Hop, a Language for Programming the Web 2.0

Manuel Serrano
Inria Sophia Antipolis
INRIA Sophia Antipolis2004 route des Lucioles - BP 93F-06902 Sophia Antipolis, Cedex, France
http://www-sop.inria.fr/-members/Manuel.Serrano

Erick Gallesio
Université de Nice
Inria Sophia Antipolis
930 route des Colles, BP 145, F-06903 Sophia Antipolis, Cedex, France
http://www.essi.fr/~eg

Florian Loitsch
Inria Sophia Antipolis
INRIA Sophia Antipolis2004 route des Lucioles - BP 93F-06902 Sophia Antipolis, Cedex, France
http://www.inria.fr/mimosa/-Florian.Loitsch

Abstract

Hop is a new higher-order language designed for programming interactive web applications such as web agendas, web galleries, music players, etc. It exposes a programming model based on two computation levels. The first one is in charge of executing the logic of an application while the second one is in charge of executing the graphical user interface. Hop separates the logic and the graphical user interface but it packages them together and it supports strong collaboration between the two engines. The two execution flows communicate through function calls and event loops. Both ends can initiate communications.

The paper presents the main constructions of Hop. It sketches its implementation and it presents an example of a simple web application written in Hop.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages,, Concurrent, distributed, and parallel languages, Design languages

General Terms Design, Languages

Keywords Web programming, Functional programming

Download

Hop is available at: `http://hop.inria.fr`.

The web site contains the distribution of the source code, the online documentation, and various demonstrations.

1. Introduction

The recent evolution of the web makes it suitable for replacing traditional graphical user interfaces (henceforth GUIs). The combination of fast HTML rendering of modern web browsers (such as Gecko 20051111, shipped with Firefox 1.5), generalized support of CSS2 [15], yet expected to be rapidly supplanted by CSS3, and the recent adoption of asynchronous transactions (aka Ajax, the acronym of *Asynchronous JavaScript and XML*), makes web applications nearly as fancy and reactive as traditional GUIs. Some famous applications such as Google/mail, Google/map, or Zimbra's mailer demonstrate that web applications have bridged the gap with traditional GUIs.

In addition to allowing reactive and graphically pleasing interfaces, web applications are *de facto* distributed. Implementing an application with a web interface makes it instantly open to the world and accessible from much more than one computer. The web also partially solves the problem of platform compatibility because it physically separates the rendering engine from the computation engine. Therefore, the client does not have to make assumption on the server hardware configuration, and vice versa. Lastly, HTML is highly durable. While traditional graphical toolkits evolve continuously, obsoleting existing interfaces and breaking backward compatibility, modern web browsers that render on the edge web pages are still able to correctly display the web pages of the early 1990's.

For these reasons, the web is arguably ready to escape the beaten track of n-tiers applications, CGI scripting and interaction based on HTML forms. However, we think that it still lacks programming abstractions that minimize the overwhelming amount of technologies that need to be mastered when web programming is involved. As a step in this direction, we propose Hop, a higher order language aimed at programming interactive web applications. It is built on top of HTML, CSS, and JavaScript that are considered, in this work, as assembly languages.

1.1 The HOP programming language

Hop is mainly designed for programming small- to medium- sized *interactive applications* across the web. It is designed as a general-purpose web programming language which targets applications such as electronic agendas, photographs browsers, music players, mailer clients, operating system administration tools, and so on. In addition to enabling programming distributed applications over the web, Hop is also convenient for implementing applications that run on a single computer, on behalf of a single user. In that particular case, Hop is considered as a replacement for traditional graphical toolkits.

In contrast with most web-oriented languages and frameworks such as PHP and Ruby On Rails, the design of Hop is not database-centric. That is, while its standard library provides APIs for managing databases, Hop is not specially tuned for programming applications that access databases via the web. Hop is designed for programming various kinds of applications that need graphical user interfaces amongst which some might access databases.

Hop follows the path opened by Tcl/Tk, Java/Swing, or C/GTK+ but it differs from its ancestors by enforcing a strict separation between the programming of the interface from the programming of the logic of the applications. For that, it exposes a dual core execution model where one core executes the computations needed by the logic of the program while the second core executes the com-

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

putations needed by the graphical interface. We have deliberately provided Hop with a stratified language approach in order to emphasize the duality of these programs. However Hop tightly links the code of the interface and the code of the logic:

- it packages a whole application in a single location (e.g. a file).
- it supports function calls that traverse the strata.
- it supports data exchange between the strata.

Hop helps programming web applications because:

- it eases the deployment of applications by hiding URLs and by packaging the components of an application in a single place.
- it simplifies the control flow of the web application by allowing symmetric communications that can be initiated by both ends.
- it supports efficient event loops that avoid busy waiting.
- it eases the communication between servers and clients by supporting transparent function calls and partially shared name spaces.
- it provides a library of pre-defined widgets.
- it allows users to implement their own set of widgets that can be combined with the standard library for implementing complex GUIs.

1.2 Overview of the paper

The paper is organized as follows. Section 2 informally presents the stratified language design. Section 3 presents its syntax. The following Section 4 presents its semantics. It zooms in the function calls, and it presents the Hop's event loop. Section 5 presents an example of Hop programming. It shows a simplistic IMAP web client. Section 6 sketches the current implementation of Hop. Section 7 presents related work and envisions future work.

2. Overall language design

In this section, we present the rationale of Hop. We informally present its execution model and its syntax. This section only gives the intuition of what programming with Hop means. The technical presentations are left for Sections 3 and 4.

2.1 Rationale

Hop fosters a model where the *main* computation of an application is executed on a *server* and the graphical user interface is executed on remote *clients*. From the user point of view, a Hop program is executed within a web browser and it is associated with a well known URL. Once the program starts, the server and the clients continuously communicate. The exchanges are implemented by remote function calls and event loops. Hop is well suited for implementing applications that need frequent communications between the server and the client. For instance, we have implemented a music player with Hop that continuously displays, on the client, the elapsed time of the songs played on the server. On that particular application, as with many others we have implemented with Hop, the server and the client are frequently hosted by the same physical computer.

2.2 A dual core execution

A Hop program is executed simultaneously on several engines. The *main* engine is dedicated to executing the logic of the program. It executes CPU demanding computations and operations that require system privileges for accessing files or other resources. The other engines, henceforth called *GUI engines*, are dedicated to executing actions related to the programming of the graphical user interfaces. Engines are mapped to actual physical computers. More than one engine can be mapped to a single computer.

