Rethinking Traditional Web Interaction

Vincent Balat
Univ Paris Diderot – Sorbonne Paris Cité – PPS, UMR 7126 CNRS, Inria – Paris, France
Email: vincent.balat @ univ-paris-diderot.fr

***Abstract*—Web sites are evolving into ever more complex distributed applications. But current Web programming tools are not fully adapted to this evolution, and force programmers to worry about too many inessential details. We want to define an alternative programming style better fitted to that kind of applications. To do that, we propose an analysis of Web interaction in order to break it down into very elementary notions, based on semantic criteria instead of technological ones. This allows defining a common vernacular language to describe the concepts of current Web programming tools, but also some other new concepts. This results in a significant gain of expressiveness. The understanding and separation of these notions also makes it possible to get strong static guarantees, that can help a lot during the development of complex applications, for example by making impossible the creation of broken links. Most of the ideas we propose have been implemented in the Ocsigen Web programming framework that make possible to write a client-server Web applications as a single program. We will show that the interaction model we propose is fully compatible with this kind of applications.**

***Keywords*—Typing; Web interaction; Functional Web programming; Continuations**

I. INTRODUCTION

Nowadays, Web sites behave more and more like real applications, with a high-level of interactivity on both the server and client sides. For this reason, they deserve well-designed programming tools, with features like high-level code structuring and static typing. These tools must take into account the specificities of that kind of application. One of these specificities is the division of the interface into *pages*, connected to each other by links. These pages are usually associated to *URLs* which one can bookmark. It is also possible to turn back to one page using the *back button*. This makes the dynamics of the interface completely different from a regular application. Another specificity is that this kind of applications is highly dependent on standards as they will be executed on various platforms.

Web programming covers a wide range of fields, from database to networking. The ambition of this paper is not to address them all, nor to deal with the full generality of *service oriented computing*. We concentrate on what we will call *Web interaction*; that is, the interaction between a user and a Web application, through a browser interface. We place ourselves in the context of writing such an application, that communicates with one or several servers, and with the ability to make part of the computation in the browser. A few similar Web interaction systems have already been described before (for example Links [1], or Hop [2]). The goal of this paper is mainly to focus on one feature that has been very rarely addressed before, namely: *service identification*, that is, how the service to handle a request is chosen. We want to show that a good service identification mechanism can help programmers a lot, and that the concepts we present here allow to take into account very concrete needs of Web developpers that are usually not addressed by more theoretical works.

We want to make a first step towards a formalization of this interaction with a twofold goal. First, we want to increase the expressiveness of Web frameworks. Second, we want to improve the reliability of Web applications by using well defined concepts and static validation of the code.

The concepts we present here have been implemented in the Ocsigen Web programming framework [3], [4], [5] (Eliom project). It allows to program fully in OCaml both the server and client parts of a Web application, with a consistent abstraction of concepts. A compiler to Javascript is used to run the client parts in the browser [6], [7].

Work partially supported by the French national research agency (ANR), PWD project, grant `ANR-09-EMER-009-01`, and performed at the IRILL center for Free Software Research and Innovation in Paris, France

A. A common vernacular language for Web interaction

Web development is highly constrained by technologies. First, it relies on the HTTP protocol, which is non-connected and stateless. Then, Web applications must be executable in various browsers that implement more or less accurately common standards and recommendations.

We want to detach ourselves as much as possible from these constraints and think about how we would like to program Web interaction. Obviously this is also a question of taste. But rather than proposing yet another programming model from scratch, we start by analyzing common Web programming practices, in order to understand the notions they use. Then we decompose them in very elementary notions that can be used to describe the features of Web programming tools, from PHP to JSP or Miscrosoft.NET Web Forms, etc. We will observe that current frameworks impose too many artificial restrictions. Ideally, we would like to give a generic language, flexible enough to describe all possible behaviours, without imposing any artificial restriction due to one technology.

We place ourselves at a semantic level rather than at a technical one. Moving away from technical details will allow to increase the *expressiveness* of Web programming frameworks. In the domain of programming languages, high-level concepts have been introduced over the years, for example genericity, inductive types, late binding, closures. They make easier the implementation of some complex behaviours. We want to do the same for the Web. For example the notion of *"sending a cookie"* benefits from being abstracted to a more semantic notion like *"opening a session"* (which is already often the case today). Also it is not really important for the programmer to know how URLs are formed. What matters is the service we want to speak about (and optionally the parameters we want to send it).

