Static and Dynamic Semantics of the Web

Christopher Fry
Clear Methods, 1 Broadway, Cambridge, MA

Mike Plusch
Clear Methods, 1 Broadway, Cambridge, MA

Henry Lieberman
Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA

Introduction

The original perception of the Web by the vast majority of its early users was as a static repository of unstructured data. This was OK for browsing small sets of information by humans, but this static model is now breaking down as programs attempt to dynamically generate information, and as human browsing is increasingly assisted by intelligent agent programs.

The next phase of the Web, as represented in this book's movement towards the "Semantic Web", lies in encoding properties of and relationships between, objects represented by information stored on the Web. It is envisioned that authors of pages include this semantic information along with human-readable Web content, perhaps with some machine assistance in the encoding. Parsing unstructured natural language into machine-understandable concepts is not feasible in general, although some programs may be able to make partial sense out of Web content.

However, this semantics is primarily *declarative* semantics, semantics that changes only relatively slowly as Web pages are created, destroyed, or modified, typically by explicit, relatively coarse-grained human action.

Less concern has been given to *dynamic* semantics of the Web, which is equally important. Dynamic semantics have to do with the *creation* of content, actions which may be guided by

- User-initiated interface actions
- Time
- Users' personal profiles
- Data on a server

and other conditions.

Even less concern has been given to methods for cleanly integrating the static and dynamic aspects of the Web. In this paper, we discuss the need for dynamic semantics, and show how dynamic semantics will be enabled by and useful to the new generation of intelligent agent software that will increasingly inhabit the Web. Though there will be autonomous agents, the more interesting agents will cooperate interactively with humans, helping them to achieve their goals more efficiently than a user could on their own. We also present a new language, *Water*, that fully integrates static and dynamic semantics, while keeping close to the XML framework that has made the Web successful.

Beyond that, we envision that Web end users and Web application developers will not be routinely writing code directly in Water or any other formal semantic language, but instead avail themselves of Programming by Example [Lieberman, ed. 01] and other interactive agent interfaces that will hide the details of the formal languages. But transition to these future interfaces will be greatly aided by a foundation that can cleanly integrate static and dynamic semantics.

Static Semantics

The Web's link structure mimics a semantic net

A semantic net [Woods 75] is a network of concepts linked by relations. The Web is, of course, a network of pages, each containing text, pictures, other media types, and links to other Web pages. Though the Web has far less structure than typical AI semantic nets, the Web pages that constitute the nodes of the Web's network often do represent concepts and the links between them represent relations between those concepts. For example, my home page is the Web's representation of me, in a sense. The links leading off my home page - to my publication list, my e-mail address, courses that I teach, etc. - represent the relation of me to, e.g. the articles I have published.

The only problem is that these relations are expressed in human-understandable natural language and human-understandable pictures. Short of full natural language understanding, it is difficult for a computer program to automatically extract those concepts and relations in order to do query answering, inference and other tasks.

Should the Semantic Web be static?

The Semantic Web movement, represented in many of the articles in this book, and in Web standards initiatives such as RDF, DAML, and OIL, is an attempt to introduce common formal languages for expressing concepts and relations in a machine readable way. To leverage existing Web tools and emulate the Web's social success, such efforts strive to embed the descriptive information in pages similar to the way text, pictures and conventional media are already described using HTML and XML.

However, all this is basically limited to *static* semantics. The interaction paradigm is that knowledge descriptions are authored by a developer in the same way that HTML pages are presently authored by a Web designer, then "published" to the Web to make them available in the network. Sometimes, the descriptions may be automatically produced by a program from the results of user interaction in a manner similar to the way many WYSIWYG Web editors such as Macromedia Dreamweaver or Adobe GoLive produce HTML automatically from user editing operations.

In all of this, there is the underlying assumption that semantics

- Is *represented declaratively.* It is represented in passive data that is descriptive and can be retrieved, e.g. a Web page on a Web server, and
- *Changes relatively slowly.* The Web publishing events happen rarely relative to actions such as browsing a page or clicking a link.

Structured static semantics are a huge advance for the web since they will enable software agents to find and reason about a colossal volume of information, essentially turning mere data into knowledge. But the larger and more complex the knowledge, the less complete and useful static representations become.

Dynamic Semantics

In addition to the static semantics of Web pages, links, and Web markup, there is also what we call dynamic semantics. Dynamic semantics

- Is *represented procedurally*. It can be computed by programs running on the client or server side, based on immediate interactive user input. This computation can depend on the immediate *context* - including time, personal information about the user, user-initiated actions, etc.
- *Changes relatively rapidly.* A single user click can cause the semantics to be generated or to change, or it can be changed by the actions of programs continuously in real-time.

