Multitier programming in Hop

A first step toward programming 21st-century applications

Manuel Serrano and Gérard Berry, INRIA

The Web is becoming the richest platform on which to create computer applications. Its power comes from three elements: (1) modern Web browsers enable highly sophisticated GUIs with 3D, multimedia, fancy typesetting, etc.; (2) calling existing services through Web APIs makes it possible to develop sophisticated applications from independently available components; and (3) open data availability allows applications to access a wide set of information that was unreachable or that simply did not exist before. The combination of these three elements has already given birth to revolutionary applications such as Google Maps, radio podcasts, and social networks.

The next step is likely to be incorporating the physical environment into the Web. Recent electronic devices are equipped with various sensors (GPS, cameras, microphones, metal detectors, speech command sensitivity, thermometers, motion detection, etc.) and communication means (IP stack, telephony, SMS, Bluetooth, etc.), which enable applications to interact with the real world. Web browsers integrate these features one after the other, making the Web runtime environment richer every day. The future is appealing, but one difficulty remains: current programming methods and languages are not ideally suited for implementing rich Web applications. This is not surprising as most were invented in the 20th century, before the Web became what it is now.

Traditional programming languages have trouble dealing with the asymmetric client-server architecture of Web applications. Ensuring the semantic coherence of distributed client-server execution is challenging, and traditional languages lack transparent support for physical distribution. Thus, programmers need to master a complex gymnastics for handling distributed applications, most often using different languages for clients and servers. JavaScript is the dominant Web language but was conceived as a browser-only client language. Servers are usually programmed with quite different languages such as Java, PHP, and Ruby. Recent experiments such as Node.js propose using JavaScript on the server, which makes the development more coherent; however, harmonious composition of independent components is still not ensured.

In 2006, three different projects—namely, GWT (Google Web Toolkit), Links from the University of Edinburgh,¹ and Hop from INRIA (<http://www.inria.fr>)⁵—offered alternative methods for programming Web applications. They all proposed that a Web application should be programmed as a single code for the server and the client, and written in a single unified language. This principle is known as *multitier programming*.

Links is an experimental language in which the server holds no state, and functions can be symmetrically called from both sides, allowing them to be declared on either the server or the client. These features are definitely interesting for exploring new programming ideas, but they are difficult to implement efficiently, making the platform difficult to use for realistic applications.

GWT is more pragmatic. It maps traditional Java programming into the Web. A GWT program looks like a traditional Java/Swing program compiled to Java bytecode for the server side and to

JavaScript for the client side. Java cannot be considered the unique language of GWT, however. Calling external APIs relies on JavaScript inclusion in Java extensions. GUIs are based on static components declared in external HTML files and on dynamic parts generated by the client-side execution. Thus, at least Java, JavaScript, and HTML are directly involved.

The Hop language takes another path, relying on a different idea: incorporating all the required Web-related features into a single language with a single homogeneous development and execution platform, uniformly covering all the aspects of a Web application: client side, server side, communication, and access to third-party resources. Hop embodies and generalizes both HTML and JavaScript functionalities in a Scheme-based³ platform that also provides the user with a fully general algorithmic language. Web services and APIs can be used as easily as standard library functions, whether on the server side or the client side.

MULTITIER PROGRAMMING AND HTML ABSTRACTION

This article presents the Hop approach to Web application programming, starting with the basic example of a Web file viewer. In the first version, files and directories are listed on a bare page.

FIGURE 1 A file browser Web Widget

```

1: (define-tag <BROWSER> ((id (gensym))           ;; HTML identifier
2:                      (class #f)              ;; HTML class
3:                      (path::bstring "/tmp"))  ;; default initial path
4:   (define (<dir-entry> path)
5:     ;; function to build HTML display for directories
6:     (<DIV> :data-hss-tag "dir-entry"
7:       :onclick ~(with-hop ($(service ()) (<directory> path)))
8:       (lambda (h)
9:         (innerHTML-set! $id h)))
10:    name))
11:   (define (<file-entry> path)
12:     ;; function to build HTML display for files
13:     (<DIV> :data-hss-tag "file-entry"
14:       (basename path)))
15:   (define (<directory> path)
16:     ;; main display function
17:     (cons
18:       ;; HTML representation of the parent directory
19:       (<dir-entry> (make-file-name path ".."))
20:       ;; HTML representation of files
21:       (map (lambda (np)
22:             (if (directory? np)
23:                 (<dir-entry> np)
24:                 (<file-entry> np)))
25:         (sort string<? (directory->path-list path))))))
26:   ;; main HTML element
27:   (<DIV> :data-hss-tag "browser"
28:     :id id :class class
29:     (<directory> path)))

```

Directories are clickable to enable dynamic browsing, and plain file names are displayed. This simple problem illustrates the most central aspect of realistic Web applications—namely, the tight collaboration and synchronization of servers and clients. Here, the server owns the files while the browser visualizes them. When the user clicks on a directory, the client requests new information from the server.