This abstraction from technology allows two things:

- First, it increases the expressiveness of the language by introducing specific concepts closer to the behaviours we want to describe (and irrespective of the way they are implemented). From a practical point of view, this allows to implement complex behaviours in very few lines of code.
- Having well-designed dedicated concepts also allows to avoid wrong behaviours. We forbid unsafe technical possibilities either by making them inexpressible, or by static checking.

B. Improving the reliability of Web applications

As Web sites are currently evolving very quickly into complex distributed applications, the use of strongly and statically typed programming languages for the Web becomes more and more helpful. Using scripting languages was acceptable when there was very little dynamic behaviour in Web pages, but current Web sites written with such languages are proving to be very difficult to evolve and maintain. Some frameworks are counterbalancing their weaknesses by doing a lot of automatic code generation (for example [8]). But this does not really improve the *safety* of programs. In the current state of knowledge, we are able to do much better, and Web programming must benefit from this.

Static validation of pages: One example where static typing revolutionizes Web programming concerns the validation of pages. Respecting W3C recommendations is a necessity to ensure portability and accessibility of Web sites. The novelty is that there now exist typing systems sophisticated enough to statically ensure a page's validity [9], [10], [11] We do not mean checking the validity of pages once generated, but really to be sure that the program that builds the XML data will always generate something valid, even in the most particular cases.

For example, even if a programmer has checked all the pages of his site in a validator, is he sure that the HTML table he creates dynamically will never be empty (which is forbidden)? What if for some reason there is no data? He must be very conscientious to think about all these cases. It is most likely that the evolutions of the program will break the validity of pages. In most cases, problems are discovered much later, by users.

In lots of cases, such errors will even make the generated output unusable, for example for XML data intended to be processed automatically. The best means to be sure that this situation will never happen is to use a typing system that will prevent one from putting the service on-line if there is the slightest risk for something wrong to be generated.

For people not accustomed to such strong typing systems, this may seem to impose too much of a constraint to programmers. Indeed, it increases a bit the initial implementation time (by forcing to take into account all cases). But it also saves such a huge amount of debugging time, that the use of such typing systems really deserves to be generalized. For now, these typing systems for XML are used in very few cases of Web services, and we are not aware of any major Web programming framework. Our experience is that it is not difficult to use once one get used to the main rules of HTML grammar, if error messages are clear enough.

Validity of Web interaction: Static checking and abstraction of concepts can also benefit in many other ways to Web programming, and especially to Web interaction. Here are a few examples: In a link, do the types (and names) of parameters match the types expected by the service it points to? Does a form match the service it points to? Do we have broken links?

It is not so difficult to have these guarantees, even if almost no Web programming framework are doing so now. All what is needed is a programming language *expressive* enough (in the sense we explained above).

Improving the ergonomics of Web sites: Lots of Web developers are doing implementation errors resulting in reduced ease of use (wrong use of sessions or GET and POST parameters, etc.). Take as example a famous real estate Web site that allows to browse through the results of a search; but if someone sets a bookmark on one of the result pages, he never goes back to the same page, because the URL does not refer to the advertisement itself, but to the rank in the search. We will see that a good understanding of concepts can avoid such common errors.

C. Overview of the paper

Sections II and III are devoted to the definition of our vernacular language for describing the services provided by a Web application. Section II explains the advantage of using an abstract notion of service instead of traditional page-based programming and string URLs. Section III presents a new service identification and selection method. It shows how powerful this notion of service can be made, by separating it into several kinds. This results in a very new programming style for Web interaction.

II. ABSTRACTING SERVICES

As explained above, we want to formalize Web interaction, that is, the behaviour of a Web application in reaction to the actions of the user. What happens when somebody clicks on a link or submits a form? A click often means that the user is requesting a new document: for example a new page that will replace the current one (or one part of it). But it can also cause some actions to take place on the server or the client. Let us enumerate the different kinds of reactions. A click (or a key strike) from the user may have the following main effects:

1) Modifying the application interface. That is, changing the page displayed by the browser (or one part of the page), or opening a new window or tab with a new page,
2) Changing the URL displayed by the browser (protocol, server name, path, parameters, etc.),
3) Doing some other action, like the modification of a state (for example changing some database values),
4) Sending hidden data (like form data, or files),
5) Getting some data to be saved on the user's hard disk.