As the web matures, there are many ways in which static semantics are being augmented and supplanted by dynamic semantics. As a simple example, some URLs are not addresses of static pages stored on Web servers, but rather act as directives to the server to initiate some computation. CGI scripts are an example of this. The question-mark in the URL is a signal for the server to retrieve some named program and execute it, possibly with arguments. An Active Server Page queries a database and constructs a page on the fly. Even search engine results pages, and customized ads based on cookies are examples of dynamically created Web pages. Streaming audio, video and other media also make the Web more dynamic.

Transparency between static and dynamic semantics

The key is that, to the browser, the HTTP stream delivered by the procedure is identical to that which would have come from a statically stored page, so that the requester need not concern themselves with the question of whether that information was stored or computed.

That kind of transparence between static and dynamic data is an extremely important property that a system can have. It means that a system can always be extended by replacing some piece of formerly static data with a new, dynamically generated object, as long as the new object's behavior is compatible with the old. This property has long been appreciated in the communities of AI and HCI languages such as Lisp, Prolog and Smalltalk, though it has been underappreciated in the communities of users of more conventional languages such as C and Java. In the knowledge representation community, this has long been studied under the name *procedural attachment*. Our aim is to extend the principle of equivalence of static and dynamic data as the Web moves toward encoding more semantics.

Sources of Dynamic Semantics

In addition to the examples of procedural Web data cited above, there are several other sources of dynamic semantics for the Web. The first is in the process of Web browsing itself.

Web browsing generates dynamic semantics

The process of the user navigating through Web pages might result in new objects and relations being created that need to be represented, both on the client's machine and also on the Web servers with which the client communicates. These may either be represented statically and stored explicitly, or produced dynamically as a result of user action. For example, personal information about the user, such as their current interests, might be communicated from the client to the server. Now, cookies are used as a very primitive means of client-to-server communication, but they can only communicate a single piece of information. Information like the user's current interests might be represented by a complex structure with dynamic contingencies.

Agents generate dynamic semantics

An alternative to having Web page creators explicitly author meta-data, is to have the meta-data computed by agents from human-readable information. An active area of research is having agents "read" Web pages containing natural language and formatted text intended for humans, and compute meta-data by using machine learning to infer "wrappers" that describe the structure of the text [Kushmerick, Weld & Doorenbos 97]. This can be done in many cases without having to completely solve the natural language problem. The field of Information Extraction uses partial parsing to perform tasks like semi-automatic topic categorization and summarization. Examples are price-comparison shopbots that extract prices from on-line catalogs, or filters that remove advertisements. Users may even define these wrappers dynamically [Bauer, Dengler & Paul 01].

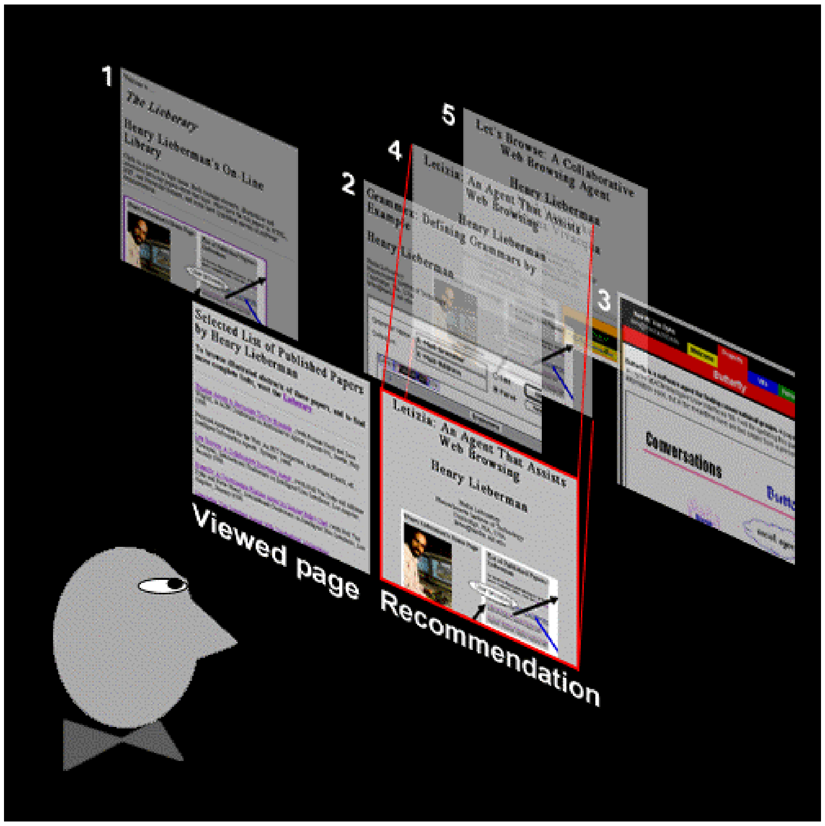
Web editing generates dynamic semantics

Finally, the evolution of the Web itself leads to dynamic semantics. As pages are modified, new pages added and old ones disappear, both Web clients and Web servers might want to track and represent how pages change, or keep history that may be dynamically accessible in different ways. This could be as simple as displaying the date of the last change to a page, or highlighting the parts that have changed.

