AspectScript: Expressive Aspects for the Web

Rodolfo Toledo Paul Leger Éric Tanter[*]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
http://www.pleiad.cl

ABSTRACT

JavaScript is widely used to build increasingly complex Web applications. Unsurprisingly, these applications need to address crosscutting concerns. Therefore support for aspect-oriented programming is crucial to preserve proper modularity. However, there is no aspect-oriented extension of JavaScript that fully embraces the characterizing features of that language: dynamic prototype-based programming with higher-order functions. In this paper, we present AspectScript, a full-fledged AOP extension of JavaScript that adopts higher-order programming and dynamicity as its core design principles. In AspectScript, pointcuts and advices are standard JavaScript functions, bringing the benefits of higher-order programming patterns to define aspects. In addition, AspectScript integrates a number of state-of-the-art AOP features like dynamic aspect deployment with scoping strategies, and user-defined quantified events. We illustrate AspectScript in action with several practical examples from the realm of client Web applications, and report on its current implementation. AspectScript is a practical extension that provides better modularity support to build Web applications, and that will eventually make it possible to empirically validate the benefits brought by advanced aspect language mechanisms in an ever-growing application domain.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*

General Terms

Languages, Design

Keywords

AspectScript, JavaScript, aspect-oriented programming, higher-order programming, Web applications, scoping strategies, quantified events.

[*]Partially funded by FONDECYT project 1090083.

1. INTRODUCTION

There is a clear trend in the software industry towards Web-based applications, as witnessed by the increasing number of popular Web-based applications like Facebook, YouTube, and Blogspot. For the development of these applications, the JavaScript language is one of the most used, mainly because almost all modern browsers support it. A consequence of this trend is that JavaScript, which was initially used only for small client-side scripting, is now used to build complex applications. In such applications, crosscutting concerns are likely to appear and end up being scattered at many places in the code, tangled with other concerns.

While modularization of crosscutting concerns has long been considered in Web technologies [28] (*e.g.* separate CSS files for presentation style), there is not much support at the scripting code level. Aspect-Oriented Programming (AOP) [12] addresses this issue by introducing new means of modularizing programs, in particular through the pointcut-advice mechanism. The potential benefits of AOP for JavaScript have already been identified and resulted in a number of aspect-oriented proposals for JavaScript [1, 4, 7, 14, 25, 36]. However, these proposals are fairly basic in that they at best attempt to mimick AspectJ [16] in JavaScript. However, beyond the fact that it is dynamically typed, JavaScript is a language that is also fundamentally different from Java. As a result, existing proposals fail to properly integrate with the characterizing features of JavaScript, *i.e.* a dynamic prototype-based object model with full support for higher-order functions.

In contrast, AspectScript builds upon advances in AO language design research to address the specificities of JavaScript in a novel way. Inspired by the work on aspects in higher-order procedural languages by Dutchyn, Tucker, and Krishnamurthi [34, 11], AspectScript supports first-class aspects; both pointcuts and advices are defined using first-class functions, providing the full benefits of higher-order programming. In line with this work and the inherently dynamic nature of JavaScript, AspectScript supports dynamic deployment of aspects, as found for instance in CaesarJ [3] and AspectScheme [11]. Also, AspectScript integrates scoping strategies [30, 31] for proper control over the scope of dynamically-deployed aspects. In addition, AspectScript avoids issues of infinite loops due to aspect reentrancy [29]. Finally, AspectScript not only supports implicitly-generated join points following the language model, but also provides the possibility to define custom join points triggered explicitly, as in Ptolemy [26]. This combination of features is unique in the space of current aspect languages.

This paper puts a strong emphasis on the practical benefits of integrating these features in an AO extension of JavaScript. Section 2 therefore exposes several concrete examples addressed with AspectScript, progressively revealing how each feature comes into play. Section 3 then reviews the main elements of the AspectScript

language in a systematic manner. Section 4 presents key points regarding the implementation of the core of AspectScript, and Section 5 describes how two features, custom join points and scoping strategies, have been added to the core language. Section 6 discusses related work and Section 7 concludes.
Availability. AspectScript is available online [17] and currently supports Mozilla Firefox.

2. ASPECTSCRIPT IN ACTION

We introduce AspectScript through concrete examples. More precisely, we implement a number of extra functionalities to Facebook, a representative Web-2.0 application. For each of the five cases, we show how an aspect-oriented solution with AspectScript enables a modular and straightforward specification.

