webstream's architecture. Section 3 describes some of its pipeline components in detail, while Section 4 shows how to compose pipelines together. In Section 5, we present three sample applications: a concurrent metasearch engine, a queryable web crawler, and a persistent theatre listings agent, and discuss some general concepts for designing programs in the language. Section 7 reviews related work on web programming languages, Section 8 describes Webstream's current status and future work, and Section 9 concludes.

2. OVERVIEW

Webstream is designed as a small domain-specific language[18] embedded within the April agent programming language. In April, agents are represented as lightweight concurrent processes that can interact, maintain state, deliberate, and move from place to place. Communication among distributed agents is carried out using asynchronous message passing in a manner similar to that of Actors[1]. Typically, the main mode of operation of an agent is to wait in a loop for messages to arrive and act on them appropriately. April also provides a number of useful high-level facilities such as pattern matching on complex symbolic structures and function closures that can be treated as first-class data.

In Webstream, web pages are represented by streams of messages flowing through sequences of April processes. Streams originate as raw data downloaded from webservers and are operated on by filter processes that perform various transformations on the data passing through them. Filters execute concurrently and can be arbitrarily composed into pipelines that continuously produce partial results as the data stream flows through them. In this way, computation can be efficiently distributed among multiple concurrent processes, providing higher throughput and modularising processing in different parts of the application. Pipeline outputs can be collected in set abstractions that support querying by external processes or agents using high-level declarative expressions.

2.1. Streams

A stream originates with a source process that creates an HTTP request for a document. As the data arrives, the source process incrementally passes it along to another process as a stream of messages. The receiving process executes concurrently in a separate thread and can operate asynchronously on the data, independent of the progress of the download. The download can be transparently stopped and retried or even redirected to another server without affecting the receiver.

Streams can be passed through sequences of filters to perform HTML/XML parsing and transformation operations. Filters are independent April processes whose input and output message queues can be chained together in series to form a *pipeline* that progressively transforms data flowing through it. A variety of different filters exist that can parse raw text into HTML or XML tags, extract subset streams based on arbitrary computable patterns within tags or regular expression-like sequences of tags, and split or combine streams in various ways.

2.2. Pipelines

The pipeline paradigm is useful because of its ability to easily express arbitrary combinations of components from a standard toolbox. Further, since messaging is asynchronous, all of the components in a pipeline can execute in parallel with one another and with the page download. This arrangement permits downstream components to begin processing data before their upstream predecessors have fully completed evaluation, in a manner analogous to lazy evaluation in a functional programming language.

Here is an example of a Webstream pipeline:

```
|> get_url("http://www.yahoo.com/") |> tags() |> elems("a") |> headw(5) |>>;
```

The components of the pipeline are procedure closures that are forked by the application as concurrent April subprocesses. They are linked together by the `|>` operator, which directs outgoing messages from its left-hand operand to the incoming message queue of its right-hand operand. At the tail of the pipeline, the `|>>` operator directs the final output back to the parent application.

This particular pipeline begins with the data source `get_url`, which retrieves data from a webserver using HTTP and breaks it into a sequence of strings corresponding to individual HTML or XML elements. The raw strings are then converted into a stream of higher-level structures representing parsed HTML/XML tags by the `tags` filter. Next, the data passes

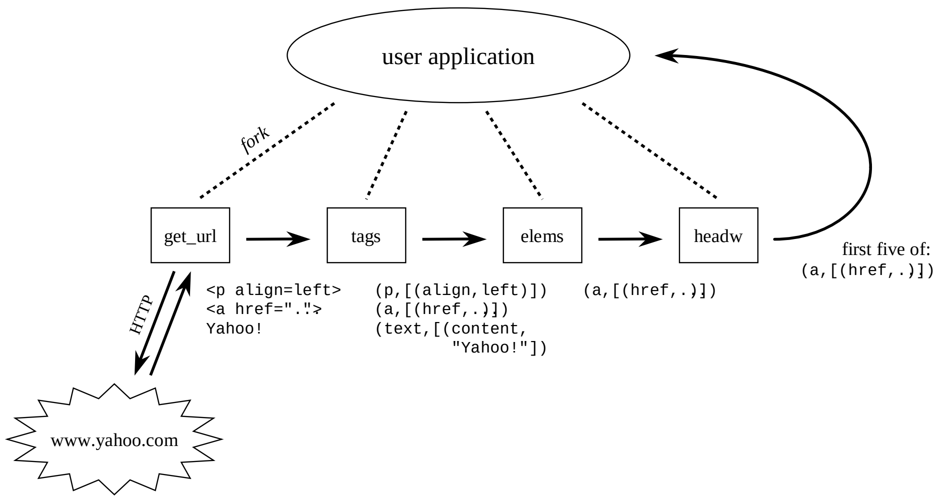

Figure 1. A sample pipeline. The messages in the stream are transformed from raw text strings to parsed tag structures and then filtered by various criteria as they pass through the pipeline.

through two selection filters: `elems`, which selects tags of a given type, and `headw`[†], which outputs the initial portion of its input. Finally, the pipeline terminates with the sink operator `|>>` and sends the final output back to the parent process. (See Figure 1 for an illustration.) The overall effect of this sequence is to extract the first five anchor tags from the Yahoo homepage.

Because of the concurrency in the pipeline, the first five anchors might be identified and sent back before the web page is completely downloaded. This behavior is similar to that of Unix pipelines, but Webstream goes further by permitting pipelines to be merged and split, as well as providing early termination when further results are no longer needed. In this example, once the `headw` filter has seen all five anchors, it will exit and propagate a termination message back along the pipeline to stop any further unnecessary downloading and processing.

Most other web programming languages require documents to be downloaded in full before doing any processing. Another advantage is that multiple pipelines downloading different pages can execute concurrently with one another. Since web downloads often have long latencies, significant speedups can result.

[†]The "w" stands for Webstream and serves to distinguish this filter from the standard April list operator `head`.

2.3. Stream protocol

Webstream messages follow a specific type protocol, but are otherwise identical to normal April messages and can be processed using ordinary April constructs. Hence Webstream can be easily integrated into an April agent system. In fact, the language is actually implemented as a set of macros that are expanded to April code by the compiler.

The stream protocol is defined by the following I/O type declaration:

```
stream_t ::=
     string
   | ws_tag(tag_t)
   | ws_seq(tag_t[][])
   | ws_textseq(string[])
   | ws_status(symbol,attr_t[])
   | ws_end;
```

where `tag_t` is the datatype for parsed tag structures, defined as a tuple containing the tag's type and a list of `attr_t` attribute/value pairs:

```
tag_t ::= (string,attr_t[]);
attr_t ::= (string,string);
```

The various message types are used to carry different types of data, namely raw strings, parsed tags (`ws_tag`), sequences of tags (`ws_seq`), or sequences of strings (`ws_textseq`). As a stream passes through a pipeline, the types of its messages will change to reflect the transformations performed on its data. Some examples of these messages are shown in Section 3.