Web Agents Make Use of Dynamic Semantics

A revolution that is currently underway is the growing popularity of intelligent agents on the Web. Agents are programs that act as assistants to the user in the interface. They can track user interests, explore the Web proactively, learn through interacting with the user, provide personalized data and services, and much more.

Examples are Letizia and Powerscout [Lieberman, Fry & Weitzman 01], which act as *reconnaissance agents*, building a profile of the users' interests by watching his or her Web browsing, and dynamically and incrementally crawling the Web or using traditional search engines as subsidiary tools to suggest related material in real time.



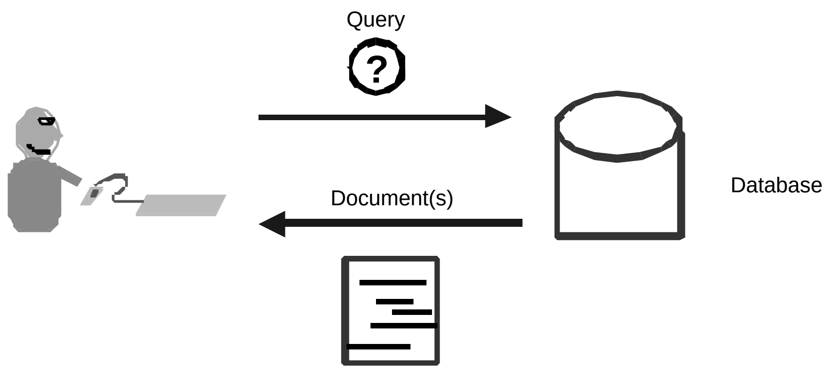
Letizia "spirals out" from the user's current page, filtering pages through the user's interest profile

What sets these kinds of agents apart from the more traditional tools like search engines and cookie-personalized sites is their more dynamic nature. They are computing concepts and relations dynamically from the data stored and retrieved on the Web, procedures which are attached to that data, and also from the dynamic process of the user's interactive navigation through the Web sites.

Information-Retrieval and Theorem-Proving Perspectives

It is instructive to consider how the advent of the Web changed the perspective of information-seeking activities from the old "information retrieval" model, essentially a static model, to a newer model of dynamic, continuous and cooperative information navigation. As we embed more semantics into the Web, we need to make a similar shift in perspective from the old "theorem proving" model of inference, to a new model of dynamic and cooperative reasoning and problem solving.

Here's a caricature of what we consider the dominant paradigm of Information Retrieval field to have been prior to the advent of the Web. The old Information Retrieval model was that information was stored in a "database", essentially a large, static big bag of "records" and "fields" that changes only slowly [via "database updates"]. The user's only option for interfacing with the database is to throw a "query" at it, a statement of the user's interests in a formal query language like SQL. The user was expected to have in mind a Platonic "ideal document" that was described by the query. The job of the database was to return one or more documents in the database ordered by how well they satisfied the query. Today's search engines are the modern manifestation of this paradigm.



The traditional Information Retrieval paradigm

What's wrong with this paradigm for the Web? Well…

- *It's difficult for users to express queries precisely.* In the old days of IR, users were expert librarians who formulated Boolean queries in formal languages. Now, ordinary users type one or two words to search engines. These don't express intent precisely

- *There may not be a "best document" in the Web.* Any query can potentially return an infinite number of documents. You can't tell which is "best" from a single query alone.

- *Query-response interaction is a "batch processing" view.* It is sequential - either you or the retrieval system are working, not both at the same time. Except for query refinement, each query and response is essentially an independent event, and unrelated to anything else you might be doing.

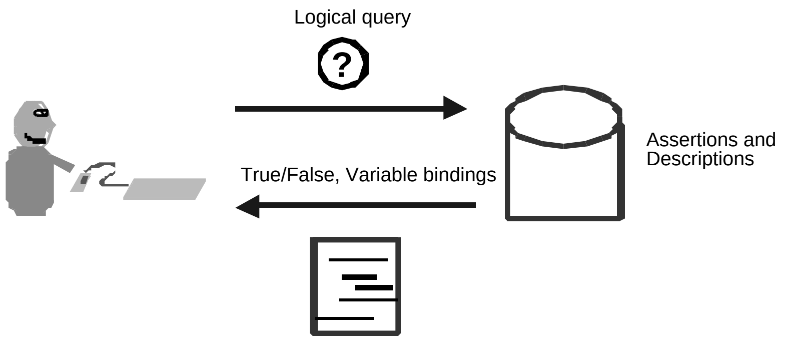
There are also many other critiques of the old IR paradigm, but that will suffice for the moment.