Upon receipt, the client updates the page accordingly. The file viewer is easy to implement if each request delivers a new complete page. By adding the requirement that the file viewer should be a widget embedded in a complex Web page, the problem becomes slightly more difficult, since updates now concern only incremental subparts of the documents. This demands a new kind of collaboration between the server and the client.

Implementing such a Web client-server file viewer with a single Hop program is almost as straightforward as implementing it for a single computer, because HTML is built in and execution partitioning between servers and clients is automatic. Figure 1 shows a complete Hop solution to this problem.

Hop identifiers can contain special characters such as “<” and “?”. In Hop, `directory?`, `string=?`, and `<SPAN>` are legal identifiers. In the example, `<BROWSER>` is the name of a variable bound to a function that creates an HTML `div` element.

HTML objects are created by Hop functions with the same names (`<DIV>...`), (`<SPAN> ...`), etc.; no HTML closure is needed because the code deals with parenthetical functional expressions instead of texts. The full power of Scheme is used to build `DIVs` and `SPANS`. The `<dir-entry>` and `<file-entry>` auxiliary functions are used to build fragments of the final HTML document. The `map` operation in the main `<directory>` function is used to build the global HTML page out of these HTML fragments, with help from the Hop `sort` function to sort the printed output alphabetically. Figure 2 shows how values produced by the `<div-entry>` function are compiled into HTML and JavaScript.

The `~` and `$` constructs appearing at line 7 of Figure 1 are the multitier programming operators. Let us explain how they work. Expressions prefixed by `~` are client-side expressions evaluated by the browser; unannotated expressions are evaluated on the server. Code is compiled on the server. A client code is seen as a value, computed or elaborated by the server and automatically shipped to the client on demand. The `$` construct is used to inject a server value within the client

FIGURE 2

Hop compiled into HTML+JavaScript

```
onclick='HOP.with_hop( "/hop/anonymous314",
function( h ) {
  var _el3 = document.getElementById( "g1592" );
  _el3.innerHTML = HOP.eval( h );
} )'
/tmp
</div>
```

code at elaboration time. For example, `~(alert $(hostname))` prints the server name on the client screen. In the example of figure 1, line 7 provokes the injection of an automatically generated service named `anonymous314` in the HTML excerpt of figure 2. The `~-`-prefixed expression at line 7 specifies the action to be executed by the client when the user clicks on a directory. The action invokes the anonymous service, called by the client and run on the server.

Runtime communication between servers and clients involves services that extend the notion of a function. A *service* is the binding of a function to an URL. The URL is used to invoke the function remotely, using the special form `(with-hop (<service> <arguments>) <callback>)`. The arguments are marshaled for network transmission and the remote procedure call is initiated; on remote-call completion, the callback is invoked on the caller side with the unmarshaled remote-call result. Extra options control how errors and timeouts are handled (these are not presented in this article).

The example here contains one anonymous service defined in the `<dir-entry>` auxiliary function. This service is invoked each time a user clicks on a directory. It calls the `<directory>` function that traverses the server file system. Because services are statically scoped just like functions, the `<directory>` identifier in the service code refers to the `<directory>` server function defined later in the same mutually recursive define clause. The `path` parameter is bound by the definition of the `<dir-entry>` function and used in the service call from within the client code. Lexical binding is the same for the server and client code.

Once the `<BROWSER>` function is defined, it can be freely used as a new HTML tag in any regular HTML element within Hop. For example, the code shown in figure 3 instantiates two file browsers in a table.

Making variations on `<BROWSER>` that affect both the server and the client is very easy. Figure 4 shows a one-line modification of the `<file-entry>` function body that displays the file name and size.