For conciseness, in the code fragments, we use the variables AS and PCs as abbreviations for AspectScript and AspectScript.Pointcuts respectively. All the examples are included with the distribution, and can be downloaded from the AspectScript website [17].

2.1 A Simple Example

In Facebook, users can freely tag people (supposedly) appearing in photos. If the user being tagged is a friend of the tagger, the tag becomes a link to the user profile, and the picture is added to the personal photo album. However, to avoid wrong tags, Facebook provides a way to remove them through a "remove tag" option.

For most cases, this simple solution is satisfactory. However, if a user has been wrongly tagged in hundreds of photos, he has to remove each tag individually. Let us devise an extension to Facebook that suggests an automatic removal of all tags once a user untags herself from a certain number of pictures (say 3) in a given album. This feature can be implemented as an aspect in AspectScript:

```
var pc  = PCs.exec(removeTag).and(nTimes(3));
var adv = function(jp) {
  var userId = jp.args[0]; //1st argument to removeTag
  var albumId = jp.args[1]; //2nd argument to removeTag
  showRemoveAllTagsDialog(userId, albumId);
};
AS.after(pc, adv); //deployment
```

This simple example illustrates how to define and deploy a simple aspect: defining a pointcut pc, an advice adv, and finally deploying the aspect. AspectScript provides AS.before/around/after(..) functions, which are syntactic sugar for manually creating and deploying an aspect with a particular advice kind. The aspect is deployed with global scope (more on this later).

The aspect matches three executions of the removeTag function. This function is parametrized by the id of the user, the id of the album, and the id of the photo (userId, albumId, and photoId respectively). The advice adv removes all tags of the user in the current album, after confirmation. The variables that identify the user and album are obtained from jp.args, which stores the values with which the function was executed. The pointcut pc is a conjunction of two pointcuts. The first is obtained using the exec pointcut designator (*i.e.* a function that returns a pointcut), and matches all executions of the removeTag function. The second pointcut is obtained using the nTimes pointcut designator. It matches whenever it is evaluated a given number of times. The nTimes PCD is not part of the standard PCD library; it is defined as a higher-order function:

```
var nTimes = function(n) {
  var times = 0;
  return function(jp) {
    return ++times % n == 0;
  };
};
```

The actual conjunction of pointcuts is performed using the and method of the pointcut returned by exec(removeTag). This shows how pointcuts provided by AspectScript support a *fluent interface*[1] for operands such as or, and, and inCflow. Note how the combination of first-class functions and fluent interfaces allow for an extensible pointcut language within standard JavaScript syntax.

2.2 Pointcuts: Matching Sequences

While the previous aspect definition appropriately modularizes the new feature, it is a typical example of a *stateful* aspect [10], implemented with book-keeping code (the times variable in the pointcut). Recognizing this fact, we now present an alternative implementation that uses a general-purpose trace PCD for matching *sequences* of events:

```
var rt = PCs.exec(removeTag);
var adv = function(jp) { /* as above */ };
AS.after(trace(rt,rt,rt), adv);
```

Note that the pointcut rt only matches execution of removeTag, and that at deployment time, we actually specify that we are interested in a sequence of three such events.

The higher-order trace function is a pointcut designator that receives a list of pointcuts. This list specifies the sequence of join points to match. Interestingly, trace is also simply defined within AspectScript as follows[2]:

```
var trace = function() { //variable-arg function
  var state = 0;
  var pcs = arguments; //actual arguments
  return function(jp) {
    if(pcs[state](jp)) {
      if(state == pcs.length-1) {
        state = 0;   // reset state
        return true; //match
      }
      state++;  //go to the next pointcut
    }
    return false;
};};
```

Note that the pointcut resets its state after matching the whole sequence (our previous definition with nTimes did not reset). The trace pointcut designator is mostly illustrative. It is far from a full-fledged tracematch mechanism in which sequences can be defined with regular expressions and free variables, and where multiple *instances* of a sequence can be matched simultaneously [2].

Exposing context information. In order to expose context information to either the advice or other pointcuts, a pointcut can take as optional parameter an environment, and define new bindings in it. The pointcut returns the environment if it matches[3]. For instance, we can define a time PCD that always matches any join point and only extend the given environment with a new binding associating a given identifier to the current time:

[1]Fluency is a structuring principle to make APIs closer to embedded DSLs, see http://martinfowler.com/bliki/FluentInterface.html

[2]In JavaScript, it is possible to omit the declaration of formal parameters. The actual parameters can then be accessed using the implicit variable arguments, as an array.