Agent software on the Web breaks all of these assumptions. Agents like Letizia and Powerscout track the history of user browsing and use it as a persistent context from which to infer user interests. Rather than globally ranking documents, they present context-dependent suggestions that can improve over time. Rather than use the query-response paradigm, they are always active, and deliver their recommendations in a continuous stream in real-time. They essentially integrate browsing and searching into a unified process.

The goal is not to find the best document, but to make the best use of the user's time. In short, they treat browsing as a continuous, co-operative activity between one or more humans and one or more software agents.

The theorem-proving paradigm

Adding semantics to the Web allows agents to do problem-solving by using traditional theorem-proving techniques to process assertions and descriptions stored on the Web. Again, though, we believe that the paradigm has to change to accommodate the dynamic and fluid nature of the Web. By analogy, we present a caricature of the old theorem-proving paradigm.



The traditional theorem-proving paradigm

The old theorem-proving paradigm treated a knowledge base as a big bag of assertions and descriptions expressed in a logical language. Again, the way the user was assumed to interact with this was to issue a logical query in the form of an assertion to be proved true or false, possibly filling in the values of logical variables contained in the assertion. This would launch an inference procedure that would find "the answer".

What's wrong? Well…

- *It's difficult for users to express logical queries precisely.* We can't expect users of Web applications to be mathematicians. We need to have interfaces that interact with the user and help formulate queries before they can be sent to an inference system.

- *There may not be a "best answer".* How much is that plane flight? Well, when are you asking? Do you have a discount? Should we ask for a price on Priceline? Buy one on Ebay? How long are you willing to wait for the answer? Can you leave a day earlier? … We need the ability to put procedural hooks in the answers.

- *Query-response interaction is a "batch processing" view.* Again, in the standard view, no opportunity for dynamic user input or user-agent cooperation is provided for.

  Once again, the modern view should be that problem-solving and inference should be a cooperative venture between one or more humans and one or more computer agents. Interaction and parallelism should be available at any point in the process. After all, the expensive component is not the net connection, the client computer or even the servers that store the information, it's the user's *time*. This means that the components of the future Semantic Web should be able to integrate not only text, pictures, links, and semantic descriptions, but also dynamic and procedural objects. But how?

The Semantic Web shouldn't sit on the Tower of Babel

The approach to the Semantic Web advocated in this book via languages like DAML and OIL aims to leverage existing Web standards like XML and RDF, and add new facilities that allow expressing formal semantic information embedded in traditional HTML documents and enabling inference. This has the advantage that encoding semantics in the Web becomes an evolutionary and not disruptive step in the evolution of the Web. However, procedural information has not been covered by any of the existing proposals.

Not my department?

One approach is simply to say, as the current standards do, that it is out of the scope of Web standards to dictate how Web applications will be programmed. Fear of the divisiveness of arguments over programming languages in the computer science community has been motivating this abdication of responsibility for encoding of procedures.

But look where it's gotten us. Many of today's Web applications are programmed piecemeal in a bewildering array of programming languages, for example,

- Javascript
- Java
- Perl
- VBScript
- PHP

just to name a few. It is not uncommon for what appears to the end user as a single Web application to use, in total, 7 or 8 of these languages. The disadvantages of such multi-language environments should be obvious.

- *Difficulty of learning multiple programming languages.* Not only does it duplicate the effort multiple times of learning syntax and semantics, but there is no rational relationship between the languages, e.g. Perl has nothing to do with Java. Maddeningly, despite their names and superficial similarity, Javascript and Java contain major differences in their type and object systems. Having to learn two or more to get a task done is more than twice as hard as one language because the interface between the languages is always additional hair, usually poorly documented. You also must learn *when* to use one language or the other.

- *Difficulty of integrating Web data with multiple languages.* It is worthwhile to keep in mind that the majority of these languages were originally designed for a purpose that had nothing to do with the Web. Java was designed as a control language for small devices; Javascript was a scripting language for Netscape; VBScript as an extension of Basic for Microsoft desktop applications; Perl, for string manipulation via regular expressions. They were all pasted together in an unprincipled manner.

- *Necessity of conversion between different formats.* Conversion wastes computation time and storage and opens up the possibility for errors due to mismatches in data semantics.