Hop relies on a mixed dynamic/static type discipline. When a variable or a function is annotated with a type, the compiler uses this type to statically check type correctness and improve the generated code. To detect compile-time errors, Hop does not compete with statically typed languages such as ML or Haskell but trades static guarantees for expressiveness: for example, it supports programming constructs that are out of reach of statically typed languages, such as CLOS-like

FIGURE 3

A Web file browser

```
(<HTML>
  (<TABLE>
    (<TR>
      (<TD> (<BROWSER> :class "usr" :path "/usr"))
      (<TD> (<BROWSER> :class "local" :path "/usr/local")))))
```

object-oriented programming style. Also note that the children of HTML nodes can be of any type (a list, string, number, another node, etc.) without requiring type annotations, as specified by the W3C recommendation of XML Schema for HTML. In the example, only the path variable is type-annotated.

A Hop widget may declare its own CSS (Cascading Style Sheets) rules to abstract away its implementation details. The declaration in figure 5 creates a new CSS type associated with the `<BROWSER>` tag. Figure 6 shows how one can use the browser CSS type to add extra icons before directory and leaf nodes.

In Hop, HTML objects on the server are members of a full-fledged data structure that implements the abstract syntax tree of the client HTML document; this contrasts with most Web environments, in which they are reduced to bare strings of characters. A Hop server computation elaborates a DOM (document object model) representation of HTML similar to the one used by the browser, compiles it on the fly to actual HTML, and ships it on browser demand.

This multistage process eliminates several drawbacks of traditional Web frameworks. First, the Hop runtime environment guarantees several properties of the generated HTML such as syntactic correctness—for example, it reports HTML tag misspellings as unbound-variable errors. Second, a single document can be automatically compiled into different HTML versions on demand. The same document can be optionally compiled into a mix of HTML4 and Flash for old browsers, into HTML5 for more skilled browsers, or even into XHTML+RDFa for semantic annotations.

FIGURE 4

A file browser modified for file size

```
(define (<file-entry> path)
  (<DIV> :data-hss-tag "file-entry"
    (basename path)
    (<SPAN> (round (file-size path) 1024)))) ;; <-- added line for size
```

FIGURE 5

Extending CSS with a new type and new attributes

```
$(define-hss-type browser "div[data-hss-tag=browser]"
  (define-hss-property (directory-image v)
    (format "div[data-hss-tag=dir-entry] {
      background-image: ~l;
      background-position: left;
      background-repeat: no-repeat;
```

Most current client-side Web libraries abstract over HTML by providing a set of JavaScript functions that accept regular HTML nodes as parameters; typically, `DIV` and `SPAN` are used as new widget containers. This approach has several drawbacks. First, HTML extensions do not look like regular tags. The implementation of a GUI thus requires assembling different formalisms. Second, extension initializations are complex to schedule. In general, the application must resort to window `onload` JavaScript events to ensure that the API constructors are not called before the DOM tree is fully built. Third, to configure the graphical rendering of the extension, implementation details must be unveiled to let programs designate the HTML elements to be configured. This jeopardizes maintainability.

CODE REUSE

The example file-viewer application presented here involves the main ingredients of a Web application—a distributed architecture, HTML abstraction, and CSS configurations—but it is still too simple to be realistic. To develop richer Web applications with comparatively little effort, it is now common practice to rely on the wide set of publicly available JavaScript APIs. All Web frameworks offer one way or another of using them, often through backdoors that let the programmer insert alien JavaScript calls into the natively supported language.

Hop follows a different approach by integrating these APIs into its language. Calling a JavaScript function or creating a JavaScript object from Hop is as easy as manipulating a standard Hop entity. Furthermore, once compiled, client-side Hop-generated JavaScript can be distinguished from handwritten JavaScript only by the name mangling used to map Hop identifiers to JavaScript identifiers. Everything else is identical: Hop data, variables, and functions are directly mapped to their JavaScript counterparts.