[3]Returning true in a pointcut has the same meaning as returning the same environment originally passed to the pointcut.

```
function time(id) {
  return function(jp,env) {
    return env.bind(id, new Date().getTime());
  };
}
```

We can now revisit our simple trace PCD to allow pointcuts in a sequence to expose context information to be used in later pointcuts:

```
var trace = function() {
  //... as above
  var env = AS.emptyEnv;
  return function(jp) {
    match = pcs[state](jp,env);
    if(match) {
      env = match;
      if(state == pcs.length-1){ state = 0; return env; }
      // ... as above
```

Note how trace now passes a (local) environment to each pointcut in the sequence, and keeps the bindings defined by previous pointcuts. This progressively builds up a sequence-local environment. When the whole sequence finally matches, the environment is returned. We can use this enhanced trace PCD to express the fact that the removal of tagging advice is triggered only if the three tag removals are performed within a given time interval:

```
var timeDiff = function(jp,env) {
  return env.t1 - env.t0 < 10000; // 10 seconds
};

var seq = trace(rt.and(time("t0")),rt,rt.and(time("t1")));
var timedSeq = seq.and(timeDiff);
AS.after(timedSeq, adv);
```

We define a timeDiff pointcut that expects t0 and t1 to be bound in the environment and checks the time difference. We then define the sequence pointcut using trace, making sure that the first and third occurrences of rt bind the current time as expected. This brief example obviously does not take into account all the intricacies of a proper tracematch mechanism, but it does illustrate the flexibility brought by first-class, user-definable pointcuts in AspectScript.

2.3 Giving Life to JavaScript Values

In Facebook, user pages are updated with the last messages from friends without the need to reload the page. To achieve this, Facebook uses Ajax [13]: a set of Web development techniques for asynchronous client-to-server communication. However, when page elements (*e.g.* the last messages) depend on Ajax responses, they must be updated after every communication with the server to show up-to-date information. The complexity of Ajax updates can be alleviated by introducing basic support for *reactive values*: when a value originating from an Ajax response changes, all page elements depending on it are updated accordingly. Using AspectScript, we introduce a library for basic reactive values, *e.g.*:

```
var msgs = new ReactiveValue("lastMsgs.php?id=...", 3000);
newMessages.innerHTML = msgs;
```

We create a new reactive value msgs, passing the URL from which the value must be obtained, as well as the update interval. We then assign this value to the innerHTML property of the page element displaying new messages. Every time the reactive value is updated, the property is automatically updated.

The library has two elements: the ReactiveValue constructor and an aspect that takes care of the assignment of reactive values to page elements. The aspect, defined below, intercepts all assignment join points where the left-value is a page element and the right-value is a ReactiveValue object. Instead of proceeding, the around advice stores the join point in the reactive value itself. Therefore, the assignment proper does not happen yet.

```
var pc = function(jp) {
  return jp.isPropWrite() &&
    isDomElement(jp.target[jp.name]) && //left-value
    jp.value instanceof ReactiveValue;  //right-value
};
var adv = function(jp) {
  var reactiveValue = jp.value;
  reactiveValue.add(jp); //store jp for future executions
};
AS.around(pc, adv);
```

A reactive value therefore holds a list of assignment join points, corresponding to assignments to page elements that must be refreshed whenever the value changes. This is done each interval milliseconds by scheduling the Ajax query. Whenever the new value of the Ajax query is ready, onreadystatechange is applied. This function simply invokes proceed on all the assignment join points:

```
function ReactiveValue(url, interval) {
  var ajaxResquest = ...; //new remote connection
  var jpList = [];  //list of assignment join points
  function query() {
    //make request
  }
  this.add = function(jp) {
    jpList.push(jp);
  };
  ajaxRequest.onreadystatechange = function() {
    for(var i = 0; i < jpList.length; ++i)
      jpList[i].proceed(ajaxRequest.responseText);
  };
  //schedule query for repetition
  setInterval(query, interval);
}
```

As a result, all page elements that directly depend on a reactive value are reassigned (join points are re-proceeded) each time the value changes. Note that this simple library for reactive values does not support full-fledged reactive computation (including indirect dependencies), like Flapjax [22]. This said, the library is still useful to address direct dependencies between values

