The HOP Development Kit

Manuel Serrano

Inria Sophia Antipolis
2004 route des Lucioles - BP 93 F-06902 Sophia Antipolis, Cedex, France
`http://www.inria.fr/mimosa/Manuel.Serrano`

ABSTRACT

Hop, is a language dedicated to programming reactive and dynamic applications on the web. It is meant for programming applications such as web agendas, web galleries, web mail clients, etc. While a previous paper (*Hop, a Language for Programming the Web 2.0*, available at `http://hop.inria.fr`) focused on the linguistic novelties brought by Hop, the present one focuses on its execution environment. That is, it presents Hop's user libraries, its extensions to the HTML-based standards, and its execution platform, the Hop web *broker*.

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

Along with games, multimedia applications, and email, the web has popularized computers in everybody's life. The revolution is engaged and we may be at the dawn of a new era of computing where the web is a central element.

Many of the computer programs we write, for professional purposes or for our own needs, are likely to extensively use the web. The web is a database. The web is an API. The web is a novel architecture. Therefore, it needs novel programming languages and novel programming environments. Hop is a step in this direction.

A previous paper [1] presented the Hop programming language. This present paper presents the Hop execution environment. The rest of this section presents the kind of end-user applications Hop focuses on (Section 1.1) and the technical solutions it promotes (Section 1.2). The rest of this paper assumes a familiarity with strict functional languages and with infix parenthetical syntaxes such as the ones found in Lisp and Scheme.

Because it is normal for a web application to access databases, manipulate multimedia documents (images, movies, and music), and parse files according to public formats, programming the web demands a lot of libraries. Even though it is still young, Hop provides many of them. In an attempt to avoid a desperately boring presentation this paper does not present them all! Only the library for building HTML graphical user interfaces is presented here. It is presented in Section 2, along with a presentation of the Hop solution for bringing abstraction to Cascade Style Sheets.

The section 3 focuses on programming the Hop web broker. Its presents basic handling of client requests and it presents the facilities for connecting two brokers and for gathering information scattered on the internet. The Section 4 presents the main functions of the broker programming library.

1.1 The web 2.0

In the newsgroup `comp.lang.functional`, a Usenet news group for computer scientists (if not *researchers* in computer science) someone reacted rather badly to the official announce of the availability of the first version Hop:

"I really don't understand why people are [so] hyped-up over Web 2.0. It's just Java reborn with a slower engine that doesn't even have sandboxing capabilities built into it. I guess this hype will taper off just like the Java hype, leaving us with yet another large technology and a few niches where it's useful."

This message implicitly compares two programming languages, namely Java and JavaScript and reduces Hop to *yet another general-purpose programming language*. This is a misunderstanding. The point of Hop is to help writing new applications that are nearly impossible (or at least, discouragingly tedious) to write using traditional programming languages such as Java and the like. As such, its goal is definitively *not* to compete with these languages.

As a challenge, imagine implementing a program that represents the user with a map of the United States of America that : lets the user zoom in and out on the map, and also helps with trip planning. In particular the user may click on two cities, and the application responds with the shortest route between the cities, the estimated trip time, the price of the gas for the trip (using local pump prices) the weather forecasts along the route (for the appropriate tires), and where to find the best pizza and gelatos in each town along the way. Although it is possible to write such a program using Java or C and existing resources available online, the web 2.0 is the infrastructure that makes it *feasible* to write such programs. Because the web 2.0 provides the potential to easily combine fancy graphics and information from disparate sources online into new, information-aware applications. Unfortunately, the programming model for the web 2.0 is missing. Hop is one attempt to provide the right model, and the rest of this paper explains how.

1.2 The HOP architecture

Hop enforces a programming model where the graphical user interface and the logic of an application are executed on two different engines. In theory, the execution happens as if the two engines are located on different computers even if they are actually frequently hosted by a single computer. In practice, executing a Hop application requires:

- A web browser that plays the role of the engine in charge of the graphical user interface. It is the *terminal* of the application. It establishes communications with the Hop *broker*.
- A Hop *broker* which is the execution engine of the application. All computations that involve resources of the local computer (CPU resource, storage devices, various multi-media devices, ...) are executed on the broker. The broker it also in charge of communicating with other Hop brokers or regular web servers in order to gather the information needed by the application.