The direct integration of JavaScript APIs within Hop makes Web-platform development by component combination easy. To illustrate API integration, we show how to write a PDF viewer that displays the contents of files with the nice visual effect of turning the pages of an actual book. Figure 7 shows a snapshot of this application, and figure 8 displays the actual source code. It is based on jQuery, a popular JavaScript library; a jQuery plug-in called `turn.js` (<http://www.turnjs.com>),

CSS user rules

```
browser {
  directory-image: url( http://nowhere.org/dir.png );
  leaf-image: url( http://nowhere.org/leaf.png );
}

browser.blue {
  directory-image: url( http://nowhere.org/blue/dir.png );
  color: blue;
  font-weight: bold;
}
```

FIGURE 7

A PDF viewer in a Web browser

FIGURE 8

A PDF viewer invoking two JavaScript APIs

```

(<HTML>
  (<HEAD>
    ;; Third-party JavaScript libraries
    :jscript "http://code.jquery.com/jquery-1.7.1.min.js"
    :jscript "http://www.turnjs.com/turn.min.js"
    ;; Hop bindings for external JavaScript
    :include "pdfjs-api-1.0.0.hz")
  (<DIV>
    ~(<add-event-listener! window "load"
      (lambda ()
        ;; initialize the turnjs plugin when the client document is ready
        (let ((bk (jQuery "#book"))) ;; direct use of a jQuery function
          (bk.turn))) ;; direct use of a turnjs method
    (<DIV> :id "book"
      (map (lambda (i)
        ;; create one HTML element per PDF page
        (<PDF> :src (make-file-path REPOSITORY "queue12.pdf") :page i))
        (iota 10 1))))))

```

which implements the page-turning effect; and the JavaScript PDF previewer implemented by the Mozilla Foundation (<https://github.com/andreasgal/pdf.js>). It is assumed that the PDF previewer is packaged in the same way as the <BROWSER> first presented. In the code, the method `turn` initializes

the turn.js plug-in. It prepares an HTML element for use as a book container. This JavaScript method is directly used as a regular Hop function.

The turn.js API automatically invokes JavaScript listeners before and after page turns. This feature can generate page content on demand on the server. Figure 9 presents a complete example where the client requests page contents from the server right before turning a page. This example uses the turn.js method `bind` to bind an anonymous Hop function to user events, in this case associating a user

FIGURE 9

A dynamically generated book using a JavaScript API

```
(define-service (filebook count)

  (define cache
    ;; a local cache to avoid computing several times the same page
    (make-vector count #f))

  (define getpage
    ;; the service that returns the pages content
    (service (i)
      (unless (vector-ref cache i)
        ;; not in the cache, fill the page with a random quotation
        (vector-set! cache (system->string "fortune")))
      ;; returns the cache entry
      (<BLOCKQUOTE> (vector-ref cache i))))

  (<HTML>
    (<HEAD>
      :jscript "http://code.jquery.com/jquery-1.7.1.min.js"
      :jscript "http://www.turnjs.com/turn.min.js"
      :css "filebook.hss")

    ~ (define (turnpage i)
      ;; function called for each visualized page
      (with-hop ($getpage i)
        (lambda (h)
          ;; insert result of the getpage call in the i-th div
          (innerHTML-set! (format "page-~a" i) h))))

    ~ (add-event-listener! window "load"
      ;; initialize the turn plugin when the client document is ready
      (lambda (_)
        (let ((el (jQuery "#dir")))
          (el.turn)
          (el.bind "turning"
            (lambda (_ page)
              ;; function invoked by the turnjs plugin when
              ;; the user clicks on a next/prev page button
              (turnpage page)
              (turnpage (+ 1 page)))))))

    (<DIV> :id "dir"
      ;; create one empty div per document page; these divs are
      ;; filled by the client-side turnpage function
      (map (lambda (i) (<DIV> :id (format "page-~a" i) ""))
        (iota count))))
```


function with page-turn events. This shows that Hop functions can be invoked directly by JavaScript as regular JavaScript functions.

No platform other than the Web has ever let anyone write sophisticated GUIs so simply, by combining APIs and codes provided by different parties.

OPEN DATA

Governments and public organizations have recently understood that the data they generate is a valuable asset. New companies such as Data Publica were created with the sole purpose of collecting, organizing, and redistributing open data.

Traditionally, open data was exogenous to the Web. For example, the French government agency INSEE (National Institute of Statistics and Economic Studies; <http://www.insee.fr>), created in the middle of the 20th century, is in charge of conducting all sorts of statistical analyses of the population and economy. INSEE has become an important open data provider.

Surprisingly, the data might be endogenous, too. Google's flu trends analysis is a remarkable example of this kind. Google researchers have observed that the number and locations of search requests about flu are strongly related to the propagation of the illness.² Google makes this data available country by country, making it easy for anyone to implement applications using these statistics.