There are also two special messages: the `ws_status` message, which indicates the status of a given URL as a pair of the HTTP status returned by its server plus a list of returned header/value pairs, and the `ws_end` message, which terminates a stream.

3. PIPELINE COMPONENTS

Webstream provides a toolbox of basic components for creating and parsing data streams, extracting tags and sequences of tags, and splitting and combining streams in various ways. Here we give a sampling of some of them.

3.1. Data sources

`get_url`, `get_url_with_args`, and `post_url` create new streams from URLs by initiating an HTTP connection and requesting the document using the GET or POST methods, respectively. They take as arguments a URL to retrieve and (for `get_url_with_args` and `post_url`) a list of attribute/value pairs to be sent as arguments.

```
-- send GET request for New York Times front page
|> get_url("http://www.nytimes.com/") |>>;
```

```
-- send POST request to Yahoo stock quote service
-- set parameter "s" (stock symbol) to IBM
-- set parameter "d" (quote type) to 5-day chart
|> post_url("http://quote.yahoo.com/q", [("s","IBM"), ("d","5d")]) |>>;
```

Streams can also be created from local HTML/XML files or resumed partway through a previously-abandoned download, using the `get_file`, `resume_get`, and `resume_post` components.

Protocol-dependent processing is performed on the HTTP response to the Webstream request. For example, this might involve re-requesting data which has moved to a different URL. As the text of the document arrives, it is split into pieces at HTML/XML element boundaries and sent out as a stream of strings. For example, the New York Times pipeline shown above would take the following (simplified) web page:

```
<html>
<head><title>The New York Times on the Web</title>
<meta http-equiv="Refresh" content="900">
<meta http-equiv="Expires" content="0">
<a href="/pages/world/index.html">International</a><br>
<a href="/pages/national/index.html">National</a><br>
   ⋮
</html>
```

and generate the stream:

```
"<html>"
"<head>"
"<title>"
"The New York Times on the Web"
"</title>"
"<meta http-equiv=\"Refresh\" content=\"900\">"
"<meta http-equiv=\"Expires\" content=\"0\">"
"<a href=\"/pages/world/index.html\">"
"International"
"</a>"
"<br>"
"<a href=\"/pages/national/index.html\">"
"National"
"</a>"
"<br>"
   ⋮
"</html>"
ws_end
```

Streams are terminated by the end-of-stream token `ws_end`. If an error occurs and the document cannot be retrieved, then an empty stream consisting solely of `ws_end` will result.

3.2. Parsing

The `tags` filter parses a stream of unparsed HTML/XML text into a stream of structures representing tags. Each tag is represented as a pair of the tag's type and a list of its attribute/value pairs. Free text that is not contained within a tag is converted to a special tag type, `text`, having a single `content` attribute whose value is the corresponding literal text. For example, adding a `tags` filter to the end of the New York Times pipeline:

```
-- get and parse New York Times front page
|> get_url("http://www.nytimes.com/") |> tags() |>>;
```

would give the following stream of tag structures:

```
ws_tag("html", [])
ws_tag("head", [])
ws_tag("title", [])
ws_tag("text", [("content","The New York Times on the Web")])
ws_tag("/title", [])
ws_tag("meta", [("http-equiv","Refresh"), ("content","900")])
ws_tag("meta", [("http-equiv","Expires"), ("content","0")])
ws_tag("a", [("href","/pages/world/index.html")])
ws_tag("text", [("content","International")])
ws_tag("/a", [])
ws_tag("br", [])
ws_tag("a", [("href","/pages/national/index.html")])
ws_tag("text", [("content","National")])
ws_tag("/a", [])
ws_tag("br", [])
    ⋮
ws_end
```

3.3. Selection

`elems` filters a stream of tag structures, passing only HTML/XML elements of a given type. For example, extending the New York Times pipeline with the following:

```
-- select meta tags
... |> elems("meta") |>>;
```

would result in the stream:

```
ws_tag("meta", [("http-equiv","Refresh"), ("content","900")])
ws_tag("meta", [("http-equiv","Expires"), ("content","0")])
    ⋮
ws_end
```

Similarly, `elemset` selects HTML/XML elements belonging to a given set of types. Extending the pipeline with the following filter:

```
-- select anchors and text
... |> elemset(["a","text"]) |>>;
```

would give us the stream:

```
ws_tag("text", [("content","The New York Times on the Web")])
ws_tag("a", [("href","/pages/world/index.html")])
ws_tag("text", [("content","International")])
ws_tag("a", [("href","/pages/national/index.html")])
ws_tag("text", [("content","National")])
   ⋮
ws_end
```

pats selects tags matching a given pattern specified by a boolean function closure. This function is applied to each incoming tag, and those yielding true are selected. Any April code fragment returning a boolean value can be used, which might perform arbitrary computations, send and receive messages, or even spawn subsidiary pipelines. For example, suppose we define a function broken that extracts URLs from anchor tags and uses a pipeline to test whether they can be retrieved successfully. pats could use this function to find the broken links in a document:

```
-- select broken links
... |> elems("a") |> pats(broken) |>>;
```

The function broken might be defined as follows:

```
-- is this link broken?
broken(tag) => valof {
    -- send HEAD request for URL referenced by tag
    |> head_url(get_href(tag)) |>>;

    -- check for returned HTTP_OK status
    receive ws_status(status,headers) ->>
        valis (status != 'http_ok)
};
```

Here, get_href is a helper function that extracts the URL reference from a tag. head_url performs a HEAD request that returns a URL's status and associated HTTP headers in a ws_status message. The April receive statement waits for a status message to arrive from the head_url pipeline, and the valof...valis construct ("value of... is") sets the return value of the function by comparing the returned status against the literal symbol 'http_ok.

get_href is defined as follows:

```
-- extract URL from href attribute in tag
get_href(tag) => valof {
    (tagtype,attrlist) .= tag;
    if (("href",url) in attrlist) then
        valis url
    else
        valis ""
}
```

In this code, the April pattern match operator .= is used to decompose a tag structure into a pair of its tag type and attribute list. The list is then searched for an href attribute and the corresponding value is bound to the url variable and returned.

Other selection filters are headw, dropw, and tailw, which select the first N, all but the first N, or the last N messages in a stream, respectively. Note that tailw introduces a synchronisation delay because the stream must finish before anything can be sent.

3.4. Sequence extraction

grep searches a stream for sequences of tags matching a given regular expression on the alphabet of HTML/XML element types (that is, html, /html, head, /head, and so on, plus text). The syntax follows that of Unix grep, with the addition that braces are used to delimit capturing subexpressions—any subsequences matching these are extracted and returned. This functionality is particularly useful for writing wrappers to extract information from web pages.