The Hop programming language provides primitives for managing the distributed computing involved in a whole application. In particular, at the heart of this language, we find the `with-hop` form. Its syntax is:

```
(with-hop (service a₀ ..) callback)
```

Informally, its evaluation consists in invoking a remote *service*, i.e., a function hosted by a remote Hop broker, and, on completion, locally invoking the *callback*. The form `with-hop` can be used by engines executing graphical user interfaces in order to spawn computations on the engine in charge of the logic of the application. It can also be used from that engine in order to spawn computations on other remote computation engines.

2. Graphical User Interfaces

This section presents the support of Hop for graphical user interfaces. It presents the library of widgets supported by Hop and its proposal for bringing more abstraction to Cascade Style Sheets (CSS).

2.1 HOP Widgets

Graphical user interfaces are made of elementary graphical objects (generally named *widgets*). Each of these objects has its own graphical aspect and graphical behavior and it reacts to user interactions by intercepting mouse events and keyboard events. Hence, toolkits for implementing graphical user interfaces are characterized by:

1. the mechanisms for catching user interactions, and
2. the composition of graphical elements, and
3. the richness of them widgets.

HTML (either W3C's HTML-4 or XHTML-1) do a good job at handling events. Each HTML elements is *reactive* and JavaScript, the language used for programming events handlers, is adequate. CSS2, the HTML composition model based on boxes, is close to be sufficient. The few lacking facilities are up to be added to the third revision. On the other hand, the set of HTML widgets is poor. It mainly consists of boxes, texts, and buttons. This is insufficient if the web is considered for implementing modern graphical user interfaces. Indeed, these frequently use *sliders* for selecting integer values, *trees* for representing recursive data structures, *notepads* for compact representations of unrelated documents, and many others. HTML does not support these widgets and, even worse, since it is not a programming language, it does not allow user to implement their own complementary sets of widgets. Hop bridges this gap.

Hop proposes a set of widgets for easing the programming of graphical user interfaces. In particular, it proposes a *slider* widget for representing numerical values or enumerated sets. It proposes a WYSIWYG editor. It extends HTML tables for allowing automatic sorting of columns. It supports various *container* widgets such as a *pan* for splitting the screen in two horizontal or vertical re-sizable areas, a *notepad* widget for implementing *tab* elements, a *hopiwindow* that implements a window system in the browser, etc. In this paper, we focus on one widget that is representative of the container family, the *tree* widget.

2.1.1 The tree widget

A tree is a traditional widget that is frequently used for representing its eponymous data structure. For instance, it is extensively used for implementing file selectors. The syntax of Hop trees is given below. The meta elements required by the syntax are expressed using lower case letters and prefixed with the character `%`. The concrete markups only use upper case letters. The meta element `%markup` refers to the whole set of Hop markups.

```
%markup  ⟶ ... | %tree

%tree  ⟶  (<TREE> %tree-head %tree-body)
%tree-head  ⟶  (<TRHEAD> %markup)
%tree-body  ⟶  (<TRBODY> %leaf-or-tree*)
%leaf-or-tree  ⟶ %leaf | %tree
%leaf  ⟶  (<TRLEAF> %markup)
```

As an example, here is a simple tree.

```
(define (dir->tree dir)
  (<TREE>
    (<TRHEAD> dir)
    (<TRBODY>
      (map (lambda (f)
             (let ((p (make-file-name dir f)))
               (if (directory? p)
                   (dir->tree p)
                   (<TRLEAF> :value qf f))))
           (directory->list dir)))))
```

When an expression such as `(dir->tree "/")` is evaluated on the broker, a tree widget representing the hierarchy of the broker files is built. It has to be sent to a client for rendering.