In this section we demonstrate how to create a video showing the propagation of the flu over time. This is mostly a tool-combination problem since all the hard work can be delegated to external parties. Thus, the work shown here consists only of collecting, combining, and reusing all the tools

FIGURE 10

Visualizing flu propagation worldwide

```
(module flutrend
  (import (gchart "gchart-api-1.0.*.hz")))

(<HTML>

  ~ (define sources
      $(map (lambda (data) (google-chart :type 'map :data data))
        (with-input-from-file "http://www.google.org/flutrends/data.txt"
          (lambda (p)
            ;; skip legal Google notice
            (read-lines p 11)
            ;; parse the actual flu trends values
            (read-csv-records p))))))

  (<IMG> :src (google-chart :type 'map :area 'europe)
    :onload ~ (when (pair? sources)
      (after 300
        (lambda ()
          (set! this.src (car sources))
          (set! sources (cdr sources)))))))
```

and data at our disposal. The actual Hop implementation, shown in figure 10, contains no more than 15 lines of code.

The application code relies on a binding that makes the Google Chart API directly accessible from Hop. This Web API creates all kinds of charts and geographical maps, generating images according to the arguments specified on appropriate URLs. This API is combined with the extraction of flu statistics. First, image URLs are generated out of statistical values parsed using a Hop spreadsheet-elements library. Then images generated on the fly by Google Chart are displayed one after the other, every 300 milliseconds.

This example shows that new and deep knowledge might emerge when the ability to collect, compare, and analyze vast sets of open data is easily accessible. Hop is specifically geared toward this objective, with composition and reuse as the two central features.

BIGGER THAN THE WEB

The human Web can be made even bigger by binding sensors from and actuators to the physical environment, such as those provided by smart-phone sensors, multimedia equipment, home or automotive automation equipment, and the countless other electronic devices that surround us. Only the most popular of these will be natively supported by the next Web browsers, however. For example, the upcoming versions of the Firefox and Chrome navigators have announced support for video and audio capture; accessing other remote equipment through browsers or programmed applications will still require third-party support. The Hop philosophy is to migrate the remote interface facilities to the server, give the user full Hop programming power on the associated data, and deliver results to clients in the standard way when using browser-based interfaces.

Figure 11 shows how to implement a Hop application that integrates features provided by a mobile phone inside a Web application. The contents of incoming SMSs are displayed in a browser window, letting users choose whether each SMS should be spoken aloud.

This application uses three separated tiers: a Web browser, a Web server, and a phone server. The phone server is responsible for receiving the SMSs and speaking them on demand. The Web server plays the role of orchestrator; when the phone notifies it that a new SMS has been received, it alerts the interested clients and, depending on the user configuration, reads the message aloud.

The application takes advantage of another useful feature of Hop: allowing a server to be a new source of program-generated Web events for clients or other servers. In the phone example, the server starts establishing a bridge with the phone (using the `android` variable), then waits for the notification of an SMS and accesses the phone text-to-speech facility (using the `tts` variable).

The server code registers a function invoked each time a new SMS is received to forward a software Web event to interested Web clients.

IMPLEMENTATION

Hop is an open source project whose source code can be downloaded from the Web site <http://hop.inria.fr>. All the actual source code shown in this article can also be downloaded from this Web site. The development kit contains the compilers, interactive documentation, various tools for creating and installing applications, and the runtime environment, which is a dedicated Web server that runs on all Un*x platforms—Linux, Mac OS X, and Android. It is operational on mainstream modern architectures such as x86/32, x86/64, PowerPC, and the ARM family.

The Hop runtime environment requires only 3-4 MB of RAM. This small memory footprint makes it suitable for smart phones or even smaller machines, such as the new ARM-equipped computers (Phidget SBC2, Raspberry Pi, etc.) that offer only a few megabytes to applications. As for speed, the Hop Web server delivers dynamic Web pages significantly faster than traditional servers such as Apache or Lighthttpd. The performance gain is a result of the Hop-in-Hop implementation, which enables the server to respond to requests involving dynamic computations without costly execution context switching.⁴

A NEW PLAYGROUND

Hop is the conjunction of a multitier programming language and a runtime environment for the Web. Its main goals are to unify the linguistic features needed for Web applications, to automate the