Two important things to notice are that each of these items is optional, and may either involve a distant server, or be processed locally (by the browser).

This decomposition is important, as a formalization of Web interaction should not omit any of these items in order not to restrict the freedom of the programmer. All these items are described semantically, not technically.

A. The role of URLs

The item "Changing the URL" above is a really significant one and is one key to understand the behaviour of Web applications. This section is devoted to the understanding of that notion. URLs are entry points to the Web site. Changing the URL semantically means: giving the possibility to the user to turn back to this point of interaction later, for example through bookmarks.

Note that, unlike many Web sites, a good practice is to keep the URL as readable as possible, because it is an information visible to users that may be typed manually.

1) Forgetting technical details about URLs: The syntax of URLs is described by the Internet standard STD 66 and RFC 3986 and is summarized (a bit simplified) here:
`scheme://user:pwd@host:port/path?query#fragment`

The path traditionally describes a file in the tree structure of a file system. But this view is too restrictive. Actually, the path describes the hierarchical part of the URL. This is a way to divide a Web site into several sections and subsections.

The query string syntax is commonly organized as a sequence of '*key*=*value*' pairs separated by a semicolon or an ampersand, e.g., `key1=value1&key2=value2&key3=value3`. This is the part of the URL that is not hierarchical.

To a first approximation, the path corresponds to the service to be executed, and the query to parameters for this service. But Web frameworks are sometimes taking a part of the path as parameters. On the contrary, part of the query, or even of the host, may be used to determine the service to call. This will be discussed later in more detail.

The *fragment* part of the URL only concerns the browser and is not sent to the server.

The item "Changing the URL" is then to be decomposed semantically into these sub-tasks:

1) Changing the protocol to use,
2) Changing the server (and port) to which the request must be made,
3) Choosing a hierarchical position (path) in the Web site structure, and specifying non hierarchical information (query) about the page,
4) And optionally: telling who the user is (credentials) and the fragment of the page he wants to display.

 ISBN: 978-1-61208-280-6

2) URL change and service calls: There are two methods to send form data using a browser: either in the URL (GET method) or in the body of the HTTP request (POST method). Even if they are technical variants of the same concept (a function call), their semantics are very different with respect to Web interaction. Having parameters in the URL allows to turn back to the same document later, whereas putting them in the request allows to send one-shot data to a service (for example because they will cause an action to occur).

We propose to focus on this semantical difference rather than on the way it is implemented. Instead of speaking about POST or GET parameters, we prefer the orthogonal notions of *service calls* and *URL change*. It is particularly important to forget the technical details if we want to keep the symmetry between server and client side services. Calling a local (javascript for example) function is similar to sending POST data to a server, if it does not explicitly change the URL displayed by the browser.

Semantically speaking, in modern Web programming tools, changing the URL has no relation with *calling a service*. It is possible to call a service without changing the URL (because it is a local service, or because the call uses POST parameters). On the contrary, changing the URL may be done without calling a service. There is only one reason to change the URL: give the user a new entry point to the Web site, to which he can come back when he wants to ask the same service once again, for example by saving it in a bookmark.

B. Services as first class values

The first main principle on which is based our work is: *consider services as first class values,* exactly as functional languages consider functions as first class valuesThat is: we want to manipulate services as abstract data (that can for example be given as parameter to a function). This has several advantages, among which:

- The programmer does not need to build the syntax of URLs himself. Thus, it is really easy to switch between a local service and a distant one.
- All the information about the service is taken automatically from the data structure representing the service, including the path to which the service is attached and parameter names. This has a very great consequence: if the programmer changes the URL of a service, even the name of one of its parameters, he does not need to change any link or form towards this service, as they are all built automatically. This means that links will never be broken, and parameter names will always be correct (at least for internal services, i.e., services belonging to the Web site).

Some recent frameworks already have an abstraction of the notion of service. We want to show how to take the full benefit of it. Our notion of service must be powerful enough to take into account all the possibilities described above, but without relying on their technical implementation.