Hop containers (i.e., widgets that contain other widgets) are *static*, as in the example above, or *dynamic*. A static container builds its content only once. A dynamic container rebuilds its content each time it has to be displayed. A static tree has a fixed set of subtrees and leaves. A dynamic tree recomputes them each time unfolded. A dynamic tree is characterized by the use of the `<DELAY>` markup in its body. The syntax of this new markup is:

```
(<DELAY> thunk)
```

The argument `thunk` is a procedure of no argument. Evaluating a `<DELAY>` form on the Hop broker installs an anonymous service whose body is the application of this `thunk`. When the client, i.e., a web browser, unfolds a dynamic tree, its invokes the service associated with the `thunk` on the broker. This produces a new tree that is sent back to the client and inserted in the initial tree.

```
(define (dir->dyntree dir)
  (<TREE>
    (<TRHEAD> dir)
    (<TRBODY>
      (<DELAY>
        (lambda ()
          (map (lambda (f)
                 (let ((p (make-file-name dir f)))
                   (if (directory? p)
                       (dir->dyntree p)
                       (<TRLEAF> :value qf f))))
               (directory->list dir)))))))
```

Even if the function `dir->dyntree` only differs from `dir->tree` by the use of the `<DELAY>` markup, its execution is dramatically different. When the expression `(dir->dyntree "/")` is evaluated, the broker no longer traverses its entire hierarchy of files. It only inspects the files located in the directory `"/"`. When the client, i.e., a web browser, unfolds a node representing a directory, the broker traverses only that directory for scanning the files. Contrary to `dir->tree`, the directories associated with nodes that are never unfolded are never scanned by `dir->dyntree`.

2.1.2 Extending existing HTML markups

Because Hop is not HTML it is very tempting to add some HTML facilities to Hop, for instance by adding new attributes to markups. In order to keep the learning curve as low as possible, we resist this temptation. Hop offers the HTML markups as is, with on exception: the `<IMG>` markup. In HTML, this markup has a `src` attribute that specifies the actual implementation of the image. It can be an URL or an in-line encoding of the image. In that case, the image is represented by a string whose first part is the declaration of a mime type and the second part a row sequence of characters representing the encoding (e.g., a *base64* encoding of the bytes of the image). While this representation is close to impractical for a hand-written HTML documents, it is easy to produce for automatically generated documents, such as the ones produced by Hop. Hop adds a new attribute `inline` to HTML images. When this attribute is set to `#t` (the representation of the value *true* in the concrete Hop syntax), the image is encoded on the fly.

This tiny modification to HTML illustrates why a programming language can dramatically help releasing documents to the web. Thanks to this `inline` attribute, it is now easy to produce stand alone HTML files. This eliminates the burden of packaging HTML documents with external tools such as `tar` or `zip`.

2.2 HOP Cascade Style Sheets

Cascading Style Sheets (CSS) enable graphical customizations of HTML documents. A CSS specifies rendering information for visualizing HTML documents on computer screens, printing them on paper, or even pronouncing them on aural devices. A CSS uses selectors to designate the elements onto which a customization applies. Attributes, which are associated with selectors, specify the rendering information. The set of possible rendering attributes is rich. CSS exposes layout principles based on horizontal and vertical boxes in the spirit of traditional text processing applications. CSS version 2 suffers limitations (for instance, it only supports one column layout) that are to be overcome by CSS version 3. CSS is so expressive that we think that when CSS v3 is fully supported by web browsers, HTML will compete with text processors like Latex for printing high quality documents.

CSS selectors are expressed in a little language. The elements to which a rendering attribute applies are designed either by their identities, their classes, their local or global positions in the HTML tree, and their attributes. The language of selectors is expressive but complex, even if not Turing-complete. On the one hand, the identity and class designations are suggestive of object-oriented programming. On the other hand, they do not support inheritance. Implementing re-usable, compact, and easy-to-understand CSS is a challenging task. Frequently the HTML documents have to be modified in order to best fit the CSS model. For instance, dummy `<DIV>` or `<SPAN>` HTML elements have to be introduced in order to ease the CSS selection specification. We think that this complexity is a drawback of CSS, and Hop offers an improvement.

Like the Hop programming language, Hop-CSS (HSS in short) uses a stratified language approach. HSS extends CSS in one direction: it enables embedding, inside standard CSS specifications, Hop expressions. The CSS syntax is extended with a new construction. Inside a HSS specification, the `$` character escapes from CSS and switches to Hop. This simple stratification enables arbitrary Hop expressions to be embedded in CSS specifications. We have found this extension to be useful to avoiding repeating constants. For instance, instead of duplicating a color specification in many attributes, it is convenient to declare a variable holding the color value and use that variable in the CSS. That is, the traditional CSS:

```
button {
  border: 2px inset #555;
}
span.button {
  border: 2px inset #555;
}
```

in Hop can be re-written as:

```
$(define border-button-spec "2px inset #555")

button {
  border: $border-button-spec;
}
span.button {
  border: $border-button-spec;
}
```