FIGURE 11

Reacting to SMS

```
(module androidemo
  (library mail phone hopdroid))

(define android (instantiate::androidphone))
(define tts (instantiate::androidtts (phone android)))

(define-service (phone/sms)

  (define speak #t)

  (add-event-listener! android "sms-received"
    (lambda (e::event)
      ;; function invoked on the server (i.e., the phone)
      ;; upon sms-reception
      (let ((val e.value))
        ;; broadcast to all clients the reception of the sms
        (hop-event-broadcast! "sms" val)
        ;; speak out the sms content on the phone
        (when speak (tts-speak tts (cadr val))))))

  ;; create two empty HTML elements that will be filled
  ;; upon SMS reception
  (let ((sms-num (<TD>))
        (sms-body (<TD>)))
    (<HTML>
      ~ (add-event-listener! server "sms"
        ;; function invoked on the client (i.e., a web browser)
        ;; upon reception of the server "sms" event
        (lambda (e)
          ;; insert the calling number in the HTML page
          (innerHTML-set! $sms-num (car e.value))
          ;; insert the SMS content in the HTML page
          (innerHTML-set! $sms-body (cadr e.value))))
      (<TABLE> (<TR> sms-num sms-body))
      (<BUTTON> :onclick ~(with-hop ($ (service () (set! speak (not speak)))))
        "mute/speak"))))
```

physical distribution of code in a fully transparent way, and to help programmers reuse and combine external resources. All of these are keys to program simplicity and conciseness.

This article has presented several complete Web application examples written in Hop. The first example illustrates how to raise the HTML abstraction level by using an algorithmic programming language approach. The second example shows how to reuse third-party code. The third example shows that combining Web technologies with open data opens the path for new application ideas. The last example presents a simple diffuse application on the Web. These examples have all been chosen for their simplicity; the same technique would be equally effective when dealing with much bigger applications in domains such as home automation or multimedia.

None of the applications presented here exceeds 30 lines of Hop source code. This is not accidental; it is the consequence of a perception shift where the Web is no longer regarded as a mere platform for sharing documents but as a revolutionary runtime environment. Note, however, that Hop is a realistic programming language, which implies a lot of additional bells and whistles. This article has ignored most of them, concentrating on multitier programming and the connection to official Web technologies.

Finally, note that the Hop core language is a strict functional programming language, with runtime type checking. This design choice comes mostly from a personal bias of the first author. Arguably, it fits well with the current programming trends of the Web, but other flavors should be possible, yielding other programming languages. We hope to see such new languages appearing, because the multiplicity of approaches will foster creativity and possibly open a new era in language and tool design.

REFERENCES

1. Cooper, E., Lindley, S., Wadler, P., Yallop, J. 2006. Links: Web programming without tiers. Presented at the 5th International Symposium on Formal Methods for Components and Objects.
2. Ginsberg, J., Mohebbi, M., Patel, R., Brammer, L., Smolinski, M., Brilliant, L. 2009. Detecting influenza epidemics using search engine query data. *Nature* 457 (February 19): 1012-1014.
3. Kelsey, R., Clinger, W., Rees, J. 1998. The revised (5) report on the algorithmic language scheme. *Higher-Order and Symbolic Computation* 11(1); <http://www-sop.inria.fr/indes/fp/Bigloo/doc/r5rs.html>.
4. Serrano, M. 2009. Hop, a fast server for the diffuse Web. In *Proceedings of the 11th International Conference on Coordination Models and Languages*, Lisbon, Portugal.
5. Serrano, M., Gallesio, E., Loitsch, F. 2006. Hop, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

MANUEL SERRANO is a senior scientist at INRIA, leading the INDES (Informatique Diffuse et Sécurisée) team in Sophia-Antipolis. After completing his Ph.D. at the Pierre and Marie Curie University (UPMC) on the compilation of functional languages, he moved to Nice and created the Bigloo development environment for Scheme. He joined INRIA in 2001 and has focused on development environments for the diffuse web since 2005.

GÉRARD BERRY, director of research at INRIA, works on programming languages, their mathematical

semantics, and program verification technologies. His focus has been on the lambda calculus and its models; reactive and realtime programming and verification; and high-level digital circuit specification, synthesis, and verification. Prior to joining INRIA he was chief scientist of Esterel Technologies and was the main author of the Esterel language, which has been used in academia and industry for applications ranging from complex circuit synthesis to airplane control.

© 2012 ACM 1542-7730/12/0600 \$10.00