When a Hop program starts, it first executes on the main engine. This elaborates the description of a graphical user interface that is sent to a GUI engine. From that moment, the execution flows from the GUI engine to the main engine and vice versa. The GUI engines may invoke *services* from the main engine by the means of function calls. The main engine may *signal* events to the GUI engines. Each event carries an identity and a value. Events are handled asynchronously. They are used by the main engine to notify GUI engines when a new information is available. They are a means for implementing *pushing* on the web.

2.3 A dual language

Hop is a *stratified* language. The first stratum is dedicated to programming the *main* engines, or the servers. The second stratum is dedicated to programming graphical user interfaces, or the clients.

Both strata provide different facilities. On the one hand, the main stratum provides an API for accessing the file system and the other resources of the computer that hosts it, but it does not support any facility for handling graphical user interfaces. On the other hand, the GUI strata is provided with a full set of functionalities for dealing with graphical interactions but it has drastically restricted accesses to the resources of the computer it executes on. Because they are using different APIs, in general, an expression of the main stratum cannot be executed on the client and vice versa.

2.4 Objectives

As presented in Section 3, Hop uses a compact syntax that is close to the syntax used in traditional XML authoring. However, it is a complete programming language that subsumes many web technologies. As such, it allows the implementation of libraries that can be combined for implementing complex applications.

For the sake of the example and to give an intuition of what Hop programming looks like, here is a complete Hop program:

```
(let ((def (<DIV> ""))
      (svc (service (w)
              (<P> (sql-select
                    "FROM dict WHERE (def=~a)" w)))))
  (<HTML>
    (<BODY>
      (<TABLE>
        (<TR>
          (<TD> "search")
          (<TD> (<INPUT>
                   :type "text"
                   :onkeyup
                   ~(with-hop ($svc this.value)
                      (lambda (h)
                         (set! $def.innerHTML h))))))
        (<TR>
          (<TD> :colspan 2 def))))))
```

This program acts similarly to Google/suggest. The client displays an input box. It interactively reacts to key press events. Each time a new character is entered, the client invokes a service on the server which searches in a database the definition of the word sent by the client. On success, the definition is displayed back in the client display. The most important part of this example is the `with-hop` construction that invokes, from the client, a function located on the server. It is detailed in Section 4.3

3. The HOP syntax

This section presents the syntax of Hop. It first presents the syntax of the main stratum. Then, it presents the escape syntactic construction that switches to the GUI stratum. The formal definition of the syntax is given in the Appendix.

3.1 The syntax of the main stratum

At first glance, the syntax of the main stratum of Hop is a mere variation around HTML involving superficial modifications. It merely introduces an extra open parenthesis before any markup and replaces the closing markup with the single closing parenthesis. It also encloses string literals within " characters. Therefore, the HTML expression:

```
<HTML>
   <BODY>
      <B>A plain text</B>
   </BODY>
</HTML>
```

is written in Hop, as:

```
(<HTML>
   (<BODY>
      (<B> "A plain text")))
```

In Hop, attributes are introduced by an identifier starting with a colon character (`:`) and their value is separated from the name by white spaces. Hence the corresponding Hop program of the HTML document of Figure 1 is written as in Figure 2.

```
<HTML>
   <BODY>
      <TABLE width="100%">
         <TR> <TD>0</TD></TR>
         <TR> <TD>1</TD></TR>
         <TR> <TD>2</TD></TR>
         <TR> <TD>3</TD></TR>
      </TABLE>
   </BODY>
</HTML>
```

Figure 1. A simple HTML file.

In spite of the strong resemblance, there is a very important difference between the semantics of the two sources. While the HTML source can be interpreted as the external representation of a tree, the Hop source is actually a computer program that can be evaluated in order to produce a document. This is detailed in Section 4.

```
(<HTML>
   (<BODY>
      (<TABLE> :width "100%"
         (<TR> (<TD> 0))
         (<TR> (<TD> 1))
         (<TR> (<TD> 2))
         (<TR> (<TD> 3)))))
```

Figure 2. A simple HOP program.

3.2 The syntax of the GUI stratum

The GUI stratum is composed of *GUI expressions*. They are nested inside *main expressions*. The "`~`" character escapes from main expressions to GUI expressions. The GUI expressions are usually used as values of attributes, as can be seen in the following example:

```
(<HTML>
 (<BODY>
  (<BUTTON>
     :onclick ~(alert (* (Math.atan 1) 4))
     "Click me to see an approximation of PI")))
```

The character "`$`" escapes from the GUI stratum back to the main stratum. That is, it introduces an expression of the main stratum inside an expression of the GUI stratum. There is no limit to the nesting level so these main stratum expressions may, in turn, use the "`~`" character to escape back to the GUI stratum. For the sake of the example, let us consider a re-writing of the previous example where the approximation of π is moved to the main stratum:

```
(<HTML>
 (<BODY>
  (<BUTTON>
     :onclick ~(alert $(* 4 (atan 1)))
     "Click me to see an approximation of PI")))
```

Note that these two programs are likely to show different approximations since no provision is taken to guarantee that the precision of the arithmetic of the main stratum and the precision of the GUI stratum are the same.

4. The HOP dual evaluation

Hop brings abstraction to HTML. While HTML is a mere syntax that carries no semantics, Hop is a programming language. While a HTML expression denotes a *tree*, a Hop expression is evaluated in order to produce a *value*. While HTML markups are syntactic elements, in Hop, they are functions. More precisely, the meaning of the Hop expression:

(*fun* a_0 a_1...)

is the application of the function *fun* to the arguments a_0, a_1. Provided with this semantics, we can reconsider the previous Hop expression: `(<B> "A plain text")`
This expression is actually the call of the function `<B>` with the literal string `"A plain text"` as argument. It should be noted that Hop identifiers may use more characters than most programming languages. In particular, the characters `<` and `>` are legal identifiers characters, as letters, digits, `_`, `?`, `!`, and many others (see the Appendix).