In other situations, the computation power of Hop significantly helps the CSS specifications. As an example, imagine a graphical specification for 3-dimensional borders. Given a base color, a 3-dimensional inset border is implemented by lightening the top and left borders and darkening the bottom and right borders. Using the two Hop library functions `color-ligher` and `color-darker` this can be implemented as:

```
$(define base-color "#555")

button {
  border-top: 1px solid $(color-lighter base-color);
  border-left: 1px solid $(color-lighter base-color);
  border-bottom: 1px solid $(color-darker base-color);
  border-right: 1px solid $(color-darker base-color);
}
```

The specification of the buttons border is actually a compound property made of four attributes. It might be convenient to bind these four attributes to a unique Hop variable. Since the HSS `$` escape character enables to inject compound expressions, this can be wriiten as:

```
$(define base-color "#555")
$(define button-border
   (let ((c1 (color-lighter base-color))
         (c2 (color-darker base-color)))
      { border-top: 1px solid $c1;
        border-left: 1px solid $c2;
        border-bottom: 1px solid $c2;
        border-right: 1px solid $c1 }))

button {
   $button-border;
}
```

3. Programming the HOP web broker

The Hop web broker implements the execution engine of an application. While the client executes in a sandbox, the broker has privileged accesses to the resources of the computer it execution on. As a consequence, the client has to delegate to the broker the operations it is not allowed to execute by itself. These operations might be reading a file, executing a CPU-intensive operation, or collecting information from another remote Hop broker or from a remote web server. In that respect, a Hop broker is more than a web server because it may act has a client itself for handling external requests. Still, a Hop broker resembles a web server. In particular, it conforms to the HTTP protocol for handling clients connections and requests. When a client request is parsed, the broker elaborates a response. This process is described in the next sections.

3.1 Requests to Responses

Clients send HTTP messages to Hop brokers that parse the messages and build objects representing these requests. For each such objects, a broker elaborates a response. Programming a broker means adding new rules for constructing responses. These rules are implemented as functions accepting requests. On return, they either produce a new request or a response. The algorithm for constructing the responses associated with requests is defined as follows.

```
(define (request->response req rules)
   (if (null? rules)
       (default-response-rule req)
       (let ((n ((car rules) req)))
          (cond
             ((is-response? n)
              n)
             ((is-request? n)
              (request->response n (cdr rules)))
             (else
              (request->response req (cdr rules)))))))
```

The *else* branch of the conditional is used when no rule applies. It allows rules to be built using `when` and `unless`, without having to be a series of nested `if`s.

A rule may produce a response. In that case, the algorithm returns that value. A rule may also annotate a request or build a new request from the original one. In that case, the algorithm applies the remaining rules to that new request.

The default response rule, which is used when no other rule matches, is specified in the configuration files of the broker.

3.2 Producing responses

The broker has to serve various kind of responses. Some responses involve local operations (such as serving a file located on the disk of the computer where the broker executes). Some other responses involve fetching information from the internet. Hop proposes several type of responses that correspond to the various ways it may fulfill client requests.

From a programmer's point of view, responses are represented by subclasses of the abstract class `%http-response`. Hop proposes an extensive set of pre-declared response classes. The most important ones are presented in the rest of this section. Of course, user programs may also provide new response classes.

3.2.1 No response!

Responses instance of the class `http-response-abort` are actually *no response*. These objects are used to prevent the broker for answering unauthorized accesses. For instance, on may wish to prevent the broker for serving requests originated from a remote host. For that, he should had a rule that returns an instance of `http-response-abort` for such requests.

Hop provides predicates that return true if and only if a request comes from the local host. Hence, implementing remote host access restriction can be programmed as follows.

```
(hop-add-rule!
 (lambda (req)
    (if (is-request-local? req)
        req
        (instantiate::http-response-abort))))
```

3.2.2 Serving files

The class `http-response-file` is used for responding files. It is used for serving requests that involve static documents (static HTML documents, cascade style sheets, etc.). It declares the field `path` which is used to denote the file to be served. In general these responses are produced by rules equivalent to the following one.

```
(hop-add-rule!
 (lambda (req)
    (if (and (is-request-local? req)
             (file-exists? (request-path req)))
        (instantiate::http-response-file
           (path (request-path req))))))
```

In order to serve `http-response-file` responses, the broker reads the characters from the disk and transmit them to the client via a socket. Some operating systems (such as Linux 2.4 and higher) propose system calls for implementing this operation efficiently. This liberates the application from explicitly reading and writing the characters of the file. With exactly one system call, the whole file is read and written to a socket. For this, Hop uses subclasses of `http-response-file`.