For example, the following pipeline:

```
-- select anchor and description
... |> grep("p {a} {(text|b|/b)*} /a /p") |>>;
```

searches for HTML paragraphs (p.../p) that contain an anchor (a) followed by an arbitrary amount of text description (text) mixed with bold tags (b, /b). Consider the following portion of a search engine result for the singer Dar Williams:

```
<ol>
<li>
<p><a href="http://darweb.org/">
Since 1995, <b>Dar</b> Web has been the flagship site...</a>
</p></li>
<li>
<p><a href="http://www.darwilliams.com/">
<b>Dar</b> <b>Williams</b>.com - the green version</a>
</p></li>
</ol>
```

This fragment contains two such paragraphs, one for darweb.org and one for www.darwilliams.com. Given this input, the pipeline above would extract the sequences:

```
ws_seq([[("a", [("href","http://darweb.org/")])],
        [("text", [("content","Since 1995, ")]),
         ("b", []),
         ("text", [("content","Dar")]),
         ("/b", []),
         ("text", [("content"," Web has been the flagship site...")])]])
ws_seq([[("a", [("href","http://www.darwilliams.com/")])],
        [("b", []),
         ("text", [("content","Dar")]),
         ("/b", []),
         ("text", [("content"," ")]),
         ("b", []),
         ("text", [("content","Williams")]),
         ("/b", []),
         ("text", [("content",".com - the green version")])]])
```

Each ws_seq message corresponds to one match and contains a list of captured subexpressions. Each such subexpression, in turn, consists of a sequence of matching tags. Hence a ws_seq message contains a list of lists of tags. In this case, there are two subexpressions per match: the first, matching the anchor, contains only a single tag, a; while the second, matching the description, contains a variable number of text, b, and /b tags.

Note that the p, /a, and /p tags are not extracted, since they are not contained within any capturing subexpressions but are only used for context in finding matches.

3.5. Text processing

After using grep, it is frequently convenient to use elems or elemset to discard irrelevant tags. For example, adding on to the previous search engine pipeline with:

```
-- select anchors and text
... |> elemset(["a","text"]) |>>;
```

would give us the stream:

```
ws_seq([[("a", [("href","http://darweb.org/")])],
        [("text", [("content","Since 1995, ")]),
         ("text", [("content","Dar")]),
         ("text", [("content"," Web has been the flagship site...")])]])
ws_seq([[("a", [("href","http://www.darwilliams.com/")])],
        [("text", [("content","Dar")]),
         ("text", [("content"," ")]),
         ("text", [("content","Williams")]),
         ("text", [("content",".com - the green version")])]])
```

Notice that the description text is broken into multiple pieces where the bold markup used to be. This problem can be fixed by the collapsew filter, which collapses adjacent text tags into a single one. In the running example, extending the pipeline with:

```
-- collapse adjacent text fragments
... |> collapsew() |>>;
```

would now give us:

```
ws_seq([[("a", [("href","http://darweb.org/")])],
        [("text", [("content","Since 1995, Dar Web has been the flagship
                              site...")])]])
ws_seq([[("a", [("href","http://www.darwilliams.com/")])],
        [("text", [("content","Dar Williams.com - the green version")])]])
```

Finally, we can convert these tag structures to printable strings by using the stringify filter. In this example, ending the pipeline with:

```
-- flatten into strings
... |> stringify() |>>;
```

would give us:

```
ws_textseq(["http://darweb.org/",
            "Since 1995, Dar Web has been the flagship site..."])
ws_textseq(["http://www.darwilliams.com/",
            "Dar Williams.com - the green version"])
```

A complete example of this type of pipeline can be seen in the code for the metasearch engine in Section 5.1.

4. COMPOSING PIPELINES

Pipelines can be composed together in various ways. The `orw` operator joins two pipelines such that the composite pipeline forwards messages from whichever one starts sending first, terminating the other. This behavior is useful when two sources are mirrors of each other and either will do. For example:

```
-- choose fastest mirror site (US or Ireland)
|> get_url("http://www.cpan.org/") orw
       get_url("http://cpan.indigo.ie/") |>>;
```

`unionw`, `intersectw`, and `diffw` merge pipelines according to the usual set semantics. That is, `unionw` forwards messages that are received from either pipeline, removing duplicates, while `intersectw` forwards only messages that are received from both. The `diffw` operator forwards those sent by its left-hand argument but not its right-hand argument. These behaviors are useful when two sources provide data that may overlap. For example:

```
-- merge two bookmark files
|> (get_file("bookmark1.html") |> tags() |> elems("a")) unionw
       (get_file("bookmark2.html") |> tags() |> elems("a")) |>>;
```

Note that `diffw` does not show the lazy evaluation semantics since its right-hand argument must be fully evaluated before it can produce any output. `intersect` is limited by a similar problem. As soon as its inputs diverge, it must wait until the head of one input stream matches a message somewhere in the other stream. Until it finds a match for one side or the other, it cannot output anything, since messages must remain in order. If the head of one input truly has no match in the other stream, then that side will stay blocked until the other stream completes and confirms this fact. `union`, on the other hand, has no such difficulties since it can drop duplicates as soon as they are seen—it doesn't have to wait for anything.

Since pipeline statements return immediately after forking the appropriate processes, a sequence of such statements will create a set of concurrently-executing pipelines. Incoming messages from each will be interleaved nondeterministically in the parent's incoming message queue, although messages from any particular pipeline will remain in the correct relative order. This behavior is useful when combining data from multiple independent sources. For example:

```
-- conduct parallel search engine queries
H1 = get_url("http://www.google.com/search?q=april") |>>;
H2 = get_url("http://www.altavista.com/cgi/?query=april") |>>;
```

Here, instead of the usual statement form of the pipelines (that is, beginning with `|>`), we use a functional form that returns the *handles* of the spawned pipelines. Handles are unique process identifiers that are used to identify the senders and receivers of April messages. These particular handles are stored in the variables `H1` and `H2`, in order to later be able to determine which pipeline a particular message came from.

Streams can also be duplicated by giving a list of pipelines after a `|>` operator. This causes each incoming message from the left-hand side to be multicast to all of the pipelines on the right-hand side. For example, the following pipeline:

```
-- select the first 10 and last 10 tags
...|> tags() |> [headw(10) |>>, tailw(10) |>>];
```

splits a tag stream into two copies, one of which is filtered by `headw` and the other of which is filtered by `tailw`.

5. PRACTICAL EXAMPLES

To demonstrate some of Webstream's capabilities, we describe three sample applications: a metasearch engine, a queryable web crawler, and a theatre listings agent.

5.1. Metasearch engine

The first example is a simple metasearch engine, implemented as a Webstream procedure. It takes a query string as its single argument. By starting multiple concurrent pipelines to different search engines, it can collect and display results on the fly from whichever sites respond first. See Figure 2 for an illustration of its architecture.