- *Difficulty of debugging across multiple languages and systems.* If something goes wrong with a multi-language program, how do you tell who's at fault? None of these languages has anything approaching a decent debugger, even for programs written solely in that language, so if more than one language is involved, it becomes a nightmare.

And, perhaps the worst problem as far as Web semantics is concerned, is that the plethora of languages prevents representing procedural semantics in any sort of principled way. We need consolidation of existing Web functionality.

We need another language like a hole in the head

New Web languages seem to be born monthly. Rather than introduce a new language for implementing some specialized functionality, we recognize that sooner or later that functionality will need general purpose utilities like conditional execution, loops, functional abstraction, etc. When these are added on to special purpose languages late in the game, elegance is compromised by compatibility with the original specialized language and we end up with a mess.

Instead, we propose, as DAML/OIL does for static Web semantics, to gently extend the semantics of the widely accepted HTML for markup and XML for static data, to encode procedural semantics in Web pages. The extension is called Water.

Water is a language for the web that embodies the three primary functionalities needed for general purpose information manipulation into one unified language:

- *Markup.* Since Water is a superset of HTML, it inherits all its capability.

- *Data*. Water permits the description of persistent structured data on the web via XML like syntax yet having the capability of dynamically computing values that may contain self-referential interconnections.

- *Code*. Water is a general purpose object oriented programming language that is, at its core, more flexible than Javascript, VBScript and Java.

  Water provides a way to define functions and call them in addition to defining and instantiating objects. The object system is a multiple-inheritance, prototype-based object system with annotations. Thus it is not merely a vanilla procedural/object oriented general purpose language grafted on to HTML and XML, but rather a language whose very core supports the intimate mixture of unstructured content, structured data and active values through a highly dynamic object system.

Is procedural attachment rope to hang yourself?

Advocates of declarative representations always get nervous when someone advocates letting the user call a procedure from any point in a declarative representation. There is the danger that the code can have unpredictable effects, and that the reasoning systems cannot guarantee anything about the behavior of descriptions that contain embedded code. In this view, procedural attachment is giving the user rope to hang themselves.

To some extent, that's true. But we should recognize that many operations that Web applications will want to do will inevitably go beyond purely declarative representations. I/O operations, especially network I/O, user interface operations that interact in real time with the user, and side effects to shared data structures, often have this character. If we disallow procedures to be embedded in the declarative representation, we are merely pushing the procedural information outside of the declarative representation entirely. While this might gain some cleanliness in the declarative part, this doesn't alter whatever properties of unpredictability the entire system, procedures and data, has as a whole. And, after all, that's what we're really interested in.

Procedural markup

A better approach is to allow procedures to be embedded in data, accepting the risk, but to also encourage the programmer to do what we might call *procedural markup*, analogous to markup for text. Markup for text [e.g. <b>] supplies additional declarative annotation that serves as advice to the renderer about how to display the text. Markup for procedures is to include with the embedded procedures a declarative representation of the result of the procedure, so that the reasoning system can operate on it. For example, a procedure that computes the price of an airline ticket might assert that the result is a positive number in US dollars.

Of course, the results of the reasoning will only be valid if the markup accurately describes the result of the procedure. Although it is impossible in general to predict the result of an arbitrary program, in some cases, automatic program verification might actually be able to either generate the assertions automatically or prove that the code actually satisfies the assertions. To enable this, it is best to keep the code close at hand to the places where it is used. And for debugging, it is invaluable to have a seamless interface that integrates procedures and data so the programmer can see if there is a mismatch between what the program produced and what he or she is expecting.

Water's provides a smooth path from text to code

Like HTML, Water permits you to just write a paragraph of English text and call it a web page. Also like HTML, you can take another pass and add markup without having to rewrite your original text. You need only learn the HTML tags that you need to know.

However, should you require more dynamic behavior, with Water you needn't learn a new scripting syntax such as Javascript, or worse, a whole new programming environment such as Java. You can simply insert Water tags into your page using the same basic syntax that your markup is in. Plain text, markup, data, and code can be freely intermixed with no barriers between those functionalities and no "impedance mismatches". Water tries hard to give the user the most functionality per learning effort, emphasizing economy and elegance at the expense of hairy and rarely used features.

Who's the user?

The target audience for Water is the web site author who needs more than HTML can offer yet is intimidated (and rightly so) by multi-language programs. Although Water can fulfill most of the needs of the general purpose programmer, someone who's spent years learning a difficult language like C or Java is hard to convince to give up on their investment even if they can exceed their previous productivity in a few weeks. The HTML author looking for more flexibility is an easier sell. And there are millions of them. But the suitability of Water is not limited to toy programs. Water is not simplified at the expense of being able to accommodate complex programming tasks.