The class `http-response-shoutcast` is one of them. It is used for serving music files according to the shoutcast protocol[1]. This protocol adds meta-information such as the name of the music, the author, etc., to the music broadcasting. When a client is ready for receiving shoutcast information, it must add an `icy-metadata` attribute to the header of its requests. Hence, in order to activate shoutcasting on the broker one may use a rule similar to the following one.

```
(hop-add-rule!
 (lambda (req)
    (if (and (is-request-local? req)
              (file-exists? (request-path req)))
        (if (is-request-header? req 'icy-metadata)
            (instantiate::http-response-shoutcast
               (path (request-path req)))
            (instantiate::http-response-file
               (path (request-path req)))))))
```

Note that since the rules scanned in the inverse order of the their declaration, the shoutcast rule must be added after the rule for regular files.

3.2.3 Serving dynamic content

Hop provides several classes for serving dynamic content. The first one, `http-response-procedure`, is used for sending content that varies for each request. The instances of that class carry a procedure that is invoked each time the response is served. In the example above, we add a rule that create a *virtual* URL `/count` that returns the value of an incremented counter each time visited.

```
(let ((count 0)
      (resp (instantiate::http-response-procedure
               (proc (lambda (op)
                        (set! count (+ 1 count))
                        (printf op
                                "<HTML>~a</HTML>"
                                count))))))
   (hop-add-rule!
    (lambda (req)
       (when (and (is-request-local? req)
                  (string=? (request-path req) "/count"))
          resp))))
```

3.2.4 Serving data

Hop programs construct HTML documents on the server. On demand they are served to clients. These responses are implemented

[1] http://www.shoutcast.com/.

by the http-response-hop class. When served, the XML tree inside a response of this type is traversed and sent to the client. As an example, consider a rule that adds the URL /fact to the broker. That rule computes a HTML table filled with factorial numbers.

```
(hop-add-rule!
 (lambda (req)
    (when (and (is-request-local? req)
               (string=? (request-path req) "/fact"))
       (instantiate::http-response-hop
          (xml (<TABLE>
                  (map (lambda (n)
                          (<TR>
                             (<TH> n)
                             (<TD> (fact n))))
                       (iota 10 1))))))))
```

Instead of always computing factorial value from 1 to 10, it is easy to modify the rule for adding a range.

```
(hop-add-rule!
 (lambda (req)
    (when (and (is-request-local? req)
               (substring? (request-path req) "/fact/"))
       (let ((m (string->integer
                 (basename (request-path req)))))
          (instantiate::http-response-hop
             (xml (<TABLE>
                     (map (lambda (n)
                             (<TR>
                                (<TH> n)
                                (<TD> (fact n))))
                          (iota m 1)))))))))
```

Next, we now show how to modify the rule above so that the computation of the HTML representation of the factorial table is moved from the broker to the client. As presented in Section 1.2, the Hop programming language supports the form with-hop. This invokes a service on the broker and applies, on the client, a callback with the value produced by the service. This value might be an HTML fragment or another Hop value. On the server, HTML fragments are represented by responses of the class http-response-hop. The other values are represented by the class http-response-js. When such a response is served to the client, the value is serialized on the broker according to the JSON format[2] and unserialized on the client. We can re-write the previous factorial example in order to move the computation of the HTML table from the broker to the client. For that, we create a rule that returns the factorial values in a list.

```
(hop-add-rule!
 (lambda (req)
    (when (and (is-request-local? req)
               (substring? (request-path req) "/fact/"))
       (let ((m (string->integer
                 (basename (request-path req)))))
          (instantiate::http-response-js
             (value (map (lambda (n)
                               (cons n (fact n)))
                         (iota m 1))))))))
```

The /fact URL can be used in client code as follows.