2.4 Access Control with Scoping Strategies

Most modern Web applications allow third-party applications to provide extra functionality through an API. However, one of the most attractive features of Facebook is the ability to include them right *inside* Facebook pages, and since recently, third-party applications can use JavaScript to provide a richer user experience. Sadly, JavaScript can also be used by malicious applications to fool users. For instance, the following application tries to change the "home" link to point to an external page, identical to the login page of Facebook, thereby misleading the user to reinsert his access credentials:

```
var maliciousApplication = {
  fakeURL : '123.45.56.78/facebook.com',
  action : function() {
    var homeElem = ...;
    homeElem.href = this.fakeURL;
  } };
```

To avoid these kinds of applications, Facebook limits the area of the Web page that an external application can access. Suppose that a function isOutsideAppArea(elem) exists for checking whether a DOM element elem is outside the area bounds of an application page, and hence its access is forbidden[4]. Using AspectScript, we can build a modular solution based on aspects:

[4]The actual implementation in Facebook rewrites the application code to replace all references to objects in the DOM by references

```
var forbiddenElement = function(jp) {
  var elem = jp.target;
  return isDOMElement(elem) && isOutsideAppArea(elem);
};
var fa   = PCs.get("*").and(forbiddenElement);
var forbid = function(jp) {
  throw "forbidden access";
};
var securityAspect = AS.aspect(AS.AROUND, fa, forbid);
AS.deployOn(securityAspect, maliciousApplication);
```

AS.aspect is used to create the securityAspect. The fa pointcut of aspect matches all accesses to DOM elements that are outside the page area of the application, and the forbid around advice immediately throws an exception. In this example, securityAspect is deployed using deployOn. The semantics of *per-object* deployment are the same as in *e.g.* CaesarJ [3] and AspectJ (*e.g.* perthis): the aspect "sees" all join points occurring *lexically* within the methods of maliciousApplication.

Although the securityAspect aspect prevents the maliciousApplication object from directly accessing elements outside of its page area, it does not forbid the application to *indirectly* access. This is because the aspect does not see the join points that are not lexically within any method of maliciousApplication. For instance, it can use an external function to change the home link:

```
var maliciousApplication = {
  //... as above
  action : function() {
    setHomeLink(this.fakeURL); //indirect modification
} };
```

While the above pattern could be detected by using a control flow check, the malicious application can also schedule setHomeLink(..) for later execution, thereby completely invalidating any kind of control flow reasoning:

```
//modification scheduling in 100 ms
setTimeout(function(){ setHomeLink(this.fakeURL);}, 100);
```

In both cases discussed above, per-object aspect deployment is unable to properly prevent the malicious application to perform its evil deed. A solution to this problem is to use a more expressive scoping strategy for the security aspect [30]. In particular, we want to deploy the aspect in the maliciousApplication with a *pervasive* scoping strategy [31]:

```
AS.deployOn(securityAspect, maliciousApplication, pervasive);
```

In general, a scoping strategy specifies how an aspect propagates along the call stack as well as in newly-created procedural values (objects and functions in JavaScript)[5] [30, 31]. The pervasive strategy simply states that the aspect *always* propagates in both function applications and newly-created procedural values.

In our example, this ensures that the security aspect is propagated unconditionally on the stack and therefore sees all accesses to elements in the control flow of any method of maliciousApplication. The security aspect is propagated to all new procedural values as well, so the anonymous function passed to setTimeout is also considered potentially malicious. Therefore, using the pervasive scoping strategy, the security aspect prevents maliciousApplication from directly *and* indirectly accessing elements outside of its page area.

to objects that check whether an application is trying to access an element outside its area. We use the isOutsideAppArea abstraction in this example for the sake of simplicity.

[5] A scoping strategy also permits to refine the pointcuts of an aspect for a particular deployment, but we do not need this feature here.

2.5 Identifying New Kinds of Events

In Facebook, a user can interact with applications, events, messages, and many other elements. When a user interacts with one of these elements, a description is added to the user wall (*i.e.* a single page overview of recent user activity). Wall reporting is clearly a crosscutting concern; it can be modularized using an aspect:

```
var pc = ...; //user interactions with elements
var adv = function(jp) {
  var elem = ...; // extract elem from jp
  wall.add(elem.getDescription()); //desc. of the element
};
AS.after(pc, adv);
```