Here is the code listing:

```
metasearch(query) {
    -- specification of the engine name, URL, name of query argument,
    -- and result format for each underlying search engine to be used
    engines = [("google",
              "http://www.google.com/search",
              [("q",query)],
              "p {a} {(text|b|/b)*} /a br font "
                  ++ "{(text|b|/b|br)*} br font text /font "
                  ++ "a text /a text a text /a /font"),
             ("lycos",
              "http://www.lycos.co.uk/cgi-bin/pursuit",
              [("query",query)],
              "li b {a} {text} /a /b br font? {text}"),
             ("looksmart",
              "http://www.looksmart.com/r_search",
              [("key",query)],
              "dl dt {a} {text} /a /dt dd {text} /dd")];
    pipelines: [];

    -- start pipelines
```

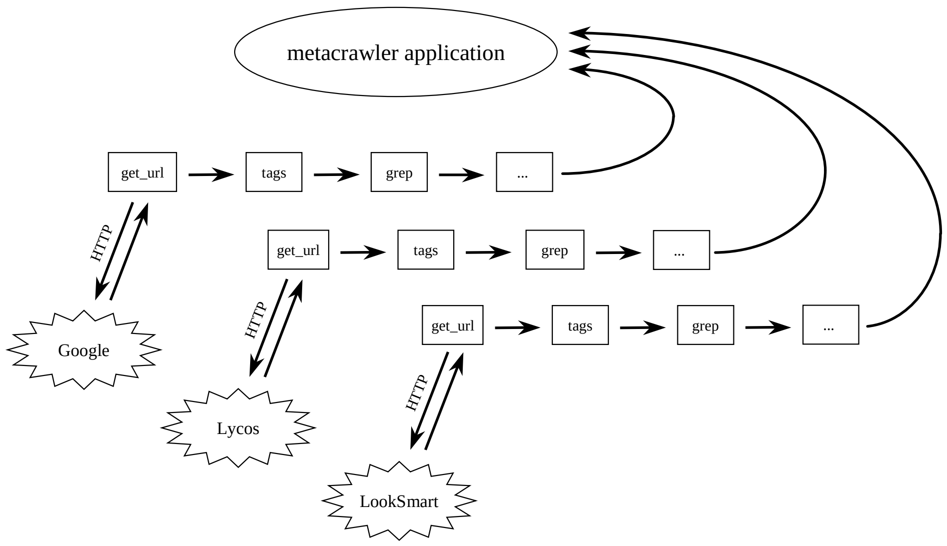

Figure 2. Architecture of a metasearch engine. The picture has been simplified by omitting some filters, represented by ellipses.

```
for (name,url,args,format) in engines do {
    H = get_url_with_args(url, args)
          |> tags() |> grep(format)
          |> elemset(["a","text"]) |> collapsew()
          |> stringify() |>>;
    pipelines := [(H,name),..pipelines];
};

-- collect results, while there are pipelines left
while (pipelines != []) do {
    receive {
        ws_textseq([link,title,description]) ->> {
            -- got a search result
            if (sender,name) in pipelines then
                fill("["++name++"]",12) ++ title ++ "\n"
                    ++ fill(description,79) ++ "\n"
                    ++ link ++ "\n\n" >> stdout;
        }
      | ws_end ->> {
            -- pipeline finished, remove from list
            pipelines := pipelines reject (sender,_);
        }
    }
```

```
        }
    }
```

5.1.1. *Explanation of the code.*

The metasearch procedure is called with a query string as an argument, and begins by initialising the `engines` list with the parameters for each search engine to be contacted (name, URL, query argument, and result format). It loops through the list and spawns a pipeline for each. The handle of the pipeline is returned in the variable `H` and is added to the `pipelines` list together with the name of the engine.

Each pipeline begins with a `get_url_with_args` source operator that opens a connection to a search engine using the appropriate URL and query argument. Next, the `tags` operator parses the returned result page into HTML tags.

The `grep` operator then scans the page to extract individual search results. The particular format of the results will vary by engine, and is specified by the regular expressions in the `engines` list. Curly braces delimit the subexpressions that we want to extract—namely, the link, the title, and the description.

Finally, the stream is filtered by `elemset` to remove irrelevant elements other than text and anchor tags, and the resulting sequences are collapsed and converted to strings.

The main part of the program is a while loop around a message `receive` statement that waits for a message to arrive and then executes different cases depending on the type of message received. When a search result to arrives in a `ws_textseq` message, the program looks up the message's `sender` in in the `pipelines` list, in order to find the name of the search engine that produced it, and formats it for printing to standard output. A `ws_end` message indicates that a pipeline has terminated and should be removed from the active pipeline list. The program ends once all of the pipelines have finished.

Notice that a new data source can easily be incorporated by adding its engine name, URL, name of query argument, and result format to the `engines` list. This simple application does not check its results for duplicates or attempt to sort them; it simply prints them out in the order received. A more sophisticated version might rank results by relevancy score and remove duplicate URLs.

5.1.2. *Program output.*

The output from a sample program run looks like this:

```
[google]    DarWilliams.com | the green version |
THE GREEN WORLD, GALLERY, TOUR, BIOGRAPHY. INTERACTIVE, PRESS ARCHIV
http://www.darwilliams.com/new_index2.html

[google]    Welcome to Dar Williams.net
 ... Fan activities: birthday, SET, ICQ. Discography of Dar Williams
http://www.darwilliams.net/

[looksmart] Dar Williams@RollingStone.com
Dar Williams news & reviews, concert pics, games, MP3s & videos..Rol
```

```
http://www.rollingstone.com/artists/default.asp?oid=2167&afl=looks

[lycos]     Dar Williams Page
Includes photos, guitar chords, quotes and links
http://www.angelfire.com/sd/darwilliams/

[lycos]     Dar Williams Pictures
Personal collection of Dar Williams photos taken mainly at concerts
http://www.mtsu.edu/~mbc2b/dar.html

[google]    Dar Williams - home
Since January 1995, Dar Web has been the flagship website in the Dar
http://www.darweb.org/
```

5.2. Queryable web crawler

The second example is a queryable web crawler. It takes three arguments: an initial URL, a maximum number of pages to visit, and a maximum number of pipelines allowed to be active at any one time (to limit resource consumption).

During execution, the crawler will spawn multiple pipelines that will be running in parallel to download different pages and extract their links. The outputs of these pipelines will all be merged in the crawler's incoming message queue, as described in Section 4. As a result, the crawler does not have to wait for any single request to complete, but can eagerly follow links from any of the active requests as they progress, greatly reducing the time spent waiting for dead links. The resulting crawl can be regarded as as a breadth-first traversal in which links closest to the user in terms of latency are followed first.