[2] http://www.json.org/.

```
(with-hop "/hop/fact/10"
   (lambda (l)
      (<TABLE>
         (map (lambda (p)
                 (<TR>
                    (<TH> (car p))
                    (<TD> (cdr p))))
              l))))
```

The point of this last example is not to argue in favor of moving this particular computation from the broker to the client. It is just to show how these moves can be programmed with Hop.

3.2.5 Serving remote documents

Hop can also act as a web proxy. In that case, it intercepts requests for remote hosts with which it establishes connections. It reads the data from those hosts and sends them back to its clients. The class http-response-remote represents such a request.

In order to let Hop act as a proxy, one simply adds a rule similar to the one below.

```
(hop-add-rule!
 (lambda (req)
    (unless (is-request-local? req)
       (instantiate::http-response-remote
          (host (request-host req))
          (port (request-port req))
          (path (request-path req))))))
```

This rule is a good candidate for acting as the *default* rule presented in Section 3.1. The actual Hop distribution uses a default rule almost similar to this one. It only differs from this code by returning an instance of the http-response-string class for denoting a *404 error* when the requests refer to local files.

3.2.6 Serving strings of characters

Some requests call for simple responses. For instance when a request refers to an non existing resource, a simple error code must be served to the client. The class http-response-string plays this role. It is used to send a return code and, optionally, a message, back to the client.

The example below uses a http-response-string to redirect a client. From time to time, Google uses *bouncing* which is a technique that allows them to log requests. That is, when Google serves a request, instead of returning a list of found URLs, it returns a list of URLs pointing to Google, each of these URL containing a forward pointer to the actual URL. Hence Google links look like:

```
http://www.google.com/url?q=www.inria.fr
```

When Hop is configured for acting as a proxy it can be used to avoid this bouncing. A simple rule may redirect the client to the actual URL.

```
(hop-add-rule!
 (lambda (req)
    (when (and (string=? (request-host req)
                         "www.google.com")
               (substring? (request-path req) "/url" 0))
       (let ((q (cgi-fetch-arg "q" path)))
          (instantiate::http-response-string
             (start-line "HTTP/1.0 301 Moved Permanently")
             (header (list (cons 'location: q))))))))
```

A similar technique can be used for implementing blacklisting. When configured as web proxy, Hop can be used to ban ads contained in HTML pages. For this, let us assume a black list of domain names held in a hash table loaded on the broker. The rule above prevents pages from these domains to be served. It lets the client believe that ads pages do not exist.

```
(hop-add-rule!
 (lambda (req)
    (when (hashtable-get *blacklist* (request-host req))
       (instantiate::http-response-string
          (start-line "HTTP/1.0 404 Not Found")))))
```

3.3 Broker hooks

When a response is generated by the algorithm presented in Section 3.1 and using the rules of Section 3.2 the broker is ready to fulfill a client request. Prior to sending the characters composing the answer, the broker still offers an opportunity to apply programmable actions to the generated request. That is, before sending the response, the broker applies *hooks*. A hook is a function that might be used for applying security checks, for authenticating requests or for logging transactions.

A hook is a procedure of two arguments: a request and a response. It may modify the response (for instance, for adding extra header fields) or it may return a new response. In the following example, a hook is used to restrict the access of the files of the directory /tmp.

```
(hop-hook-add!
 (lambda (req resp)
  (if (substring? (request-path req) "/tmp/")
   (let ((auth (get-request-header req 'authorization)))
    (if (file-access-denied? auth "/tmp")
     (instantiate::http-response-authentication
       (header '("WWW-Authenticate: Basic realm=Hop"))
       (body (format "Authentication required.")))
     resp))
   resp)))
```

When a request refers to a file located in the directory /tmp, the hook presented above forces Hop to check if that request is authenticated (a request is authenticated when it contains a header field authorization with correct values). When the authentication succeeds, the file is served. Otherwise, a request for authentication is sent back to the client.

4. The HOP library

The Hop standard library provides APIs for graphical user interfaces, for enabling communication between the clients and the broker, for decoding standards documents formats (e.g., EXIF for jpeg pictures, ID3 for mp3 music, XML, HTML, RSS, ...). It also offers APIs for enabling communications between two brokers and between brokers and regular web servers. Since the communication between two brokers is similar to the communication between clients and brokers (see the form with-hop presented Section 1.2), it is not presented here. In this section we focus on the communications between brokers and regular web servers.