A service is some function taking parameters and returning some data, with possibly some side effects (remote function calls). The server is a provider of *services*. Client side function calls can also be seen as calls to certain services. The place where services take place is not so significant. This allows to consider a Web site with two versions of some services, one on server side, the other on client side, depending on the availability of some resources (network connection, or browser plug-ins for example).

The language must provide some way to define these services, either using a specific keyword or just through a function call.

Once we have this notion, we can completely forget the old "page-based" view of the Web where one URL was supposed to be associated to one file on the hard disk. Thus, it is possible to gain a lot of freedom in the organization and modularity of the code, and also, as we will see later, in the way services are associated to URLs. One of the goals of next section is precisely to discuss service identification and selection, that is, how services are chosen by the server from the hierarchical and non-hierarchical parts of the URL, and hidden parameters.

III. A TAXONOMY OF SERVICES

A. Values returned by services

A first classification of services may be made according to the results they send. In almost all Web programming tools, services send HTML data, written as a string of characters. But as we've seen before, it is much more interesting to build the output as a tree to enable static type checking. To keep full generality, we will consider that a service constructs a result of any type, that is then sent, possibly after some kind of serialization, to the browser which requested it. It is important to give to the programmer the choice of the kind of service he wants.

A reflection on return types of services will provide once again a gain of expressiveness. Besides plain text or typed HTML trees, a service may create for example a redirection. One can also consider using a service to send a file. It is also important to give the possibility to services to choose themselves what they want to send. For example, some service may send a file if it exists, or an HTML page with an error message on the other case. The document is sent together with its *content type*, telling the browser how to display it (it is a dynamic type). But in other cases, for example when a service implements a function to be called from the client side part of the program, one probably want the type of the result to be known statically.

We also introduce a new kind of output called *actions*. Basically sending an action means "no output at all". But the service may perform some action as side effect, like modifying a database, or connecting a user (opening a new session). From a technical point of view, actions implemented server side are usually sending a 204 (No content) HTTP status code. Client side actions are just procedures. We will see some examples of use of actions and how to refine the concept in Section III-D2.

B. Dynamic services, or continuation-based Web programming

1) Dynamic services: Modern Web frameworks propose various solutions to get rid of the lack of flexibility induced by a one-to-one mapping between one URL and one service. But almost none of them take the full benefit of this, and especially of one very powerful consequence: the possibility to dynamically create new services.

For example, if we want to add a feature to a Web site, or even if we occasionally want to create a service depending on previous interaction with one user. For example, if one user wants to book a plane ticket, the system will look in a database for available planes and dynamically create the services corresponding to booking each of them. Then it displays the list of tickets, with, on each of them, a link towards one of these dynamic services. Thus, we will be sure that the user will book the ticket he expects, even if he duplicates his browser window or uses the back button. This behaviour is really simple to implement with dynamic services and rather tricky with traditional Web programming. Witness the huge number of Web sites which do not implement this correctly.

If we want to implement such behaviour without dynamic services, we will need to save somewhere all the data the service depends on. One possibility is to put all this data in the link, as parameters, or in hidden form data. Another possibility, for example if the amount of data is prohibitive, is to save it on the server (for example in a database table) and send only the key in the link.

With dynamic service creation, all this contextual data is recorded automatically in the *environment* of the closure implementing the service. This closure is created dynamically according to some dynamic data (recording the past of the interaction with the user). It requires a functional language to be implemented easily.

 ISBN: 978-1-61208-280-6

2) Continuations: This feature is equivalent to what is known as *continuation-based Web programming*. This technique was first described independently by Christian Queinnec [12], [13], [14], John Hughes [16] and Paul Graham [15].

The use of dynamic services is a huge step in the understanding of Web interaction, and an huge gain of expressiveness. Up until now, very few tools have used these ideas. None of the most widely used Web programming frameworks implement them, but they are used for example in Seaside [17], PLT Scheme [18], Hop [2], Links [1], and obviously Ocsigen.

The cost (in terms of memory or disk space consumption) is about the same as with usual Web programming: no copy of the stack, one instance of the data, plus one pointer to the code of the function.

C. Finding the right service

The very few experimental frameworks which are proposing some kind of dynamic services impose usually too much rigidity in the way they handle URLs. This section is devoted to showing how it is possible to define a notion of service identification that keeps all the possibilities described in Section II.

The important thing to take care of is: how to do the association between a request and a service? For example if the service is associated to an URL, where, in this URL, is the service to be called encoded?