Here is the code listing:

```
crawler(init_url, maxpages, maxpipes) {
    -- print opening banner
    -- string%% coerces its argument to a string
    "Starting crawler on " ++ init_url ++ "\n" >> stdout;
    "with maxpages=" ++ string%%maxpages ++ ", " >> stdout;
    "maxpipes=" ++ string%%maxpipes ++ "\n\n" >> stdout;

    -- initialize variables
    pages: 0;
    active: 0;
    pending: 1;
    links: [];
    source_urls: [];

    -- add initial URL to jobs queue
    job(init_url) >> self();

    -- repeat while there's work left to do
    while (active > 0 || pending > 0) do {
        receive {
            job(url) :: pages < maxpages && active < maxpipes ->> {
                -- got a job, and haven't reached page limit and slot is free
```

```
                    -- start new pipeline
                    H = get_url(url) |> tags() |> elems("a") |>>;
                    string%%H ++ ": starting " ++ url ++ "\n" >> stdout;

                    -- save URL and increment counters
                    source_urls := [(H,url),..source_urls];
                    pages := pages + 1;
                    active := active + 1;
                    pending := pending - 1;
                }
              | job(url) :: pages >= maxpages ->> {
                    -- got a job, but page limit exceeded
                    -- just discard it
                    pending := pending - 1;
                }
              | ws_tag(T) :: (sender,src_url) in source_urls ->> {
                    -- got an anchor tag
                    -- calculate absolute URL of destination and print it
                    dest_url = canonicalize(get_href(T), baseof(src_url));
                    string%%sender ++ ": found " ++ dest_url ++ "\n" >> stdout;

                    -- add to job queue, if necessary
                    if (followable(dest_url) && !((_,dest_url) in links)) then {
                        job(dest_url) >> self();
                        pending := pending + 1;
                    };

                    -- save link
                    links := [(source_url, dest_url),..links];
                }
              | ws_end ->> {
                    -- pipeline finished, decrement counter
                    string%%sender ++ ": done\n" >> stdout;
                    active := active - 1;
                }
            }
        }
    };


    -- is the URL followable?
    followable(url) =>
        ((A ++ (B in [".htm",".html",".cgi",".pl","/"])) .= url);
```

5.2.1. *Explanation of the code.*

The main body of the program is a message `receive` statement which is iterated while there are active pipelines or pending jobs remaining. The crawler's incoming message queue is used to manage links and pipelines. Program execution begins with the crawler sending itself a `job` message to visit the initial URL, using the message send operator `>>`. Pending jobs stay in the incoming message queue until the crawler decides to start a pipeline to visit its URL.

Whenever an executing pipeline finds a new link, the program prints the URL and then sends itself a new `job` message reminding itself to visit that page at some future time.

More specifically, there are several different cases guarded by `::` conditions that are tested by the message receive. The first case is that the program receives a `job` message containing a URL to follow. If the maximum page limit has not yet been reached and the number of active pipelines is below the pipeline limit, the crawler starts a new pipeline. This pipeline retrieves the URL, parses the downloaded data into tags, and filters out the anchor tags. Its handle is saved together with the URL in the `source_urls` list, so that we can later identify the page that an incoming link came from. The number of pages visited and the number of active pipelines are incremented, and the number of pending jobs is decremented.

In the second case, a `job` message is received, but the page limit has been reached. When that happens, the job is simply discarded.

The third case processes `ws_tag` messages containing the anchor tags that the pipelines are sending back. When the program receives a tag, it looks up the sender of the tag in the `source_urls` list. It then uses the base part of the source URL to convert the (possibly relative) destination URL to absolute canonical form and prints it, along with the handle of the pipeline that sent the tag. If the destination URL is `followable` (defined as ending with `.htm`, `.html`, `.cgi`, `.pl`, or `/`) and has not been seen before in the `links` list (that is, no element of the list matches the pattern `(_,dest_url)`, where `_` means *don't care*), then it is queued as a new `job` message and the number of pending jobs is increased by one. In any case, the source and destination URL's are added as a pair to the `links` list.

The fourth case processes `ws_end` messages sent by pipelines upon termination. When the crawler receives such a message, it decrements the active pipeline counter.

One final implicit case remains. If a pending job message is seen and the page limit has not been reached, but there are too many pipelines already executing, the message will not satisfy any of the cases and will be left in the incoming message queue. The main process then suspends, waiting for a new message to arrive. After processing the next message, it will automatically recheck its queue from the beginning. In this way, when a pipeline slot becomes available after a `ws_end`, the deferred job can be picked up and processed.

The crawling part of the program finishes when there are no more active or pending jobs, meaning that all visited pages have been completely examined and either the maximum number of pages have been visited or no more links can be found.

5.2.2. Program output.

The output from a program run looks like this:

```
Starting crawler on http://www.doc.ic.ac.uk/~twh1/
with maxpages=6, maxpipes=3

29: starting http://www.doc.ic.ac.uk/~twh1/
29: found http://www.doc.ic.ac.uk/~twh1/academic/
34: starting http://www.doc.ic.ac.uk/~twh1/academic/
29: found http://www.doc.ic.ac.uk/~twh1/personal/
39: starting http://www.doc.ic.ac.uk/~twh1/personal/
29: done
```

```
39: found http://www.doc.ic.ac.uk/~twh1/personal/calvino/
44: starting http://www.doc.ic.ac.uk/~twh1/personal/calvino/
34: found http://www.doc.ic.ac.uk/~twh1/personal/
39: found http://www.doc.ic.ac.uk/~twh1/academic/
34: found http://www.doc.ic.ac.uk/~twh1/academic/resume-web.html
39: found http://www.doc.ic.ac.uk/~twh1/6AB4876B.asc
34: found http://www.doc.ic.ac.uk/~twh1/academic/resume-web.pdf
44: found http://www.nytimes.com/books/99/10/31/specials/calvino.html
34: found http://www.freenetproject.org/
34: found http://www.independent.co.uk/story.jsp?story=42801
44: found http://www.doc.ic.ac.uk/~twh1/personal/calvino/baron.html
34: found http://www.newscientist.com/news/news.jsp?id=ns9999814
34: found http://www.doc.ic.ac.uk/~twh1/academic/newsnight-napster.rm
34: found http://www.oreilly.com/catalog/peertopeer/
```

5.3. Theatre listings agent

The final example is a theatre listings agent that collects information about current shows from various data sources. Each day, it checks its sources to see whether they have changed since the last visit, and if so, updates its set of listings. Clients can send arbitrary queries in the form of boolean function closures to be executed against the listings database. New sources can also be added by sending the agent a message containing the source's name, URL, and listing format. The source will then be added to the list of data sources and will start to be visited the next time the agent updates its listings.

The agent is divided into two sub-agents: a monitor that performs the update checking and data extraction, and a query answerer that maintains the listings database and responds to client queries.