The Hop library provides facilities for dealing with low-level network communications by the means of sockets. While this is powerful and enables all kind of communications it is generally tedious to use. In order to remove this burden from programmers, Hop provides two high-level constructions: the <INLINE> markup and the with-url form.

4.1 The <INLINE> markup

The <INLINE> markup lets a document embed subparts of another remote document. When the broker sends a HTML tree to a client, it *resolves* its <INLINE> nodes. That is, it opens communication with the remote hosts denoted to by the <INLINE> nodes, it parses the received documents and it includes these subtrees to the response sent to its client.

The <INLINE> node accepts two options. The first one, :src, is mandatory. It specifies the URL of the remote host. The example below builds a HTML tree reporting information about the current version of the Linux kernel. This information is fetched directly from the kernel home page. It is contained in an element whose identifier is versions.

```
(<HTML>
  (<BODY>
    "The current Linux kernel versions are:"
    (let ((d (<INLINE> :src "http://www.kernel.org")))
       (dom-get-element-by-id d "versions"))))
```

This program fetches the entire kernel home page. From that document it extracts the node named versions. The second option of the <INLINE> node allows a simplification of the code by automatically isolating one node of the remote document. The :id option restricts the inclusion, inside the client response, to one element whose identifier is :id. Using this second option, our program can be simplified as shown below.

```
(<HTML>
  (<BODY>
     "The current Linux kernel versions are:"
     (<INLINE> :src "http://www.kernel.org"
               :id "versions")))
```

In addition to be more compact, this version is also more efficient because it does not require the entire remote document to be loaded on the broker. As it receives characters from the network connection, the broker parses the document. As soon as it has parsed a node whose identifier is versions it closes the connection.

4.2 The with-url form

The syntax of the form with-url is as follows:

```
(with-url url callback)
```

Informally, its evaluation consists in fetching a remote document from the web and on completion, invoking the *callback* with the read characters as argument. Unlike to the <INLINE> node, the characters do not need to conform any particular syntax. More precisely, the fetched document does not necessarily need to be a valid XML document. In the example below, we show how the with-url form can be used to implement a simple RSS reader.

The function rss-parse provided by the standard Hop library parses a string of characters according to the RSS grammar. It accepts four arguments, the string to be parsed and three constructors. The first and seconds build a data structure representing RSS sections. The last one builds data structures for representing RSS entries.

```
(define (make-rss channel items)
   (<TREE>
      channel
      (<TRBODY> items)))

(define (make-channel channel)
   (<TRHEAD> channel))
```

```
(define (make-item link title date subject descr)
   (<TRLEAF>
      (<DIV>
         :class "entry"
         (<A> :href link title)
         (if date (list "(" date ")"))
         (if subject (<I> subject))
         descr)))
```

Once provided with the tree constructors, parsing RSS documents is straightforward.

```
(define (rss->html url)
   (with-url url
      (lambda (h)
         (rss-parse h make-rss make-channel make-item))))
```

Producing a RSS report is then as simple as:

```
(rss->html "kernel.org/kdist/rss.xml")
```

5. Conclusion

Hop is a programming language dedicated to programming interactive web applications. It differs from general purpose programming languages by providing support for dealing with programs whose execution is split across two computers. One computer is in charge of executing the logic of the application. The other one is in charge of dealing with the interaction with users.

This article focuses on the Hop development kit. It presents some extensions to HTML that enable fancy graphical user interfaces programming and it presents the Hop web broker programming. In the presentation various examples are presented. In particular, the paper shows how to implement simple a RSS reader with Hop in no more than 20 lines of code!

The Hop library is still missing important features for web programming. In particular, it does not provide SOAP interface, it cannot handle secure HTTPS connections, and it does not implement graphical visual effects. We continue to work on Hop, however, and would love your feedback.

6. References

[1] Serrano, M. and Gallesio, E. and Loitsch, F. – **HOP, a language for programming the Web 2.0** – Proceedings of the First Dynamic Languages Symposium, Portland, Oregon, USA, Oct, 2006.

Acknowledgments

I would like to thanks Robby Findler for his invitation to the Scheme workshop and for his extremely helpful comments on the paper.