Water's distinguishing features

Water has a unique approach to bridging the apparent gap between static and dynamic semantics by unifying what in other languages are objects and function calls. Yet it doesn't start off by throwing today's web author in cold water either.

- *Minimal differences between markup, data, and code.* We have already discussed this point.

- *Minimal differences between classes and instances.* Water's object system uses prototypes instead of classes [Lieberman 86]. No need to learn separate operations for classes and instances since there's no distinction between a class and an instance. An object can be used "as an instance", and/or it can be "subclassed" with the new object inheriting the desired characteristics of its parent(s) while other characteristics are modified to differentiate the new object from its parent(s).

  The language Self [Ungar & Smith 87] had a simple and elegant prototype-based object system. Self's object system differs from Water in that it was only single-inheritance and had a "copy down" semantics for inherited values rather than the "dynamic look-up" mechanism of Water. The dynamic lookup of Water makes object creation faster and uses less memory than "copy-down". It also preserves locality of reference for shared data which may have additional speed advantages. But if "copy-down" is desired, the "init" function for object creation of a given parent object can choose to copy down desired fields.

- *Minimal difference between "instance variables" and "methods".* Each are stored as values of fields in an object. You can get the value of an instance variable or a method by just getting the value of a named field. Methods are implemented by objects just like everything else in the language. To call a method, you use the function calling syntax which looks like using an HTML tag.

- *Minimal differences between a function call and object creation.* The tag name refers to the parent object. The parameters after the tag name serve as either field initializers or function arguments depending on the value of the tag.

- *Minimal differences between aggregate data types.* Objects, vectors and hash tables are all represented by the same data structure. An object is a set of fields whose keys can be any object [interned strings, numbers, other objects]. When you're treating an object as a vector, a looping mechanism permits looping over all fields whose keys are integers. A "size" field makes finding out the length of the vector quick as well as facilitating "adding an element to the end of the vector".

- *No differences between the object system and the type system.* For example, "integer" can be thought of as a "type" but its really just another object in the object tree that happens to be the parent object of all integers. You can reference all objects used as types via the path syntax, just like you can reference all fields of objects. In fact, since any object can become a parent, any object can effectively be a type.

- *Minimal differences between initialization and normal methods.* An *init* method can, in fact, return any type of object and should declare the returned type of object.

- *Minimal [but very significant] differences between local variables, fields, and subobjects* for creating, setting and referencing.

- *Minimal differences between a field and an annotation to that field.* Water provides a way to add information ABOUT a field. Say you have a field named "color" in an object named "car". You might want to add a comment about the field such as "all colors besides black and white cost extra" or declare that the type of the field must be an integer. Annotations are the mechanism that Water uses to associate information about a field with that field. Annotations of a field "foo" are indicated by another field with a naming convention of "foo@annotation-name". A looping mechanism makes it easy to find all annotations of a given field or to ignore all annotations of an object when you want to loop through an object's field values.

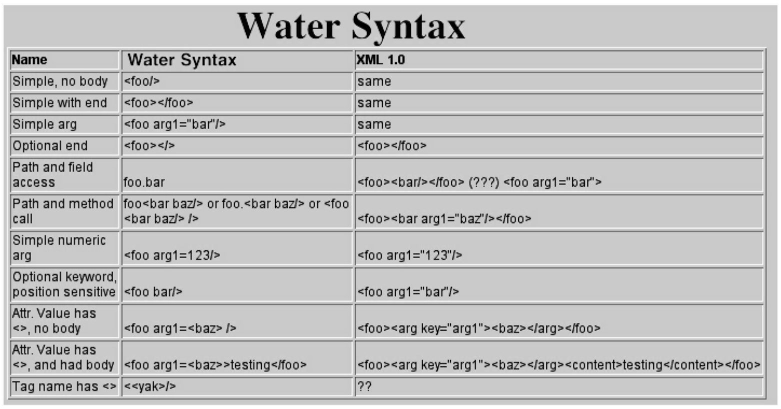
- *Water provides an easy way to specify the "evaluation kind"* of each parameter in a method definition. This controls how each argument in a call to the method is interpreted. It can be treated as code and evaluated, which is the default. It can be treated as an expression to be parsed but not evaluated, which is especially good for "delayed evaluation" where a method takes code as an argument and can evaluate the code when it chooses. It can be treated as the string of its source code (and not even parsed), and hypertext, which treats text within angle brackets as code to evaluate and other text just as strings (especially convenient for implementing HTML tags as Water objects or method calls).

- *Water permits the parent of an object to be changed at runtime.* This is not a common situation, but makes it easy for a parent to "adopt" a child.