Hop is unsurprisingly based on the Scheme algorithmic programming language [9] for which familiarity is assumed in the rest of this paper. Hop extends Scheme in many directions. It supports object-oriented programming, exceptions, modules, and multi-threading. It comes with various tools and libraries such as tools for constructing parsers and libraries for programming networks, multimedia applications, and so on. In addition to these features traditionally offered by programming languages, Hop supports original constructions specially designed for programming web applications. Since this paper focuses on web programming, it is intentionally shallow on the constructions that are not strictly related to this topic.

In the rest of this section, we present the Hop evaluation model. First, in Section 4.1 we present how a program is spawned. Then, in Section 4.2, we present how GUI expressions are built. Then, the Section 4.3 constitutes the heart of this paper. It presents how the two strata collaborate.

4.1 Dual Execution

The execution of a Hop program differs from the execution of traditional computer programs. In order to be executed, a Hop program has to be first loaded on a *HOP server*. This server conforms to the HTTP protocol [4]. It binds the program to an URL provided by the administrator of the server. This URL is used by clients (i.e. web browsers) to start the program.

A Hop program constructs a *response* to an HTTP request. In general, this response is a XML document but in some situations it can be any other data structures. For the sake of simplicity, in this paper, we focus on HTML responses only but all the presented techniques also directly apply to XML.

The execution of a Hop program is distributed. One part is executed on the *server* which evaluates the expressions of the main

stratum. The second part is executed on the *client* which evaluates the expressions of the GUI stratum. In general the execution flow switches from a server to a client and vice versa but Hop also allows two (or more) execution flows to run in parallel. The client communicates with the server via remote function calls. The server communicates with the client via signals. Both communication means allow compound values to be carried.

4.2 The Elaboration time

The purpose of most Hop programs is to build HTML pages that are visualized by web browsers. The phase of the execution on the server where the HTML pages are constructed is called *elaboration*. It takes place before any execution can start on the client.

Hop implements HTML pages as trees. The Hop libraries of the two strata provide functions for constructing and manipulating them. In both strata, the trees are first class values. Hence, they can be passed as argument to functions, returned as results, and stored into data structures and variables. For the sake of the example, Figure 3 is a re-writing of the Hop program presented Figure 2 where the four rows of the table are bound to local variables.

```
(let ((r0 (<TR> (<TD> 0)))
      (r1 (<TR> (<TD> 1)))
      (r2 (<TR> (<TD> 2)))
      (r3 (<TR> (<TD> 3))))
   (<HTML>
      (let ((table (<TABLE> r0 r1 r2 r3)))
         (<BODY> table))))
```

Figure 3. Using variables in HOP programs.

The interest of such an approach is better understood when some abstraction is used. In the example of Figure 4 a function is defined for automating the construction of the rows of the table.

```
(define (<ROW> v)
   (<TR> (<TD> v)))

(<HTML>
   (let ((table (<TABLE>
                  (<ROW> 0) (<ROW> 1)
                  (<ROW> 2) (<ROW> 3))))
      (<BODY> table)))
```

Figure 4. Using HOP naming conventions.

Note that the Hop convention is to surround the name of the functions that build HTML trees by the < and > characters and to use upper case letters. This example implicitly unveils that Hop standard HTML markups are implemented as regular functions. It also shows that defining a new markup in a user program is no more complex than defining a function.

As presented in Section 3.2, any expression of the main stratum can be nested in an expression of the GUI stratum after an escaping $ character. At elaboration time, the escaping main stratum expressions are evaluated and the resulting values are injected in the response. In consequence, when the response is shipped to the client it is totally stripped of main stratum expressions. Let's consider the Hop source of Figure 5 before elaboration.

Reflecting the two different execution times, the two strata use separate name spaces. A variable from the main stratum and a variable from the GUI stratum can hence hold the same name without conflicting. In other words, the variables defined line *1* and line *6* of Figure 5 are different. The elaboration phase replaces the occurrences of the variable x that belongs to the main stratum line *10* with the value "out" but it leaves the variable x that belongs to the GUI stratum line *9* as shown in Figure 6.

```
 1: (define x "out")
 2: (define y (vector 1 2 3))
 3:
 4: (<HTML>
 5:    (<BODY>
 6:       (<SCRIPT> ~(define x 0))
 7:       (<P> :onmouseover ~(begin
 8:                            (set! x (+ 1 x))
 9:                            (alert "over=" x))
10:            :onmouseout (alert $x)
11:            :onclick (alert $y)
12:            "foo")))
```

Figure 5. A program before elaboration.

```
 4: (<HTML>
 5:    (<BODY>
 6:       (<SCRIPT> ~(define x 0))
 7:       (<P> :onmouseover ~(begin
 8:                            (set! x (+ 1 x))
 9:                            (alert "over=" x))
10:            :onmouseout ~(alert "out")
11:            :onclick ~(alert '#(1 2 3))
12:            "foo")))
```

Figure 6. The program after elaboration.

As it can be seen here, the variable y which is bound to a Hop vector in Figure 5, line *2* is replaced with a constant vector in Figure 6, line *11*. This shows that the elaboration can inject complex data structures in the response. In particular it can inject tree branches that are under construction. This is illustrated by the next example that shows that an HTML tree can be used when constructing a document in the main Hop program (line *4* of Figure 7) *and* can also be injected in the GUI stratum (line *7*). The following example constructs an HTML page that swaps the two items of the unordered list each time the button is clicked.

```
 1: (let ((el (<UL> (<LI> "foo") (<LI> "bar"))))
 2:    (<HTML>
 3:       (<BODY>
 4:          el
 5:          (<BUTTON>
 6:             :onclick
 7:             ~(let* ((nodes (dom-child-nodes $el))
 8:                     (a (array-ref nodes 0))
 9:                     (b (array-ref nodes 1)))
10:                 (dom-append-child!
11:                  $el (dom-replace-child! a b)))
12:             "swap me"))))
```

Figure 7. Injecting a tree branch.

4.3 HOP services

This section presents one of the main technical novelty brought by Hop, namely the Hop remote services.