Specifying the appropriate pc pointcut is not straightforward. The standard approach is to define pc as the union of all calls (or executions) of the functions "corresponding to" interactions of the user with Facebook elements. Nevertheless, this does not always work. For instance, when the user adds an already-added application, the change is discarded. This is reflected in the code of addApplication:

```
function addApplication(app) {
  if (!alreadyAdded(app)) {
    //1. check compatibility between the user and the app
    //2. synchronous Ajax call to register the app
    //3. add the app to the apps bar
} }
```

Defining a pointcut that matches when addApplication *effectively* adds an application is not straightforward. For these kinds of cases, AspectScript provides custom join points as in Ptolemy [26], whose underlying idea is to trade obliviousness for precision and abstraction. Using a custom join point, the addApplication function can signal precisely when it actually adds an application:

```
  if (!alreadyAdded(app)) {
    AS.event("addApp", {elem: app}, function() {
      //steps 1, 2, 3 as above
    }); }
```

A custom join point is generated by a call to AS.event, passing three parameters: an event type (addApp), an object containing contextual information provided to pointcuts and advices ({elem: app}), and a block of code (a thunk) specifying the base code associated to the join point. With this custom join point, the specification of the pc pointcut above is straightforward:

```
var pc = PCs.event("addApp").or(/*other interactions*/);
```

The PCs.event matches all custom join points with the specified name. The context information can be accessed as properties of the join point: in this example, jp.elem is the app object.

2.6 Summary

This informal presentation of AspectScript has illustrated several features of AspectScript through concrete applications. We have shown: how to define and create aspects with different advice kinds; how to compose pointcuts and define custom PCDs, taking advantage of higher-order programming patterns; how to expose context information; how join points can be used as first-class values, stored in data structures and proceeded multiple times; how to deploy aspect globally, per-object/function, and to control the scope of deployed aspect with scoping strategies; and how to use custom join points to identify new points of interest. The following section undertakes a more systematic exposition of the main features of AspectScript.

join point	points in the program execution at which...
new	a function or object is created.
init	a function or object is initialized.
call	a function is called.
exec	a function is applied.
p-read	an object property is read.
p-write	an object property is written.
event	AS.event(..) is called.

Figure 1: Dynamic join points of AspectScript.

3. A TOUR OF ASPECTSCRIPT

We now describe AspectScript in more details, reviewing its hybrid join point model (Section 3.1), followed by the aspect model (Section 3.2), and the deployment model (Section 3.3). Finally, Section 3.4 describes how aspect reentrancy is controlled.

3.1 Hybrid Join Point Model

At its core, AspectScript adopts a join point model in the line of that of AspectJ [16], but tailored for the JavaScript language. As in any aspect-oriented language, join points are generated implicitly upon certain evaluation steps of the program. Pointcuts can then quantify over these join points. While quantification is a crucial feature of aspect languages, the obliviousness brought by implicitly-generated join points is more controversial. In particular, it suffers from a number of problems related to the difficulty of reconciling the (low) level of abstraction of standard join points with the need for quantifying over application-specific events. To this end, AspectScript also includes a mechanism for explicitly triggering custom join points, in the style of Ptolemy [26]. To the best of our knowledge, combining both implicit and explicit join point generation is a distinguishing feature of AspectScript in the current design space of aspect languages.

Standard join points. Figure 1 presents the standard join points supported in AspectScript. Save the last kind, which corresponds to custom events, all these join points are generated implicitly when a related expression is about to be evaluated. The JavaScript object model is peculiar in several respects. In particular, any function is considered a method of an object: top-level functions and anonymous functions are considered methods of the global root object of JavaScript. In addition, executing this.foo() in the context of an object o, where foo is a method of o, is different from invoking simply foo(). In the latter case, foo executes in the context of the global object, not of o. Also, a function is an object per-se, which can have properties on its own. AspectScript considers these peculiarities when generating join points, setting the target and current object properties of the join point appropriately.

Custom join points. In addition to standard, implicitly-generated join points, AspectScript also supports explicitly-generated custom join points. This mechanism corresponds to typed events in Ptolemy [26]. It addresses limitation of both implicit invocation and aspect-oriented languages, by making it possible to treat any (block of) expression(s) execution as an event that can be quantified over by pointcuts. Events have a type that closely corresponds to their intended (application-specific) semantics, rather than being tied to the base language operational semantics. It is also possible to communicate an arbitrary set of context information to other pointcuts and advices without introducing unnecessary coupling to program details. In AspectScript, typed events are supported as custom join point with a type attribute. This attribute can be a string (JavaScript does not support symbols) or any object. Because AspectScript does not modify the syntax of JavaScript, the (block of) expression(s) that corresponds to the custom join point is defined in a thunk. In addition to the type and the thunk, a custom join point has an arbitrary number of properties, which can then be used in poincuts and advices. The generation of a custom join point with AS.event was illustrated in Section 2.5. Aspects perceive custom join points like standard ones. Similarly, if no aspect apply, the original computation takes place, *i.e.* the specified thunk is applied.