- *Water allows optional keywords.* In HTML you usually use keywords to specify each attribute. XML requires the use of keywords. In Java and Javascript you can't use keywords to specify any argument. Water permits you to use keywords when you want to be more explicit or pass arguments "out of order" and pass arguments by position (without keywords) when you want to be more concise. There is no extra overhead for declaring keywords when defining a method, they are simply the parameter's name.

- *Water permits "Active Values" on object fields.* These facilitate simple constraint systems, and, along with other dynamic features, help to implement specification changes after most of the implementation is done by permitting field references and setters to turn into method calls without changing the referencing and setting code at all.

Water is especially easy to write interactive programming environment tools for, due to its extensive introspection capabilities, evaluation kinds, elegant object system, uniform syntax, ultra-dynamic behavior and small size. Below, we present a summary of Water syntax, for those interested in the details. An example of Water code follows.



Water Syntax

Name	Water Syntax	XML 1.0
Simple, no body	<foo/>	same
Simple with end	<foo></foo>	same
Simple arg	<foo arg1="bar"/>	same
Optional end	<foo></>	<foo></foo>
Path and field access	foo.bar	<foo><bar/></foo> (???) <foo arg1="bar">
Path and method call	foo<bar baz/> or foo.<bar baz/> or <foo <bar baz/> />	<foo><bar arg1="baz"/></foo>
Simple numeric arg	<foo arg1=123/>	<foo arg1="123"/>
Optional keyword, position sensitive	<foo bar/>	<foo arg1="bar"/>
Attr. Value has <>, no body	<foo arg1=<baz> />	<foo><arg key="arg1"><baz></arg></foo>
Attr. Value has <>, and had body	<foo arg1=<baz>>testing</foo>	<foo><arg key="arg1"><baz></arg><content>testing</content></foo>
Tag name has <>	<<yak>/>	??

Water syntax table

A Water Example

Water permits arbitrary code intermixed with HTML. Here's a snippet of ordinary HTML:

```
<font size="3">Hello World</font>
```

With Water we can stick code anywhere within this such as an attribute value:

```
<font size=1.<random 7/> > Hello World</font>
```

Water really has two syntaxes, a pure XML syntax and one that permits some short cuts such as "by position arguments" which makes using attribute names unnecessary. Both syntaxes can be used in the same body of code.

Here's a more complex example of generating a catalogue for a store. First we set up our inventory "database". This might come from another file or another method call but for simplicity we enter it directly as a vector of vectors like so:

```
<set root.inventory
  <vector
   <vector "Flannel Top Sheet"  "17.95"/>
   <vector "Satin Pillow Case"  "11.45"/> /> />


<set page <p>For our everyday unbeatable price we
have</p> />


root.inventory.<for_each>
      page.content.<push field_value.0/>
      page.content.<push " for the low low price of "/>
      page.content.<push field_value.1/>
      page.content.<push <br/> />
</for_each>
```

At the end of running this program the "page" variable contains our page object. We can get the string of HTML like so:

```
page.<to_html/>
```

Let's add some variability based on the day of the month:

```
<set root.is_sale_day <date/>.day_of_month.<same 1/> />
```

Now is_sale_day is true if its the first day of the month and false otherwise.

```
<set page <if root.is_sale_day <p>On Sale today we
have</p>
              true <p>For our everyday unbeatable price
we have</p>
          </if> />
```

The heading to our page is now customized, so let's do the same for the body:

```
root.items.<for_each>
      <set price
       field_value.1.<times <if> root.is_sale_day .75
                                 true 1
                                 </if>
```

```
                      />
          />
          page.content.<push field_value.0/>
                             " for the low low price of "/>
                             price
                             <br/>
                          />
    </for_each>
```

Above the price will be multiplied by .75 if we're on the first of the month.

Comparison with Java

Some of Water's features provide fixes for inconsistencies and other unnecessary complexities in Java [Lemay & Cadenhead 01].

- Unlike Java, all Water data types are full-fledged objects. There are no "ints" and "Integers", just "integers".
- Unlike Java, Water has true multiple inheritance. There is no extra "interface" mechanism.
- Unlike Java, Water supports both a prefix syntax and an infix syntax -automatically. Water does not need precedence rules or parentheses.
- Unlike Java, every object knows its own type, including objects in vectors. There is no need for converting objects to different types when you put them into, and extract them from, a vector.
- Unlike Java, the types of fields and method return values need not be declared. They default to the most general type of object. However, type declarations are encouraged as they help declare the programmer's intent and can be used by programs to find inconsistencies in the programmer's intent as well as speedup execution.
- Unlike Java, the "init" method is just like a regular method. Its body returns a value [usually but not necessarily a subobject of the called object]. It is named and has a return type declaration just like a regular method.
- Unlike Java, instance variables can be created or removed within an object at run time.
- Unlike Java, you can add methods to a system class (or any other), extending its behavior without having to edit and recompile the original source. Water gives the programmer more flexibility in modularity so that a method for a class need not reside in the same file as the class definition if that's what the programmer chooses.
- Unlike Java, Water's "evaluation kinds" permit the easy construction of high-level code manipulation methods which in many cases reduce the complexity of packaging up and calling advanced functionality. This enables application programmers to write their own control structures and other methods that can "delay evaluation" until a programmer-chosen time.