4.3.1 HOP service definition

HTML's URLs play a role similar to functions in programming languages. Let us consider the following HTML page (for simplicity, expressed in the Hop syntax):

```
(<HTML>
 ;; a link to google portal
 (<A> :href "http://www.google.com" "Google portal")
 ;; a google request
 (<FORM>
    :action "http://www.google.com/search"
    (<INPUT> :type "text" :name "q" :value "")
    (<INPUT> :type "submit" :value "search")))
```

In a plain HTML document, the URL `http://www.google.-com` could be considered as a function named `www.google.com` whose signature is:

`unit` ⟶ `HTMLtree`

It is called when a user clicks on the hyper link implemented by the `<A>` markup. Similarly, the URL `http://www.google.com/-search` denotes another function. It is called when the user clicks the `submit` button. This one accepts a parameter named `q` and its signature is hence:

`string` ⟶ `HTMLtree`

Hop transparently binds URLs to special functions called *services*. These reside on the server and they are called from the clients. They are defined by the form `define-service` whose syntax is:

```
(define-service (<ident> <ident>0 ...)
   <expression>)
```

<Ident> is the name of the service and *<ident>*$_0$, ... are its parameters. The form `define-service` is similar to the Scheme function definition form `define` but in addition to binding a function to an identifier in the server, it also binds it to an URL that can be used to run a Hop program. Let us consider the following example which is a complete Hop program:

```
1:  (define-service (portal)
2:    ;; a web page with a big lambda character
3:    (<HTML>
4:      (<BODY>
5:        (<CENTER>
6:          (<B> (<BIG> (<BIG> "&#955;")))))))
7:
8:  (define-service (rev q)
9:    ;; a web page with the argument reversed
10:   (<HTML>
11:     (<BODY>
12:       (<CENTER>
13:         (<B>
14:           (list->string
15:            (reverse (string->list q)))))))
16:
17: (<HTML>
18:   ;; a link to our Hop portal
19:   (<A> :href portal "portal")
20:   ;; a HOP request
21:   (<FORM>
22:     :action rev
23:     (<INPUT> :type "text" :name "q" :value "")
24:     (<INPUT> :type "submit" :value "reverse")))
```

Figure 8. A complete HOP program defining two services.

When this program is executed, it first binds two services: `portal` line *1* and `rev` line *8*. Then, line *17*, it elaborates an answer which is an HTML tree containing, line *19*, a call to the first service and, line *22*, a call to the second service.

Similarly to anonymous functions, Hop supports anonymous services which are not bound to any public URL. They are introduced by the form `service`:

```
(service (<ident>0 ...)
   <expression>)
```

Anonymous services are illustrated on the example presented in Figure 9. This Hop program manages a dynamic list of items. The form started at line *5* adds new entries. Clicking the submit button of line *8*, calls the anonymous service defined in line *6*. On the server, this service recursively calls the function `loop`, defined in line *1*, with the value of the input entry of line *9* added to the list.

```
1:  (let loop ((items (list "foo" "bar" "gee")))
2:    (<HTML>
3:     (<BODY>
4:      (<H3> "To do list")
5:       (<FORM>
6:         :action (service (new)
7:                    (loop (cons new items)))
8:         (<INPUT> :type "submit" :value "add")
9:         (<INPUT> :name "new" :type "text"))
10:      (<UL> (map <LI> items)))))
```

Figure 9. An example of anonymous services.

4.3.2 HOP service calls

The service calls presented in Section 4.3.1 suffer a strong restriction: they can only produce complete web pages. By the definition of HTTP and HTML they can hardly be used to compute partial results. In order to work around this limitation, web browsers have introduced a new communication means which enables clients to call services from servers and which enables clients to handle, as they wish, the results of the calls. The term *Ajax* has been coined for denoting programs using this capacity. This section, presents its support in Hop.

In addition to the `<A>` and `<FORM>` function invocations, any Hop service can be called, from the GUI stratum, with the following form:

```
(with-hop (service arg0 ...)
   [(lambda (h) ...success expression...)
    [(lambda (h) ...failure expression...)]])
```

The `with-hop` form calls the service *service* with the arguments, *arg*$_0$, ... When the call completes, on success, the optional GUI call-back procedure *success* is called. On failure, the optional call-back *failure* is called. Both call-back procedures accept one argument which is the result of the evaluation of the service on the main stratum. The example of Figure 10 shows an example of service call. In the GUI stratum, it invokes a service that returns the local date of the server which is displayed in a dialog box (line *7*).

```
1: (define-service (server-date)
2:    (current-date))
3:
4: (<HTML>
5:   (<BUTTON>
6:     :onclick ~(with-hop ($server-date)
7:                  (lambda (h) (alert h)))
8:      "Server time"))
```

Figure 10. An example of function call.

Compound data structure can transit from servers to clients and vice versa. The following example sends a list from the client to the server which, in turn, builds an HTML page containing a table and sends it back to the client. This new table replaces the initial empty element.

Because compound values can be exchanged, we could decide to modify the previous program in order to ship a list, from the server to the client, and construct the new HTML table in the client. This modification is presented Figure 12.

```
1:  (define-service (add10 lst)
2:     (<TABLE>
3:       (<TR>
4:          (map (lambda (e) (<TD> (+ 10 e))) lst))))
5:
6:  (<HTML>
7:    (<HEAD> (<HOP-HEAD>))
8:    (let ((el (<DIV> "")))
9:      (<BUTTON>
10:        :onclick ~(with-hop ($add10 (list 1 2 3))
11:                    (lambda (h)
12:                       (set! $el.innerHTML h)))
13:        el)))
```

Figure 11. Arguments of function calls.

```
1:  (define-service (add10 lst)
2:     (map (lambda (e) (+ 10 e)) lst))
3:
4:  (<HTML>
5:    (<HEAD> (<HOP-HEAD>))
6:    (let ((el (<DIV> "")))
7:      (<BUTTON>
8:        :onclick
9:        ~(with-hop ($add10 (list 1 2 3))
10:          (lambda (h)
11:             (dom-remove-child!
12:              $el (array-ref (dom-child-nodes $el) 0))
13:             (dom-append-child!
14:              $el (<TABLE> (<TR> (map <TD> h))))))
15:       el)))
```

Figure 12. Sending complex data structures.