3.2 Higher-Order Aspects

AspectScript is directly inspired by AspectScheme [11], in which aspects, pointcuts, and advices are first-class values. Consequently, they can be created and manipulated at runtime. As illustrated in Section 2, an aspect in AspectScript is a pointcut-advice pair; pointcuts and advices are plain JavaScript functions.

Pointcut model. Following standard practice, it is standard to define a pointcut as a function that takes a join point as parameter and returns an environment if it matches, or false if it does not [19]. The environment is used by pointcuts to pass information to the corresponding advice. A peculiarity of AspectScript in this respect is that pointcuts are also parameterized by an environment. This permits inner pointcuts of a given pointcut to communicate using the environment. For instance, in the pointcut (pc1 && pc2), the environment returned by pc1 (if it matches) is then passed to pc2. We have illustrated the use of this feature in Section 2.2[6].

Figure 2 presents some of the pointcut designators available in AspectScript. Because pointcuts are standard JavaScript functions, their definition does not rely on any additional syntactic constructs. Also, pointcut designators take full advantage of the potential of higher-order functions. No standard pointcut in AspectScript modifies the environment. Pointcuts exposing typical contextual information like this, target, and args in AspectJ would be redundant here because these values are available as join points properties (*e.g.* jp.target, jp.args), as illustrated in Section 2.

It is important to notice that AspectScript pointcuts do not rely on meta-data like types or annotations in order to match join points. The reason is that JavaScript does not support types nor annotations. AspectScript pointcuts also avoid using variable names to discriminate first-class values; rather they rely on value identity. This can be seen in the definitions of the call and exec pointcut designators in Figure 2, where the reference equality operator (===) is used to compare functions. Identity-based comparison is important in a language where functions are first-class values, and hence can be bound to many names, or none at all. Name-based selection in this context can result in many false negatives (*e.g.* a function execution not matched because it is applied through an alias) as well as false positives (*e.g.* a variable name that is used to refer to different functions at different moments in time).

Advice model. Advices in AspectScript are functions parameterized by a join point and an (optional) environment. The join point object passed as parameter has a proceed method, which permits the execution of the original computation at the join point. In the case of a custom join point, proceed evaluates the associated thunk. The environment corresponds to the environment returned by the associated pointcut. Like pointcuts, advices can access the bindings in the environment (*e.g.* in the example of Section 2.2, the advice can access both t0 and t1).

[6]The examples of Section 2 also make use of the fact that pointcuts can return true instead of the empty environment, and that all function parameters in JavaScript are optional.

```
function call(fun) { //call
  return function (jp,env) {
    return (jp.isCall() && jp.fun === fun)? env : false;
  } }

function exec(fun) { //execution
  return function (jp,env) {
    return (jp.isExec() && jp.fun === fun)? env : false;
  } }

function set(target,name) { //property write
  return function(jp,env) {
    return (jp.isPropWrite() &&
      jp.target == target && jp.name == name)? env : false;
  } }

function cflow(pc) { //control flow
  return function cflow(jp,env) {
    if (jp == null) {
      return false;
    }
    return pc(jp,env)? env : cflow(jp.parent,env);
  } }

function not(pc) { //negation
  return function(jp,env) {
    //'not' does not return the (possibly) modified env.
    return (pc(jp,env) != false)? true : false;
  } }
```

Figure 2: Some pointcuts available in AspectScript.

AspectScript supports a basic scheme for the composition of aspects: when several aspects apply over the same joint point, the applications of their advices are nested like in AspectJ. Like in AspectScheme, the precedence for the application of nested advices is determined by the order in which the aspects are deployed: the last-deployed aspect goes first. We come back on aspect weaving in Section 4.2, when describing the implementation of AspectScript.