5.3.1. Monitor.

Here is the code for the monitor:

```
-- this agent monitors data sources once per day
-- and sends updates to the query answering agent
monitor_agent(initial_sources) {
    sources: initial_sources;
    lastchecktime: 0;

    while true do {
        for source in sources do {
            -- check if sources were modified since lastchecktime
            if (modified_since(source, lastchecktime)) then {
                -- tell query answering agent to discard old listings
                ('updating, source) >> handle%%"query_agent";

                -- start a new pipeline to extract updated listings
                spawn extractor_agent(source);
            }
        };
```

```
            lastchecktime := now();

            -- set alarm for 1 day ahead
            spawn alarm(1 days);

            -- process messages until alarm rings
            repeat {
                ws_textseq([source,show,theatre,genre,lowprice,highprice]) ->> {
                    -- got a listing
                    -- forward to query answering agent
                    ('listing, (source,show,theatre,genre,
                                number%%lowprice,number%%highprice))
                        >> handle%%"query_agent";
                }
              | ('new_source, source) ->>
                   -- got a new data source
                   -- add to the list of sources
                   sources := [source,..sources]
            } until 'ding;
        }
    };


    -- was source modified since lastchecktime?
    modified_since(source, lastchecktime) => valof {
        (name,url,format) .= source;
        |> head_url(url) |>>;

        -- check Last-Modified timestamp
        receive ws_status(status,headers) ->> {
            if (("last-modified",lastmoddate) in headers
                   && (parsedate(lastmoddate) <= lastchecktime) then
                       valis false;
        };

        -- if no timestamp or lastmodtime > lastchecktime, return true
        valis true;
    };


    -- extract data from source
    extractor_agent(source) {
        (name,url,format) .= source;

        -- start a pipeline
        |> get_url(url)
               |> tags() |> grep(format)
               |> elems("text") |> collapsew()
               |> stringify() |>>;

        -- forward all messages to our parent process
        repeat {
            M ->> {
                M >> creator();
            }
        } until ws_end;
    };


    -- auxiliary process to keep a timer
    alarm(delaytime) {
        -- sleep for delaytime
        delay(delaytime);

        -- send alarm back to parent process
        'ding >> creator();
    };
```

The monitor is started with a list of initial data sources. Each one is checked to see whether it has been `modified_since` the `lastchecktime`. The `modified_since` function uses a HEAD pipeline to retrieve the Last-Modified header for the data source. If present, the last-modified date is parsed and converted into seconds since the epoch and compared to the last check time. The source is considered to need revisiting if either it was modified since the last check time or its Last-Modified header is not available.

If a source needs to be revisited, a message is sent to the query answering agent to drop its previous listings from the database and a new extractor process is spawned to extract its updated data. For simplicity, here we just represent a source as a `(name,url,format)` triple whose data is extracted by a simple pipeline similar to that used in the metasearch example. In general, however, the extractor could perform much more sophisticated processing such as parsing a page according to a context-free grammar and learning its format, as described in [16].

After checking all the sources, the monitor starts an alarm process to notify itself when a day has passed. It then enters a `repeat` loop waiting for messages to arrive and processing them. Extracted listings contain the name of the data source, the name of the show, the theatre where it is playing, its genre, and the low and high prices of tickets. When a listing arrives, it is repackaged and sent to the query agent to be added to the list of performances. The monitor can also be informed of a new data source through a `'new_source` message containing the source's details such as name, URL, and listing format. The new source is added to the `sources` list and will be visited the next time the sources are checked.

When the alarm `ding`s, the monitor stops processing messages and loops back to check the data sources again.

5.3.2. Query answerer.

Here is the code for the query answerer:

```
    -- this agent provides the query interface to the outside world
    query_agent(trusted_clients) {
        listings: [];
        -- repeat forever
        while true do {
            receive {
                ('listing, (source,show,theatre,genre,lowprice,highprice)) ->> {
                    -- got a listing
                    -- add to the database
                    listings := [(source,show,theatre,genre,lowprice,highprice)
                                 ,..listings];
                }
              | ('updating, source) {
                    -- monitor agent is updating listings from this source
                    -- discard old listings
                    listings := listings reject (source,_,_,_,_,_);
                }
              | ('query, query_func) :: sender in trusted_clients ->> {
                    -- got a query
                    -- find the set of matching listings
                    results = setof {listing : listing in listings
                                 && query_func(listing)};
                    ('answer, results) >> sender;
                }
            }
        }
    };
```

The query answerer is started with a list of trusted clients and an empty set of `listings`. It then loops forever processing messages. When a listing arrives from the monitor agent, it is added to the `listings` database. If the monitor tells the query agent that a source is being updated, the query agent discards all previous listings from that source using a pattern match and waits for the updated listings to arrive.

Clients can send arbitrary queries in the form of boolean function closures that are executed against the database. Recall that function closures are first-class data values in April. The agent performs a logical query over the `listings` database using an April `setof` expression. This expression applies the query function as a test to each listing in turn and collects the set of matching results, which is subsequently returned in an `'answer` message. For security purposes, only senders in the `trusted_clients` list are permitted to send queries.

On the client side, a query could be constructed in the following way:

```
    ('query, {(source,show,theatre,genre,lowprice,highprice) =>
                 genre == "comedy" && lowprice < 20})
        >> query_agent;
```

Here, the query function is a function closure that takes a `(source, show, theatre, genre, lowprice, highprice)` tuple and asks for comedy shows where the lowest-priced ticket costs less than 20 dollars.

As a more sophisticated extension, we could add code to enable clients to express queries in a higher-level format such as FIPA ACL[13], or to subscribe to be informed asynchronously whenever new data arrives that satisfies a persistent query.

6. Designing Webstream programs

Taking these sample applications as a starting point, we can begin to see some common general points for designing Webstream programs.

Webstream is not a conventional imperative programming language like C or Java, but rather combines elements of dataflow[19] and event-driven languages[26]. As illustrated by the sample applications, the structure of a typical Webstream program can be roughly divided into two main sections, a data retrieval section and a data handling section. The data retrieval section is the dataflow-like part where pipelines are set up to retrieve and transform data from websites, while the data handling section is the event-driven part where a primary loop responds to retrieved data. A basic template for this structure might look something like the following:

```
program(parameters) {
    -- start pipelines
    handle_1 = get_url(parameters) |> ... transform ... |>>;
    handle_2 = get_url(parameters) |> ... transform ... |>>;
      ⋮
    handle_n = get_url(parameters) |> ... transform ... |>>;


    -- process results
    repeat {
        data_message_1 ->> do something
      | data_message_2 ->> do something else
            ⋮
      | ws_end ->> notice completed pipeline
    } until 'done
}
```

To program in this idiom, the programmer should first think about the location and format of the sites to be visited and what data is desired to be extracted from them. She can then assemble pipelines to perform the necessary transformations. Once this is done, the programmer needs to decide how the pipelines are to be started and managed. Will they simply run once and then exit, or will additional pipelines be started to act on new incoming data? Will pipelines be started immediately or held back in a queue to conserve resources?

The other major part of the design is to consider how the transformed data will be used. It is important to adopt a parallel processing mentality from the outset, as the pipelines will execute in parallel threads and their output may be arbitrarily interleaved. The branches of the message receive statement should therefore largely be reactive and stateless, although some state can be maintained by storing information on a per-pipeline basis and using the sender handle of each message to look up which pipeline it came from. This part of the code also needs to keep track of pipeline termination messages and determine how and when (if ever) the program should exit.