In addition to enable communications from clients to servers the form `with-hop` can also be used to establish a communication between two servers. In that case, an extra parameter denoting the distant server is added. For instance, the following code can be used to fetch the date from a remote server:

```
(with-hop :host "http://remote.host:8080" ($server-date)
   (lambda (h) ...))
```

This example supposes that there is a Hop server listening to the socket port 8080 of the computer named `remote.host`. It also supposes that this server implements the service `server-date`. When one has to fetch information from a non Hop server, the form `with-url` can be used. It acts as `with-hop` except that it does not invoke a service on a distant server, it directly fetches the content of a document. Example:

```
(with-url "http://www.inria.fr/"
   (lambda (h) (xml-parse h ...)))
```

4.4 HOP Event loops

Hop provides two different kinds of event loops. The first ones are used to initiate, in the GUI stratum, computations at regular time intervals. Since, they are roughly equivalent to the JavaScript timer facilities, we only present them with the example shown Figure 13. This program polls every five seconds the server time which is updated on the client display.

Hop also provides an event mechanism which prevents client to busy wait. These events are first declared on the server, that is in the main stratum of a Hop program. In the GUI stratum, clients register to these events by the means of the dedicated markup <HOP-EVENT>. When a server emits an event, registered clients are notified. The implementation of <HOP-EVENT> liberates clients from checking periodically event notifications. This is, to our knowledge, another technical innovation brought by Hop.

```
1:  (<HTML>
2:    (<BODY>
3:      (let ((clock (<DIV> "")))
4:        (<TIMEOUT-EVENT>
5:           :timeout 5000
6:           :handler
7:           ~(with-hop ($(service () (current-date)))
8:               (lambda (h)
9:                  (set! $clock.innerHTML h)))
10:          clock))))
```

Figure 13. HOP timer loops.

Events are instances of the `hop-event` class. The function of the main stratum that signals an event has the following prototype:

```
signal-hop-event!: event × <value> ⟶ unit
```

The markup <HOP-EVENT> has the following shape:

```
(<HOP-EVENT>
   :event a-hop-event
   :handler a-client-code)
```

For the sake of the example, let us study a variation over the example of Figure 13. In this second version, the server initiates the communication with the client. That is, the client does not explicitly poll the server. It displays the server time when the server signals the event.

```
1:  (define evt
2:     (instantiate::hop-event
3:        (name "server time event")))
4:
5:  (thread-start!
6:   (make-thread
7:    (lambda ()
8:       (let loop ()
9:          (sleep! 5000)
10:         (signal-hop-event! evt (current-time))
11:         (loop)))))
12:
13: (<HTML>
14:   (<HEAD> (<HOP-HEAD>))
15:   (<BODY>
16:     (let ((clock (<DIV> "")))
17:       (<HOP-EVENT>
18:         :event evt
19:         :handler ~(set! $clock.innerHTML event)
20:         clock))))
```

Figure 14. HOP timer loops without client busy wait.

At Line *5* a thread is spawned on the server. This thread pauses during 5 seconds and then signals the event `evt` defined in line *1*. The form `instantiate` creates an instance of the class `hop-event`. The event is intercepted on the GUI stratum in line *19*. In the `:handler` block, following the JavaScript tradition of event handling, the variable `event` is automatically bound to the event that has been intercepted. The content of the `clock` <DIV> is replaced with the value carried by the event.

One may object that we have not eliminated the busy wait but moved it to the server. While, undubitably true, this is not a weakness of Hop. The point of this example is to show that *clients* may avoid busy waiting server events using the <HOP-EVENT> markups. It is up to the server to implement the appropriate signaling mechanism. The point of this section is only to show that the Hop notification implements a server *push* method.

5. Example

In this section, we show a small Hop application. We present an overly simplified IMAP client that uses a web interface. As shown in the screenshot of Figure 15, it presents a table with two columns. The left column displays the list of folders found on the IMAP servers. The upper right column shows the list of messages of the selected folder. The lower right column shows the body of the selected message. The application is interactive. Folders and messages are accessed on-demand using mouse clicks.

Hop provides a library for accessing IMAP servers. All the function that belong to this API are prefixed with `imap-`. This API being self explanatory, it is not discussed here. In order to make the application as compact as possible, we don't provide the IMAP client with a GUI for connecting to the IMAP server. Instead, the connection to the IMAP server is held in a global variable:

```
(define connection
   (imap-login
      (make-client-socket "imap.laposte.net" 993)
      "foo" "XXX"))
```

Then comes the heart of the application. It uses two Hop widgets, <TREE> and <PANED> which are popular widgets in traditional GUI programming and which are supported by Hop. This is presented in Figure 16.

The two <DIV>s, `folder` and `message` respectively contain the list of folders and the body of the selected message. The left column of the application is a tree whose label is the name of the IMAP server and whose leaves are labeled with the names of the folders.

```
(define (<IMAP-TREE> connection)
   (<TREE>
    (<TRHEAD>
      (socket-hostname connection))
    (<TRBODY>
      (map (lambda (f)
              (<TRLEAF>
               (<TT> :class "summary"
                 :onclick
                 ~(with-hop ($folder-summary $f)
     (lambda (h) (set! $folder.innerHTML h)))
                 f)))
           (imap-folders connection)))))
```

When a leaf of this tree is clicked-in, the service of the main stratum `folder-summary` is invoked with the name of the folder. The result of this function call fills the `folder` <DIV>. This is presented in Figure 17.

The service `folder-summary` builds a table with three columns containing respectively the subject, the author, and the emission date of the message. When such a message is clicked, the service `message-show` is called with two parameters, the folder name and the message identifier. The result is used to fill the `message` box.

```
3: (define message-show
4:    (service (folder msg)
5:       (imap-folder-select connection folder)
6:       (<PRE> (imap-message-body connection msg))))
```

6. Implementation

This section sketches the implementation of services and events. Readers not interested in such technical details may freely skip this section.