To make this as powerful as possible, we propose to delegate to the server the task of decoding and verifying parameters, which is traditionally done by the service itself. This has the obvious advantage of reducing a lot the work of the service programmer. Another benefit is that the choice of the service to be called can depend on parameters.

Let us first speak about distant (server side) bookmarkable services, i.e., services called by sending a GET request to a server. We will speak later about client side services, and hidden services.

1) Hierarchical services: One obvious way to associate a service to an URL is by looking at the path (or one part of it). We will call these services *hierarchical services*. These kinds of services are usually the main entry points of a Web site. They may take parameters, in the *query* part of the URL, or in the path. One way to distinguish between several hierarchical services registered on the same path is to look at parameters. For example the first registered service whose expected parameters exactly match the URL will answer.

2) Coservices: Most of the time one probably wants dynamic services to share their path with a hierarchical service, at least those which last for only a short time (result of a search for example). Also one may want two services to share the same hierarchical position on the Web site.

We will call *coservices* services that are not directly associated to a path, but to a special parameter. This is one of the main original features of our service identification mechanism and this has a huge impact on expressiveness, as we will see on example in Section III-D2. From a semantic point of view, the difference is that hierarchical services are the entry points of the site. They must last forever, whereas coservices may have a timeout, and one probably want to use the associated main service as fallback when the coservice has expired.

We will distinguish between *named coservices* and *anonymous coservices*, the difference being the value of the special parameter. Named coservices have a fixed parameter value (the name of the coservice), whereas this value is generated automatically for anonymous coservice.

Like all other services, coservices may take parameters, that will be added to the URL. There must be a way to distinguish between parameters for this coservice and parameters of the original service. This can be done by adding automatically a prefix to coservice parameters.

3) Attached and non-attached coservices: We will also distinguish between coservices *attached* to a path and *non-attached* coservices. The key for finding an attached coservice is the path completed by a special parameter, whereas non-attached coservices are associated to a parameter, whatever the path in the URL. This feature is not so common and we will see in Section III-D2 how powerful it is.

4) Distant hidden services: A distant service is said to be *hidden* when it depends on POST data sent by the browser. If the user comes back later, for example after having made a bookmark, it will not answer again, but another service, not hidden, will take charge of the request. We will speak about *bookmarkable* services, for services that are not hidden.

Hidden services may induce an URL change. Actually, we can make exactly the same distinction as for bookmarkable services: there are hierarchical hidden services (attached to a path), hidden attached coservices (attached to a path, and a special POST parameter), and hidden non-attached coservices (called by a special POST parameter).

It is important to allow the creation of hidden hierarchical services or coservices only if there is a bookmarkable (co)service registered at the same path. This service will act as a fallback when the user comes back to the URL without POST parameters. This is done by specifying the fallback instead of the path when creating a hidden service. It is a good idea to do the same for bookmarkable coservices.

Registering a hidden service on top of a bookmarkable service with parameters allows to have both GET and POST parameters for the same service. But bear in mind that their roles are very different.

5) Client side services: Client side service calls have the same status as hidden service calls. In a framework that allows to program both the server and client sides using the same language, we would like to see local function calls as non-attached (hidden) coservices. Hierarchical hidden services and (hidden) attached coservices correspond to local functions that would change the URL, without making any request to the server.

D. Taxonomy of services

1) Summary of service kinds: Figure 1 summarizes the full taxonomy of services we propose. This set is obviously *complete* with respect to technical possibilities (as traditional services are part of the table). It is powerful enough for describing in very few lines of code lots of features we want for Web sites, and does not induce any limitations with respect to the needs of Web developers. Current Web programming frameworks usually implement a small subset of these possibilities. For example “page-based” Web programming (like PHP or CGI scripts) does not allow for non-attached coservices at all. Even among “non-page-based” tools, very few allow for dynamic (anonymous) coservice creation. To our knowledge, none (but Ocsigen) is implementing actions on non-attached services as primary notions (even if all the notions can obviously be simulated).

2) Example cases: We have already seen some examples of dynamic service creation: if a user creates a blog in a subdirectory of his or her personal site, one possibility is to add dynamically a hierarchical service to the right path (and it must be recreated every time the serser is relaunched). If we want to display the result of a search, for example plane ticket booking, we will create dynamically a new anonymous coservice (hidden or not), probably with a timeout. Without dynamic services, we would need to save manually the search keyword or the result list in a table.