3.3 Deployment and Scoping Strategies

Dynamic deployment of aspects has been shown to enhance reuse and to better support software variability in general [23, 27]. A number of aspect languages and frameworks hence support dynamic aspect deployment, under different flavors and scoping semantics. In fact, a language with higher-order aspects but without dynamic deployment makes little sense [11]: if aspects can be crafted at runtime (*i.e.* by a higher-order function), it should definitely be possible to *deploy* them at runtime. Furthermore, controlling the scope of aspects is crucial, not only for software variability. When analyzing the potential problems that can arise when composing modules containing woven aspects, McEachen and Alexander make clear that developers need more control over scoping of aspects [21]. We believe this is all the more important in a dynamically-typed setting. AspectScript supports dynamic aspect deployment, further refined with scoping strategies [30].

Dynamic Deployment. As in CaesarJ [3] and AspectScheme [11], AspectScript supports dynamic deployment of aspects. Deployment options are presented in Figure 3. The first option is global scope: deploy(a) deploys aspect a such that it sees all join points in the execution of the program, until it is explicitly undeployed with undeploy(a). Internally, deploy (resp. undeploy) just adds (resp. removes) an aspect to the global aspect environment, globalAspects.

In order to deploy an aspect with dynamic scope (*e.g.* fluid-around in AspectScheme), deploy can take a thunk (no-arg function) as an extra argument. In this case, the aspect sees all join points in the dynamic extent of the evaluation of the thunk. Internally, this variant of deploy temporarily adds the aspect to the global environment, executes the thunk, and then removes the aspect:

```
function deploy(aspect, fun){
  globalAspects.add(aspect);
  var r = fun();
  globalAspects.remove(aspect);
  return r;
}
```

This mutation-based implementation is necessary because JavaScript does not support dynamic binding.

The third deployment option is per-value deployment, which follows the semantics of per-instance deployment in *e.g.* CaesarJ and AspectJ (per-this). The scope of an aspect deployed using deployOn is lexical, therefore the aspect only sees join points occurring lexically within method bodies of the objects it is deployed on. If deployed on a function, the aspect sees all join points in the body of the function, including inner functions.

Deployment expression	Scope
deploy(asp) / undeploy(asp)	global
deploy(asp, fun [,ss])	on block: dynamic by default
deployOn(asp, val [,ss])	on object: lexical by default

Figure 3: Aspect deployment in AspectScript. Scoping strategies can be optionally specified in the last two alternatives.

Scoping Strategies. Both deployment on a block (deploy) and deployment on values (deployOn) can be refined using a scoping strategies, specified as a last parameter. Scoping strategies [30, 31] permit fine-grained control over the *scope* of a dynamically deployed aspect. A scoping strategy is a triple of functions [c,d,f]. c (resp. d) is propagation function specifying whether or not an aspect propagates along the call stack (resp. in newly-created functions or objects). f is an activation function making it possible to filter out certain join points. In other words, c permits to stop the unconditional propagation of dynamic scope at certain points, d enables aspects to be captured in certain procedural values as they are created, and f specifies a deployment-local refinement of the aspect pointcut.

All three functions are boolean-returning functions that take a join point as parameter[7]. In the case of c, the join point is the call the aspect can propagate through; in the case of d, the join point is the creation of the new object or function; and in the case of f, the join point is the one subject to filtering. As an example, the pervasive scoping strategy [31] used in the example of Section 2.4 is defined as follows:

```
var pervasive = [true,true,true];
```

This strategy specifies that the aspect propagates unconditionally on the call stack (c always evaluates to true). Similarly, the aspect propagates to all new procedural values (d always evaluates true). Finally, the filter function f always evaluates to true as well, therefore no filtering is performed and the aspect sees all join points. Many examples of scoping strategies have been formulated elsewhere, for both local aspects [30] and distributed aspects [33], as well as variable bindings [31].

[7]In AspectScript, c, d and f can be specified directly as values; this is syntactic sugar for the corresponding constant function.

3.4 Control of Aspect Reentrancy

Thus far we have ignored a fundamental issue with the proposed design of AspectScript: if a function application triggers a join point, and a pointcut is a plain JavaScript function, then applying a pointcut pc triggers a function application join point against which pc should itself be evaluated, leading to an infinite loop! And the same happens with an advice that applies a function whose application is matched by its associated pointcut. In fact, while this issue of *aspect reentrancy* is exacerbated in a higher-order aspect language like AspectScript, it is latent in any aspect language[8]. Some mechanism must be provided to avoid aspects potentially matching join points triggered by their *own* execution [6, 29].