6.1 Implementation of Services

Hop is currently implemented as a web server. It accepts HTTP requests and it selects, according to the URL, the appropriate treatment to execute. Services are implemented as couples containing one function and one URL. That is, each time a service (anonymous or not) is created, a new unique URL is generated, a new function is created and the couple {*URL, function*} is stored inside a table on the server. When a request is intercepted, this table is scanned for selecting the appropriate service to execute. When a service is used in the GUI stratum, a reference to it is compiled to JavaScript. At last, expressions of the GUI stratum are compiled on the fly to JavaScript by a Hop-to-JavaScript compiler whose description is out of scope of this present paper. Let us assume the following Hop expression:

```
(<BUTTON>
    :onclick ~(with-hop ($(service (x y) (+ x y)) 1 2)
                        ...)
    "Click this link")
```

The elaboration yields the following HTML document:

```
<BUTTON onclick='with_hop(
  function() {
    return hop_service_url("/hop/???/4-57604278",
                           ["x", "y"],
                           arguments )
  }(1, 2), ...)'>
  "Click this link"
</BUTTON>
```

The dynamically created URL `/hop/???/4-57604278` is the unique identifier associated with the service. The current implementation of Hop is not able to reclaim these URLs. That is, it keeps alive for ever all the services and their URLs. It never collects URLs because we don't think that it exists a universal criterion allowing URL reclaim. Obviously, we could adopt a solution based on time stamping. For instance, we could arbitrarily decide to collect URL after two or three hours. We are reluctant to adopt such a solution but we are aware that this weakness has to be overcome in the future versions of Hop. We are investigating a solution where the server functions representing the services are encoded in the URL. More precisely according to this solution it would be no longer necessary to store the whole function on the server. Instead, only the closure environment would be encoded and sent to the client (i.e. the lexical environment active when the service was created). If this approach succeeds, it will remove the need for the table which binds URLs to functions.

The JavaScript functions `with_hop` and `hop_service_url` are defined in the standard Hop GUI stratum library. They are shipped with every generated page. As presented in Section 4.3.2 the form `with-hop` calls asynchronously a service. It can be defined as:

```
function with_hop( service, success, failure ) {
 function callback( h ) {
  if( h.status == 200 )
    if(h.getAllResponseHeaders().indexOf("hop-json")>=0)
      return success( eval( h.responseText ) );
    else
      return success( h.responseText );
  else
    return success( h );
 };

 return hop( service, callback, failure );
}
```

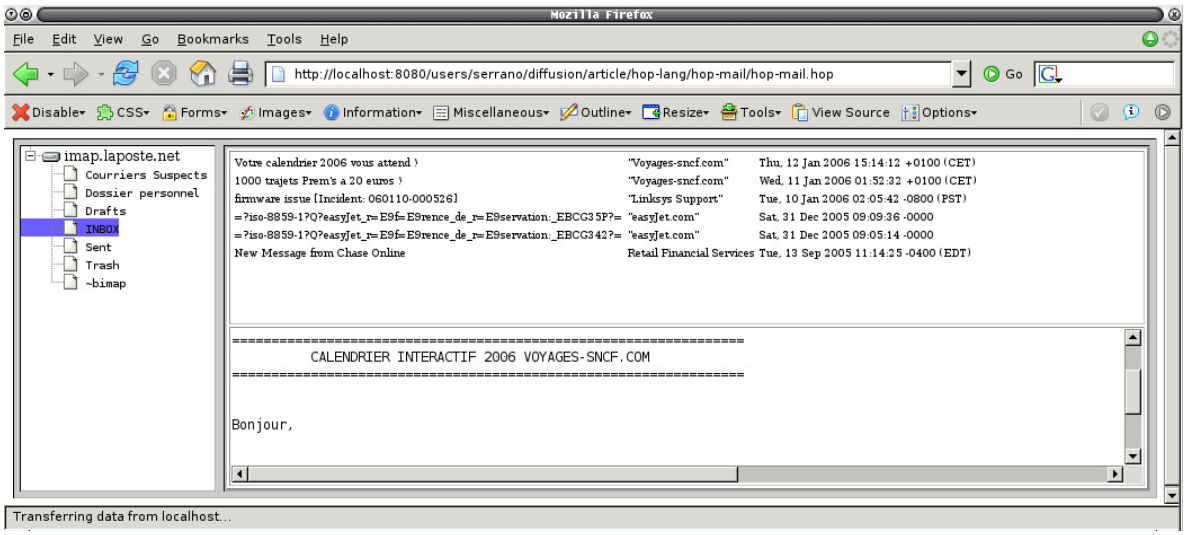

Figure 15. Webmail, a screenshot of the simple HOP webmail in Firefox.

```
(let ((folder (<DIV>))
      (message (<DIV>)))
   (define message-show ...)
   (define folder-summary ...)
   (define (<IMAP-TREE> connection) ...)
   (<HTML>
      (<HEAD>
         :css "hop-paned.css" :css "hop-tree.css" :css "hop-mail.css"
         :jscript "hop-paned.js" :jscript "hop-tree.js")
      (<BODY>
         (<PANED> :fraction 30
            (<PAN>
               (<IMAP-TREE> connection))
            (<PAN>
               (<TABLE> :width "100%" :border "2px"
                  (<TR> (<TD> :class "folder" :valign 'top folder))
                  (<TR> (<TD> :class "message" :valign 'top message))))))))
```

Figure 16. Webmail, main program.

```
(define folder-summary
   (service (folder)
      (imap-folder-select connection folder)
      (<TABLE> :class "summary"
        (map (lambda (mh)
               (<TR> :onclick ~(with-hop ($message-show $folder $(car mh))
                                 (lambda (h) (set! $message.innerHTML h)))
                 (<TD> (imap-header-get mh 'subject))
                 (<TD> (imap-header-get mh 'from))
                 (<TD> (imap-header-get mh 'date))))
             (imap-folder-headers connection)))))
```

Figure 17. Webmail, showing the messages of a folder.

The function hop_service_url maps a service to a URL. It could be implemented as:

```
function hop_service_url(service, formals, args) {
  var len = formals.length;
  var i;
  var url = service;

  if (len == 0) return service;
  for (i=0; i<len; i++)
    url+="&"+formals[i]+"="+hop_serialize(args[i]);
  return url;
}
```

The function hop_serialize marshals JavaScript values in a format compatible with the main stratum of Hop. When the server sends a value to the client which is not an HTML tree, it adds a hop-json header in the message and it uses the JSON external format that the client simply decodes with the JavaScript eval function call. The library function hop, whose code is not presented here, actually performs the JavaScript asynchronous call. It uses ad-hoc technics which are dependent of the clients web browsers.