7. RELATED WORK

Related work on web programming languages can be broadly divided into general languages for web computation and special-purpose languages for HTML information extraction.

7.1. General web languages

General web languages have a broad vision of the web as an environment for distributed computation that may include communicating agents, persistent data, and complex types embedded in HTML. One of the first descriptions of the web as a general computation environment was given by Cardelli and Davies[6], who described some of the characteristics that a web programming language might need and introduced the notion of service combinators, operators that can be used to combine primitive page retrieval actions into complex control structures in order to manage the unreliability of network transfers. The service combinator concept was implemented by WebL[20], a Java-based scripting language which also provides a markup algebra for manipulating regions of text within pages.

Connor *et al.* proposed Hippo[10], a system for persistent programming on the web. Orthogonal persistence for distributed objects is provided by `extern` and `intern` statements that can operate on either a local namespace over a filesystem or a global namespace over URL's, permitting data to be shared across the network by distributed applications. Using HTTP, Hippo can transport typed data between machines encoded in specially-structured HTML files containing type information. The Hippo Core Language[9] provides a special URL type, which when evaluated causes an immediate fetch of the URL referred to and evaluates to a string containing the contents of the file retrieved. It also provides alternate and parallel computation constructs, similar to Cardelli's service combinators, for dealing with network nondeterminism.

LogicWeb[12, 23] is an interesting attempt to apply a declarative, rather than procedural, abstraction to the web. Web pages are retrieved using the `download/4` logical predicate, which incrementally binds one of its arguments to a term representing the data as it arrives. By combining `download/4` with concurrent logic programming constructs, activities such as switching a request to a mirror site or repeating a request until it succeeds can be expressed. Once downloaded, a page is converted into a small logic program that is a collection of clauses representing facts about it (such as its title or a list of its links) plus any clauses explicitly added by the page's author using special LogicWeb markup tags. These facts can then be used as a knowledge base context for satisfying goals, for example searching for a page on a given subject. However, the conversion predicate is limited to enumerating tags of different types (links, images, etc.) and does not provide support for general parsing. Similar computations can be carried out in Webstream using `setof` expressions.

Another logic programming language, Golog[22], has been adapted[24] for performing customized user interactions with services on the emerging semantic web[4]. The extension permits users to specify personal constraints on actions to be performed, using a situation calculus notation, as well as specifying nondeterministic partial orderings of actions. An interpreter was developed that senses online to collect information needed to execute the program, while simulating actions offline to permit backtracking before producing a final plan.

The `<bigwig>` language[5] is a language for programming interactive web services. Its core is a session-centered service model that provides a template-based mechanism for dynamic page construction. The language also provides form-field validation using regular expressions and concurrency control using temporal logic. Like Webstream, `<bigwig>` is implemented by macro transformations on top of a base language.

Wexus[25] is a uniform computation model for building distributed societies of agents running across multiple web *computation places* such as servers, clients or proxies. It is based on the Nexus[14] model of computation as a series of one-way remote service requests invoked on global pointers to distributed objects. Issuing a remote service request causes a set of arguments to be serialized and transferred to the computation place pointed at by the global pointer, where a handler is started to carry out the request. When objects are no longer required, they are finalized using a distributed garbage collection algorithm. Remote service requests are integrated with HTTP in two ways: they can be transported on top of HTTP to tunnel through firewalls, and HTTP transactions can be transformed into special service requests on callback handlers set up inside Wexus-aware webservers. The latter method can be used to insert computation into web browsing sessions; for example, to implement page reference counting or shared personal caches. Wexus can also be used to build applications such as a distributed web traversal engine that runs a local computation on each server host encountered.

7.2. Extraction languages

Another set of web languages are more narrowly designed for manually writing extraction wrappers for websites. These languages are essentially text-processing languages specialized for handling HTML and typically do not perform network interactions.

The TSIMMIS information integration project[7] uses handcoded extraction wrappers written in an abstract specification language[15] to import data into the system. An extraction program consists of a set of commands of the form, [*variables, source, pattern*], that progressively refine strings to extract data. The source can be a URL to be retrieved, a scalar variable, or a slice of an array variable. To each member of the source, the pattern is applied, which consists of constant strings mixed with `*` (skip) and `#` (save) wildcards. The text matching each `#` wildcard in the pattern is stored in a corresponding output variable. Other possible patterns are the `split` pattern, which divides the source into an array of pieces according to a constant delimiter, and the `case` pattern, which specifies a set of pattern alternatives. Once extracted, the variables are packaged into a TSIMMIS result object and returned.

Editor[2] is a text manipulation language for the Araneus[3] project that uses word processor-like operations to rewrite pages. It contains primitives to select text regions using simple search patterns (consisting of constant strings and `*` wildcards) and to cut, copy, paste, and replace them. There is also a simple loop statement to iterate over all occurrences of a search pattern. Using these constructs, Editor programs can act as wrappers for web pages, but the language is extremely low-level and rather cumbersome.

Minerva[11] is a grammar-based extraction language that builds on Editor by using the latter as an exception handler. Programs in Minerva are regular (non-recursive) grammars that describe the structure of HTML documents. Rather than use a lexical tokenizer, Minerva expresses productions directly at the character level. Output is produced by a special nonterminal `$TP` that can be defined to emit captured values of other nonterminals whenever it is encountered by the parser. When declarative parsing of a production fails, an exception is thrown that propagates up the parse tree until a Editor exception handler is found. The procedural Editor code can perform rewrites on the values of all the nonterminals captured up to that point and emit a correct value for the failed nonterminal. It can also ask for modified strings to be reparsed by the Minerva grammar and then further edit the reparsed strings. In turn, the grammar parser can use special `$fail` productions to call Editor code.

Jedi[17] is an extraction language based on ambiguous context-free grammars. The basic code element is a production rule, which consists of a name, a production expression, and an optional code block. Production expressions can contain sequences, repetitions, and disjunctions of other production rules or terminal productions, which are regular expressions over ASCII characters. Within a production expression, subexpressions can have semantic predicates attached to them; such subexpressions will accept matching strings only if they also cause the associated predicate to succeed. The strings matched by subexpressions can also be stored or accumulated (for repetitions) in variables. The code block operates on these variables and returns some object as the value of the rule. Grammars can be ambiguous; for example, `([0-9]+ | .+)*`, which admits multiple parses of the same input. Jedi chooses a parse by using breadth-first search with pruning to find the most specific one. In this example, it will prefer to consider its input as a numeric string if it can; otherwise, it will treat it as a wildcard string.

A number of tools have also been developed that can generate web wrappers with varying degrees of automation; for a survey, see [21].