Coservices are not always dynamic. Suppose we want a link towards the main page of the site, that will close the session. We will use a named hidden attached coservice (named, so that the coservice key is always the same).

We will now give an example where non-attached hidden coservices allow to reduce significantly the number of lines of code. Consider a site with several pages. Each page has a connected version and a non-connected version, and we want a connection box on

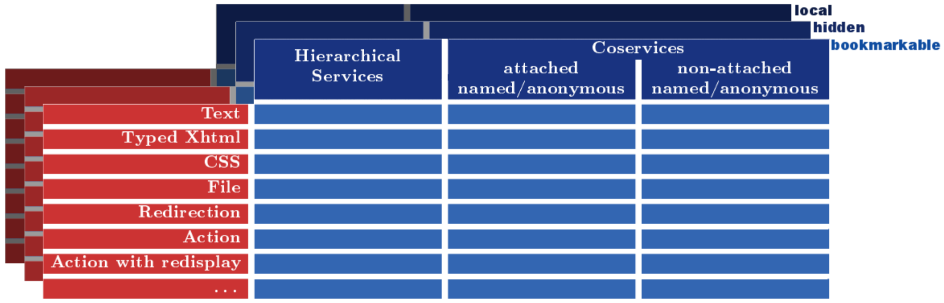

Figure 1: Full taxonomy of services.

each non-connected page. But we don't want the connection box to change the URL. We just want to log in and stay on the same URL, in connected version. Without non-attached services (and thus with almost all Web programming tools), we need to create a version with POST parameters of each of our hierarchical services to take into account the fact that each URL may be called with user credentials as POST parameters.

Using our set of services, we just need to define only one non-attached (hidden) coservice for the connection. At first sight, that service only performs an action (as defined in Section III-A): saving user information in a session table. But we probably want to return a new page (connected version of the same page). This can be done easily by returning a redirection to the same URL. Another solution if we don't want to pay the cost of a redirection, is to define a new kind of output: *"action with redisplay"* that will perform the action, then make an internal (server side) request as if the browser had done the redirection. The solution with redirection has one advantage: the browser won't try to resend POST data if the user reloads the page.

Now say for example that we want to implement a wiki, where each editable box may occur on several pages. Clicking on the *edit* button goes to a page with an edit form, and submitting the form must turn back to the original page. One dirty solution would be to send the original URL as hidden parameter in the edit link. But there is now a simpler solution: just do not change the path. The edit form is just a page registered on a non-attached service.

Our reflexion on services, also allows to express clearly a solution to the real estate site described in Section I-B. Use (for example) one bookmarkable hierarchical service for displaying one piece of advertisement, with additional (hidden or not) parameters to recall the information about the search.

3) Expressiveness: The understanding of these notions and their division into very elementary ones induces a significant gain in expressiveness. This is particularly true for actions with redisplay. They are very particular service return values and seem to be closely related to non-attached coservices at first sight. But the separation of these two concepts introduces a new symmetry to the table, with new cells corresponding to very useful possibilities (see the example of the wiki above). It is noteworthy that all cells introduced in this table have shown to be useful in concrete cases.

IV. Conclusion

A. Related work

A lot of modern Web programming frameworks (for example GWT or Jif/Sif [19]) are trying to propose integrated and high level solutions to make easier the development of Web application. They often provide some abstraction of concepts, but most of them preserve some historical habits related to technical constraints. It is impossible to make a full review of such tools, as there are numerous. We will concentrate on the main novel features presented here. One can try to make a classification of existing Web frameworks with respect to the way they do service identification.

The old method is what we called "page-based Web programming", where one path corresponds to one file. Modern tools are all more flexible and make service identification and selection independent of the physical organization of components in the Web server (for example JSP assigns an URL to a service from a configuration file). But very few belong to the third group, that allows dynamic services. Among them: Seaside [17], Links [1] and Hop [2], Wash/CGI [20]. Their service identification models are more basic, and they don't have a native notion of coservice. Some of them are using an abstraction of forms [20], [21] that is fully compatible with our model.