6.2 Implementation of Events

The current implementation of the <HOP-EVENT> markup relies on services invocation. Waiting for an event is implemented as invoking an asynchronous service that returns only when the event is emitted from the client. More precisely, when an event e1 is created, the server automatically generates a service svc_{e1}. The markup <HOP-EVENT> is compiled, during the elaboration stage, as an invocation of svc_{e1} (see Section 4.3.2). This call completes when the server emits the signal e1. At that time, the client receives the value associated with the event. It re-invokes the service svc_{e1} for waiting for other values and, in parallel, it handles the received value. Assuming the library function hop_event whose definition looks like:

```
function hop_event( event, event_handler ) {
  var http = new XMLHttpRequest();
  var url = "hop_event_wait?event=" + event;

  http.open( "GET", url, true );
  http.onreadystatechange = function() {
    if( http.readyState == 4 && http.status == 200 ) {
       // invokes the user handler
       event_handler( http );
       // recursively call the function in
       // order to wait new results
       hop_event( event, event_handler );
    }
  }
  http.send( null );
}
```

The actual implementation takes care of portability issues. The expression (<HOP-EVENT> evt handler) is then compiled to:

```
hop_event( evt, handler );
```

This implementation prevents the client from busy waiting events for the server because invoking a service using an XMLHttpRequest is an asynchronous operation that does not involve polling. Surprisingly, this simple technique appears to be robust and it allows to implement passive wait on the client quite easily. It has been suggested by Marc Feeley who deserves most of its credit.

7. Discussion and Related work

In this section we discuss the design orientation of Hop and we present some related and future work.

7.1 Related work

Hop embraces in one unique language all the facets of web programming. It emphasizes compactness of programs and interactivity of applications. It proposes a lightweight approach based on functional programming. By contrast to previous studies, it does not enforce an interaction model based on forms submissions [6,7]. It follows an opposite direction to previous studies that aimed at easing CGI programming in traditional programming languages. The Meijer's CGI library [11] is a functional representative of this kind. Hop also distinguishes from early works such as the <bigwig> project [13,1] that are aimed for larger web applications where *sessions*, *database integration*, *security*, *static checking of dynamic web pages*, and *concurrency* are important issues. Hop proposes a solution for authoring web pages, programming the interactions between the servers and the clients, and programming the reactions of the user interfaces. In that sense, it is unrelated to languages such as Mozilla's XUL or Microsoft XAML that focus on the programming of the clients. XUL is the programming language of Mozilla which is, in addition to being a web browser, a whole execution environment. Hop is independent of any browser. It is used for implementing applications that need server- and client-programming. It can also be used for implementing applications that execute totally on a server. For instance, it can be used for implementing servers delivering music or for implementing tunneling via HTTP. This kind of applications is out of scope of languages such as XUL.

One of the closest works to Hop is due to Philip Wadler and his colleagues. The Links[1] programming language shares the goal of Hop. Like Hop, Links is a functional language that manages transparent function calls across the web. Like Hop, a Links program is made of a single source file and the client codes are compiled to JavaScript and server codes and client codes can be interleaved. Contrary to Hop, Links uses only one name space and functions are annotated with client or server marks depending on where they execute. This solution is elegant because it allows the use of only one single syntactic construction for calling either client functions and server functions. The Hop expression:

```
(with-hop ($lookup n) (lambda (lst) ...))
```

in Links is nicely written:

```
lst <- (lookup n); ...
```

However, we think that this approach has several weaknesses. First, we think that it is important to reflect, in the syntax, that calling a server function is a different operation from calling a client function. The two kinds of calls have totally different implementation and execution costs. Another weakness is that the Links approach gives the illusion that programming the client and programming the server is the same. However, the two execution engines cannot support the same set of operations. Some operations are meaningless on the server and vice versa (for instance, getting the dimension of the graphical user interface window is meaningless on the server). Next, we think that the elaboration stage of Hop that allows to *inject* server values in the client code is a strength. We hope that the various examples of the paper are demonstrative enough.

The last difference between Links and Hop is the event loops (see Section 4.4) that have no direct equivalent construction in Links. However, Links offers *client processes* that allow to establish asynchronous communications between the clients and the servers. Hence, it might probably be possible to implement Hop event loops on the top of Links processes.

7.1.1 Database orientation

Until recently, most of the web applications have adopted the same architecture. On the one hand, a server hosts a database and scripts that access it. On the other hand, a client, i.e. a browser, implements the user interface to this server. This architecture is so widely spread than many solutions have been conceived for easing the development and maintenance of such applications. In the first place, libraries have been developed. Windows ASP and Java JSP are two representatives of such libraries. Because a language tuned for a particular kind of applications might ease the development, lan-

[1] http://homepages.inf.ed.ac.uk/wadler/links.

guages such as PHP have appeared. A PHP program is composed of static HTML parts and PHP expressions that dynamically generate, on the server, new HTML nodes. Databases are tightly integrated in the language and in consequence, one might very concisely produce HTML documents from database queries.

Since the rise of Ajax applications, new solutions try to combine the compactness of PHP for programming databases accesses and the programming of reactive graphical interfaces. The largely advertised *Ruby On Rails* is one of these solutions: it defines itself as "a full-stack framework for developing *database-backed web applications* according to the Model-View-Control pattern". In a Rails application, the user interacts with an Ajax *view* of the database. Requests to this database are handled by a *controller* which is built on a *model* which dynamically maps the tables of the database to Ruby classes. This framework favors convention over configuration and by this way tries to eliminate as much as possible the need for hand-written code. Most of the code of a Rail application is automatically generated from templates and database introspection. Ruby on Rails goes beyond the objectives of a programming language and is particulary efficient for the kind of applications it is envisioned for.

Hop and Rails automatically generate URLs for server services. In Hop they are mapped to services (see Section 4.3). In Rails they are mapped to Ruby methods. Contrarily to Hop, Rails does not support direct parameters passing. Hop mainly eases the development of the communication components of reactive web applications. This facet is not addressed by the previously mentioned system.

7.1.2 Functional programming

Like Hop, WASH/CGI is a linguistic approach to programming the web. It concentrates on CGI programming so it addresses problems specific to this model such as persistency. Namely, WASH/CGI manages sessions. This problem is eliminated by the design of Hop which assumes an evaluator that is hosted by a full-fledged web server. Contrarily to CGI, the execution of a Hop application is not split into several execution chunks. WASH/CGI focuses on interactions based on HTML forms. In particular, inspired by Hanus' Curry [8], it automatically handles the `action` attributes of this HTML markup. That is, WASH/CGI replaces the URL of the HTML forms with an Haskell function. WASH/CGI transparently associates a private URL and it automatically feeds the function with the actual values found in the form. Hop's services can be viewed as a generalization of this approach because they can be used in forms but also everywhere where a reference to a server is used, in any JavaScript code, and in any event loop. Contrarily to Hop, WASH/CGI does not support remote service call nor does it support event loops and passive event waiting.