- Unlike Java, there is rarely a need to create a "singleton", ie a class with one instance object. That's because in Water, a "class" object can be treated just like an instance object. This is due to Water's elegant prototype object system.You can call its methods directly using the "class" as "this". There is no need to have a distinction between static variables and instance variables, nor static and instance methods. And each "instance" object in Water can get its own special version of method, if the programmer so chooses, to distinguish its behavior from that of its sibling objects. "Static" is another concept you just don't have to know about in Water.

- Compared to Java, Water is much more consistent with itself. For example, Java has three different syntaxes for programatically determining the length of an array, a vector and a string. Water has one.

Comparison with Javascript

Water has a lot in common with Javascript [Goodman 00]:

- Water and Javascript can be embedded in HTML.

- Water and Javascript are interpreted.

- Water and Javascript both have "eval" though Javascript doesn't have a "parsed but not evaled" representation using for programs that manipulate code.

- Water and Javascript objects can both have fields added dynamically, after an object is created.

But there are a lot of differences as well:

- Water can be "more" embedded than Javascript within HTML. Generally speaking, Javascript can only be attached to certain control attributes of Javascript tags. Water permits active code just about anywhere within HTML.

- Water's syntax is much more like HTML than Javascript's.

- The object system in Javascript is designed to be a "prototype" object system. Yet it is, at the very best, poorly documented. Water's prototype object system is easier to use.

- Water permits (but does not force) the declaration of types for arguments and object fields. Javascript does not permit type declarations.

- Water has much more flexible argument definitions which can take default values and "evaluation kinds" in addition to names and types.

- Water has a simple, concise syntax for referencing objects through long paths in the object hierarchy and interconnected field values.

Conclusion

We are now at a crossroads in the evolution of the Web. The Web has evolved from a relatively static collection of pages and links, to a dynamic, interactive interface to semantic information. We are at the verge of being able to create the Semantic Web, in terms of declaratively representing objects that are already human-readable on the Web. Next, we need to make it the Dynamic Semantic Web by encoding procedures in Web material as first-class objects.

We've presented an argument for dynamic semantics, along with a language, Water, that integrates procedures seamlessly into Web pages, just as XML and DAML/OIL integrate descriptions. Some may think we place too much emphasis on the language. It is true that good environments can be built on top of mediocre languages, if you dedicate a great deal of effort. The current Web itself may be viewed an example of that. But great environments need a language that supports the easy construction of dynamic, interactive and introspective tools. The difference between good environments and great environments is tens or even hundreds of percents in speed of implementing reliable, maintainable programs. Increasing productivity of Web applications is the ultimate goal. We can do this not just by handing existing programmers more powerful tools, but by giving people who consider themselves non-programmers, as many of today's HTML authors do, the power to radically customize their computer.

References

Bauer, Dengler & Paul 01	Mathias Bauer, Dietmar Dengler, and Gabriele Paul, Programming by Demonstration for Information Agents, in [Lieberman, ed. 01].
Goodman 00	Danny Goodman's Javascript Handbook, IDG Books, 2000.
Kushmerick, Weld & Doorenbos 97	Nicholas Kushmerick, Daniel S. Weld, Robert Doorenbos, Wrapper Induction for Information Extraction. Intl. Joint Conference on Artificial Intelligence (IJCAI), 1997.
Lemay & Cadenhead 01	Laura Lemay and Rogers Cadenhead, Teach Yourself Java 2 in 21 Days, Sams Press, 2001.
Lieberman 86	Henry Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems , First Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-86], ACM SigCHI, Portland, Oregon, September 1986.
Lieberman, ed. 01	Henry Lieberman, ed., Your Wish is My Command: Programming by Example, Morgan Kaufmann, San Francisco, 2001.
Lieberman, Fry & Weitzman 01	Why Surf Alone? Exploring the Web with Reconnaissance Agents, Communications of the ACM, to appear, 2001.
Ungar & Smith 87	Self: The Power of Simplicity. ACM Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-87].

Woods 75 Woods, W. (1975), What's in a Link: Foundations for Semantic Networks, in D.G. Bobrow & A. Collins (eds.), Representation and Understanding, Academic Press.