7.3. Comparison

WebL is probably the closest analogue to Webstream, although its text manipulations have less expressive power and it does not provide concurrency between downloading and processing. Few of the other languages offer the same combination of network management and HTML processing capabilities. Hippo and LogicWeb both provide concurrent network constructs, but offer little parsing support. Wexus is a more ambitious project involving truly distributed computation rather than just client-side web programming, although it would be possible to mimic Wexus' remote service requests in a full April+Webstream agent system. Golog and `<bigwig>` are both complementary to Webstream; the former operates on a semantic rather than syntactic level, while the latter operates on the server sider rather than the client side.

The extraction languages are much more narrowly focused than Webstream and do not provide network facilities at all. Even in HTML processing, they are much lower-level, operating at the character level rather than the tag level. TSIMMIS and Editor in particular are not very powerful in their search patterns.

Web libraries have also been developed for existing general-purpose languages. Compared to these libraries, Webstream generally provides a more concise and higher-level abstraction over web programming operations. For example, a C program using the `libwww` library takes dozens of lines just to initialize the interface and download one web page, where Webstream needs only a single pipeline statement. The web crawler program described in section 5 takes about 40 lines of code, excluding comments and blank lines, while a comparable Java program that performs the same task using the `java.net` and `javax.swing.text.html` libraries takes around 100 lines and requires careful synchronization coding. More importantly, these libraries do not have the capacity for agent programming integration that Webstream does, or the support for sequence and pattern matching on tags needed by information extraction applications.

8. CURRENT STATUS AND FUTURE WORK

The Webstream language is currently being developed under the GNU General Public License as part of the Network Agents project hosted on SourceForge. It can be accessed through CVS at `http://sourceforge.net/projects/networkagent/` or downloaded as a stable package from `http://www.doc.ic.ac.uk/~twh1/academic/#software`.

Webstream could be extended in several different ways. It would be useful in some applications to be able to rewind pipelines or access them in random-access mode as well as streaming mode. Better facilities for error handling and explicit specification of service constraints such as server timeouts are also desirable. In addition, there are various capabilities we would like to add to the pipeline components, such as the ability to set printf-like output specifications in `stringify` or to perform nested captures in `grep`. Finally, Webstream does not currently have a formal definition of its semantics.

9. CONCLUSION

Webstream is a new programming language designed to simplify the development of client-side web applications. With Webstream, it is easy to write both standalone web programs and web-capable intelligent agent applications, using only a small amount of code. The language increases the efficiency of web applications by encapsulating web documents in concurrent pipelines that can compute partial results lazy-evaluation style even before the documents have been completely downloaded. It provides a rich set of operators for parsing, transforming, and extracting web data that can be arbitrarily composed in simple yet powerful ways. Pipeline outputs can be collected in set abstractions that support querying by external agents using function closures. In addition, the language makes available all of the high-level symbolic programming capabilities of the underlying April agent programming language, such as pattern matching, function closures, and asynchronous message passing. Webstream thus provides a useful platform for easily constructing a wide variety of web applications, from web crawlers to persistent information agents.

REFERENCES

1. Agha G, Hewitt C. *Concurrent Programming using Actors.* MIT Press, Cambridge, MA, 1987.
2. Atzeni P, Mecca G. Cut and paste. *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 144–153, New York, 1997. ACM Press.

3. Atzeni P, Mecca G, Merialdo P. To weave the web. *VLDB '97, Proceedings of 23rd International Conference on Very Large Data Bases*, 206–215, San Mateo, CA, 1997. Morgan Kaufmann.
4. Berners-Lee T, Hendler J, Lassila O. The semantic web. *Scientific American*, May 2001, 2001.
5. Brabrand C, Møller A, Schwartzbach MI. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, May 2002.
6. Cardelli L, Davies R. Service combinators for web computing. Research Report 148, DEC SRC, June 1997.
7. Chawathe S, Garcia-Molina H, Hammer J, Ireland K, Papakonstantinou Y, Ullman J, Widom J. The TSIMMIS project: Integration of heterogenous information sources. *Proceedings of the 10th Meeting of the Information Processing Society of Japan (IPSJ '94)*, 7–18, 1994.
8. Clark KL, McCabe FG. April—agent process interaction language. *Intelligent Agents: ECAI '94 Workshop on Agent Theories, Architectures, and Languages*, 324–340, Berlin, 1995. LNAI 890, Springer-Verlag.
9. Connor R, Sibson K. HCL—a language for internet data acquisition. ICCL '98 Workshop on Internet Programming Languages, 1998.
10. Connor R, Sibson K, Manghi P. On the unification of persistent programming and the World-Wide Web. *The World Wide Web and Databases, International Workshop WebDB '98*, 34–51, Berlin, 1998. LNCS 1590, Springer-Verlag.
11. Crescenzi V, Mecca G. Grammars have exceptions. *Information Systems*, 23(8):539–565, 1998.
12. Davison A, Loke SW. A concurrent logic programming model of the web. Technical Report 98/28, Dept. of Computer Engineering, Prince of Songkla University, 1998.
13. FIPA. FIPA ACL message structure specification. Specification XC00061, Foundation for Intelligent Physical Agents, Concord, CA, 2001.
14. Foster I, Kesselman C, Tuecke S. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.
15. Hammer J, Garcia-Molina H, Cho J, Aranha R, Crespo A. Extracting semistructured information from the web. *PODS/SIGMOD '97 Workshop on Management of Semistructured Data*, New York, 1997. ACM Press.
16. Hong TW. *Grammatical Inference for Information Extraction and Visualisation on the Web.* PhD thesis, Imperial College London, London, United Kingdom, 2003.
17. Huck G, Fankhauser P, Aberer K, Neuhold E. Jedi: Extracting and synthesizing information from the web. *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS '98)*, 32–43, Los Alamitos, CA, 1998. IEEE Computer Society.
18. Hudak P. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, Dec. 1996.
19. Johnston WM, Hanna JRP, Millar RJ. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, Mar. 2004.
20. Kistler T, Marais H. WebL—a programming language for the web. *Computer Networks and ISDN Systems*, 30:259–270, 1998.
21. Laender AHF, Ribeiro-Neto BA, da Silva AS, Teixeira JS. A brief survey of web data extraction tools. *SIGMOD Record*, 31(2):84–93, Jun. 2002.
22. Levesque HJ, Reiter R, Lespérance Y, Lin F, Scherl RB. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, April–June 1997.
23. Loke SW, Davison A. LogicWeb: Enhancing the web with logic programming. *The Journal of Logic Programming*, 36:195–240, 1998.
24. McIlraith S, Son TC. Adapting Golog for programming the semantic web. *Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, 195–202, 2001.
25. Michaelides D, Moreau L, DeRoure D. A uniform approach to programming the world wide web. *Computer Systems Science and Engineering*, 14(2):69–81, 1999.
26. Tucker A, Noonan R. Event-driven programming. *Computer Science Handbook*, 2nd edition, CRC Press, Boca Raton, FL, 2004.