There have been few attempts to formalize Web interaction. The most closely related work is by Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi and Matthias Felleisen [22], [23]. Their work is more formal but does not take into account all the practical cases we speak about. In particular their service model is much simpler and does not fully take into account the significance of URLs. Peter Thiemann [20] uses monads to create HTML pages, which makes possible an original and interesting way of handling the typing of forms, using Haskell's type system.

We think our approach is compatible with more data driven approaches [8], component based interfaces [24], or even code generation techniques. One interesting work would be to see how they can be mixed together.

B. Evolution of technologies and standards

This reflection about Web programming techniques has shown that Web technologies suffer from some limitations that slow down the evolution towards really dynamic applications. Here are a few examples:

- As mentioned above, the format of page parameters and the way browsers send them from form data does not allow for sophisticated parameters types.
- (X)HTML forms cannot mix GET and POST methods. It is possible to send URLs parameters in the `action` attribute of a form that is using the POST method, but it is not possible to take them from the form itself. This would open many new possibilities.
- A link from HTTP towards the same site in HTTPS is always absolute. This breaks the discipline we have to use only relative links (for example to behave correctly behind a reverse proxy). We have the same problem with redirections, which have to be absolute URLs according to the protocol.
- There is no means to send POST data through a link, and it is difficult to disguise a form into a link. Links and forms should probably be unified into one notion, that would allow to make a (POST or GET) request from a click on any part of the page. This limitation is not really significant if we have a client side program that does the requests itself when we click on a page element.
- Having the ability to put several `id` attributes for one tag would be very useful for automatically generated dynamic pages.

- Probably one of the main barriers to the evolution of the Web today is the impossibility to run fast code on the browser (without plug-ins), even with recent implementations of Javascript. When thinking about a Web application as a complex application distributed between a server and a client, we would often like to perform computationally intensive parts of the execution on the client, which is not feasible for now. We want to make some experiments with Google Native Client [25].

C. Concluding words and future works

This papers presents a new programming style for Web interaction which simplifies a lot the programming work and reduces the possibilities of semantical errors and bad practices. The principles we advocate are summarized here:

1) Services as first class values
2) Decoding and verification of parameters done by the server
3) Dynamic creation of services
4) Full taxonomy of services for precise service identification
5) Same language on server and client sides
6) Symmetry between local and distant services

One of the main novel feature is the powerful service identification mechanism performed automatically by the server. It introduces the notion of coservice which make the programming of sophisticated Web interaction very easy.

Beyond just presenting a new Web programming model, this paper defines a new vocabulary for describing the behaviour of Web sites, on a semantic basis. It is a first step towards a formalization of Web interaction. We started from an analysis of existing Web sites and we extracted from this observation the underlying concepts, trying to move away as much as possible from non-essential technical details. This allowed a better understanding of the important notions but above all to bring to light some new concepts that were hidden by technical details or historical habits. The main feature that allowed this is the introduction of dynamic services, and also forgetting the traditional page-based Web programming. There exist very few frameworks with these features, and none is going as far as we do, especially in the management of URLs.

Besides the gain in expressiveness, we put the focus on reliability. This is made necessary by the growing complexity of Web applications. The concepts we propose allow for very strong static guarantees, like the absence of broken links. But more static checks can be done, for example the verification of adequacy of links and forms to the service they lead to. These static guarantees have not been developed here because of space limitation. They are summarized by the following additional principles:

7) Static type checking of generated data
8) Static type checking of links and forms

This paper does not present an abstract piece of work: all the concepts we present have been inspired by our experience in programming concrete Web sites, and have been implemented. Please refer to Ocsigen's manual [26] and source code for information about the implementation. Some implementation details may also be found in [27] (describing an old version of Ocsigen that was using a more basic model of services). Ocsigen is now used in industry (for example BeSport [29], Pumgrana [28]). These concrete experiences showed that the programming style we propose is very convenient for Web programmers and reduces a lot the work to be done on Web interaction.

This paper is not a full presentation of Ocsigen. Many aspects have been hidden, and especially how we program the client side part of the application [6], [7], [30] using the same language, and with the same strong static guarantees. As we have seen, our notions of services also apply to client side functions. Obviously we are using the same typing system for services but also for HTML. It is not easy to guarantee that a page will remain valid if it can evolve over time [31]. We did not show how the server can send data to the client at any time, or even call a function on client side.