7.1.3 Sessions and Continuations

We have learned in the late 1990's from C. Queinnec, that most web applications have to deal with continuations. In his early publication [12] he has shown that a browser is a device that can call continuations multiply and simultaneously. Hence, he has concluded that an operator for capturing and restoring continuations is a natural tool of choice for implementing web pages. This point has been deeply developed and thoroughly studied by the PLT Scheme team in various publications [6]. Scheme is one of the seldom languages to support continuations. The `call/cc` (whose actual name is *call-with-current-continuation*) facility reifies a continuation into a function that can be called as any other function. Being a superset of Scheme [9] Hop supports `call/cc` so it naturally supports the programming model advocated by C. Queinnec. However, direct support for continuations as offered by `call/cc` is arguably too low level. Explicit manipulation of continuations can make programs very hard to understand, even for experts. So, we think that more restricted constructions that better fit the needs of the web programming should be studied in the spirit of the `send/suspend` and `send/finish` functions of the CONTINUE server [10].

The Smalltalk based Seaside framework[2] will be another source of inspiration for modeling and capturing the flow of control. This system decomposes web applications into stateful components. This successfully eliminates, from the source code, the burden of explicit continuations management.

Code mobility is a field that we are investigating. We are envisioning mobile code from server to client and vice versa. We are also considering mobility intra servers. In this approach, we could imagine spawning roaming agents processing remote data and carrying minimal sets of information. We could also imagine implementing load balancing of servers with mobility. Precisely, we are planning to adapt the technics developed by S. Epardaud [3] and Germain *et al.* [5] to Hop. In this context, the ideas of Obliq [2] or the technical solutions delivered by the ACUTE experiment [14] are likely to be useful.

8. Conclusion

Hop is a new computer language designed for programming web applications. It relies on two strata. The first one is used for programming the *server* side and for constructing graphical user interfaces. The second one is used for programming the animations of these interfaces and the interactions with users. The paper presents several examples of Hop programs. In particular, it shows the programming of a simplified IMAP client. This fifty lines long program displays interactively the messages stored on an IMAP server.

Hop abstracts many operations required by the web. So, for users not reluctant to functional programming, it makes the programming of these applications easier than most of the other languages we are aware of. Its main strengths are its ability to package a whole web application in a single bundle (e.g. a single file), its support for functions whose calls traverse the web, and its event notification mechanism. To our knowledge, Hop is one of the very first languages to propose a *global* solution to the web programming. Hop is one of the first language to support server *and* client programming, to manage communications initiated from both sides, and to support HTML authoring.

9. References

[1] Braband, C. and Møller, S. A. and Schwartzbach M.I., – **A runtime system for interactive Web services** – Journal of Computer Networks, 1999.

[2] Cardelli, L. – **Obliq A Language with Distributed Scope** – 122, Digital Equipment Corporation, Systems Research, Palo Alto, CA, 1994.

[3] Epardaud, S. – **Mobile Reactive Programming in ULM** – Utah, USA, Sep, 2004.

[4] Fielding, R. e. a. – **Hypertext Transfer Protocol** – RFC 2616, The Internet Society, , 1999.

[5] Germain, G. and Monnier, S. and Feeley, M. – **Termite: a Lisp for Distributed Computing** – 2nd European LISP and Scheme Workshop, Glasgow, UK, , 2005.

[6] Graunke, P. *et al.* – **Modeling Web Interactions** – European Symposium on Programming, Poland, 2003.

[7] Graunke, P. *et al.* – **Automatically restructuring programs for the Web** – Automated Software Engineering, 2004.

[8] Hanus, M. – **High-level server side Web scripting in Curry** – Practical Aspects of Declarative Languages, Las Vegas, NV, USA, 2001.

[9] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – Higher-Order and Symbolic Computation, 11(1), Sep, 1998.

[2] `http://www.seaside.st`.

[10] Krishnamurthi, S. – **The CONTINUE Server (or, How I Administrated PADL 2002 and 2003).** – Practical Aspects of Declarative Languages, New Orleans, LA, USA, Jan, 2003, pp. 2–16.
[11] Meijer, E. – **Server-Side web scripting in Haskell** – Journal of Functional Programming, 10(1), 2000.
[12] Queinnec, C. – **The influence of browsers on evaluators** – Int'l Conf. on Functional Programming, Montréal, Canada, Sep, 2000, pp. 23–33.
[13] Sandholm, A. and Schwartzbach, M. – **A type system for dynamic Web documents** – Symposium on Principles of Programming Languages, Boston, MA, USA, Jan, 2000, pp. 290–301.
[14] Sewell, P. e. a. – **Acute: High-level programming language design for distributed computation** – Int'l Conf. on Functional Programming, Tallinn, Estonia, Sep, 2005.
[15] World Wide Web Consortium, – **Cascading Style Sheets, level 2 CSS2 Specification** – REC-CSS2-19980512, W3C Recommendation, May, 1998.

Appendix

This appendix presents the syntax of Hop in EBNF form:

```
<comment> ⟶ ; <all subsequent characters up to a line break>

<expression> ⟶ <main-expression>

<main-expression> ⟶ <simple-expression>
  | ~ <gui-expression>

<gui-expression> ⟶ <simple-expression>
  | $ <main-expression>

<simple-expression> ⟶ <literal>
  | <identifier>
  | <attribute>
  | (<expression> <expression>*)

<identifier> ⟶ <initial> { <letter> | <digit> | <special> }
<initial> ⟶ <letter> | <special>
<letter> ⟶ a | b | ... | z | A | B | ... | Z
<digit> ⟶ 0 | 1 | ... | 9
<special> ⟶ _ | + | - | / | * | ? | > | < | = | ! | %
  | ~ | @ | ^ | & | \

<attribute> ⟶ : <identifier>

<literal> ⟶ <number>
  | <character>
  | <string>
  | <boolean>

<number> ⟶ <digit>+
  | <digit>+ . <digit>*
  | . <digit>+

<character> ⟶ #\ <any character>

<string> ⟶ " <string-element>* "
<string-element> ⟶ <any character other than " or \>
  | \" | \\
<boolean> ⟶ #t | #f
```