References

[1] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web programming without tiers," in In 5th International Symposium on Formal Methods for Components and Objects (FMCO). Springer-Verlag, 2006.

[2] M. Serrano, E. Gallesio, and F. Loitsch, "Hop, a language for programming the web 2.0," in Dynamic Languages Symposium, Oct. 2006.

[3] The Ocsigen project, http://www.ocsigen.org [retrieved: 03, 2013].

[4] V. Balat, J. Vouillon, and B. Yakobowski, "Experience report: ocsigen, a web programming framework," in ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming.

[5] V. Balat, P. Chambart, and G. Henry, "Client-server Web applications with Ocsigen," in WWW2012 dev track proceedings

[6] B. Canou, E. Chailloux, and J. Vouillon, "How to Run your Favorite Language in Web Browsers," in WWW2012 dev track proceedings

[7] J. Vouillon and V. Balat, "From bytecode to javascript: the js_of_ocaml compiler," Journal of Software: Practice and Experience, 2013.

[8] Ruby on rails, http://www.rubyonrails.com/ [retrieved: 03, 2013]

[9] A. Frisch, "Ocaml + xduce," in Proceedings of the international conference on Functional programming (ICFP). ACM, 2006

[10] V. Benzaken, G. Castagna, and A. Frisch, "CDuce: An XML-centric general-purpose language," in Proceedings of the International Conference on Functional Programming (ICFP), 2003

[11] H. Hosoya and B. C. Pierce, "XDuce: A statically typed XML processing language," ACM Transactions on Internet Technology, vol. 3, no. 2, May 2003.

[12] C. Queinnec, "The influence of browsers on evaluators or, continuations to program web servers," in International conference on Functional programming (ICFP), 2000

[13] C. Queinnec, "Continuations and web servers," Higher-Order and Symbolic Computation, Dec. 2004.

[14] C. Queinnec, "Inverting back the inversion of control or, continuations versus page-centric programming," ACM SIGPLAN Notices, vol. 38, no. 2, Feb. 2003.

[15] P. Graham, "Beating the averages" http://www.paulgraham.com/avg.html [retrieved: 03, 2013].

[16] J. Hughes, "Generalising monads to arrows," Science of Computer Programming, vol. 37, no. 1–3 2000.

[17] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside – a multiple control flow web application framework," in Proceedings of ESUG Research Track 2004, 2004.

[18] S. Krishnamurthi, P. W. Hopkins, J. Mccarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen, "Implementation and use of the plt scheme web server," in Higher-Order and Symbolic Computation, 2007.

[19] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," SIGOPS Oper. Syst. Rev., vol. 41, no. 6, 2007.

[20] P. Thiemann, "Wash/cgi: Server-side web scripting with sessions and typed, compositional forms," in Practical Aspects of Declarative Languages (PADL'02), 2002

[21] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "The essence of form abstraction," in Sixth Asian Symposium on Programming Languages and Systems, 2008.

[22] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen, "Modeling web interactions," in European Symposium on Programming (ESOP), April 2003

[23] S. Krishnamurthi, R. B. Findler, P. Graunke, and M. Felleisen, "Modeling web interactions and errors," in In Interactive Computation: The New Paradigm. Springer Verlag, 2006.

[24] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera, "A framework for rapid integration of presentation components," in WWW '07: Proceedings of the 16th international conference on World Wide Web. ACM, 2007.

[25] "Google native client," http://code.google.com/p/nativeclient/ [retrieved: 03, 2013].

[26] V. Balat, "Eliom programmer's guide," Technical report, Laboratoire PPS, CNRS, université Paris-Diderot, Tech. Rep., 2007

[27] V. Balat, "Ocsigen: Typing web interaction with objective caml," in ML'06: Proceedings of the 2006 ACM SIGPLAN workshop on ML

[28] "Pumgrana," http://www.pumgrana.com/ [retrieved: 03, 2013].

[29] "BeSport," http://www.besport.com [retrieved: 03, 2013].

[30] B. Canou, V. Balat, and E. Chailloux, "O'browser: objective caml on browsers," in ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML.

[31] B. Canou, V. Balat, and E. Chailloux, "A declarative-friendly api for web document manipulation," in International Symposium on Practical Aspects of Declarative Languages (PADL'13). Springer-Verlag, January 2013.