Limitations of current solutions. Higher-order procedural aspect-oriented languages like AspectML and AspectScheme adopt different solutions to this problem, though both rely on a mechanism to deactivate weaving in some way. Indeed, AspectScheme uses a primitive operator app/prim to apply a function without generating a join point, and AspectML suggests a similar disable primitive that hides the whole computation of an expression. The difference is their scope: app/prim only hides a single function application join point, but does not hide the computation triggered by that application; conversely, disable has dynamic scope. Current AspectJ patterns to address these issues are similar to disable: adding a control-flow condition to pointcuts such that join points occurring in the dynamic extent of advice execution are ruled out [6]. The AspectJ pattern however does not work in the case of reentrancy caused by if pointcuts [29].

As argued extensively elsewhere [32], relying on control flow checks to avoid reentrancy is flawed for several reasons. Most importantly, since proceed is called as part of an advice, reasoning on control flow conflates advice computation and base computation, resulting in aspects not applying when then should. Also, disabling weaving for aspectual computation simply makes it impossible for aspects to advise aspects.

Solution in AspectScript. In order to properly address issues of reentrancy without entailing conflation or sacrificing visibility of aspect computation, AspectScript relies on the notion of *execution levels* [32], a refinement of the proposal of stratified aspects [6]. In a nutshell, the idea is to structure computation into levels, starting with base computation at level 0. Aspect computation is by default considered as ocurring at level 1, and is therefore invisible to other aspects. If needed, aspects can be deployed at higher levels of execution, thereby possibly observing other aspects, or an aspect can explicitly lower part of its computation (if so, a mechanism ensures that it does not see its *own* computation). A detailed description of the implementation of reentrancy control and execution levels is however outside the scope of this paper. In-depth motivation and formal description of execution levels can be found in [32].

The bottom line is that, for the programmer, AspectScript just works as if the issue of reentrancy did not exist, *precisely* because AspectScript handles execution levels behind the scene. We have not illustrated advanced scenarios with explicit level shifting, such as aspects of aspects (see [32]); rather we have focused on illustrating the simplicity of the default, most common cases. All the pointcuts in Section 2 and Figure 2 are defined naturally, without having to resort to primitive operators like app/prim and disable. Similarly, advices can perform any computation and they never enter infinite loops caused by aspect reentrancy.

[8]http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html

4. IMPLEMENTATION

In this section we detail the implementation of AspectScript. In particular, Section 4.1 details the code transformation phase required for weaving, and Section 4.2 describes the weaving process. Finally, we end with a preliminary analysis of AspectScript performance in Section 4.3.

Given the dynamic nature of JavaScript, weaving in AspectScript is mostly done at runtime. In order to be able to dynamically weave aspects without modifying a particular JavaScript engine, our AspectScript implementation first performs a code transformation phase in which some expressions are rewritten into invocations of *reifiers* (Section 4.1). These reifiers then make it possible to perform runtime aspect weaving (Section 4.2). Relying on code transformation means that AspectScript can run on any client browser, without requiring the installation of dedicated complements or add-ons. Currently, our AspectScript implementation has been tested with the Mozilla Firefox browser (versions 3.0.* and 3.5.*). The complete transformation phase is implemented in JavaScript, using an optimized version of the parser of the Narcissus project[9]. A JavaScript implementation is interesting in order to be able to do client-side parsing; this could be useful for example in the case of remote code loading or eval invocations. These applications are subject of further exploration.

4.1 Code Transformation

The code transformation phase rewrites JavaScript source code[10] by introducing invocations of *reifiers*. Reifiers are functions that generate the join points associated with the rewritten expression.

For the reader unfamiliar with JavaScript, Figure 4 presents a subset of its syntax, relevant to the transformation performed by AspectScript (we substitute fun for function to save space). A script is a list of statements, which can be either a function or variable declaration, a block or an expression. Expressions include object creation (either with a constructor or literally), definitions of arrays and anonymous functions, function invocation and variable/property access and assignment.

$$\begin{array}{rcll}
\mathit{Script} & z & ::= & \overline{s} \\
\mathit{Statement} & s & ::= & \mathsf{fun}\ id(\overline{id})\ \{\overline{s}\} \mid \mathsf{var}\ id = e \mid \{\overline{s}\} \mid e; \\
\mathit{Expression} & e & ::= & \mathsf{new}\ e_c(\overline{e}) \mid \{\overline{id : e}\} \mid [\overline{e}] \mid \mathsf{fun}(\overline{id})\{\overline{s}\} \\
 & & & \mid e_f(\overline{e}) \mid e_t.id(\overline{e}) \\
 & & & \mid id \mid e.id \mid id = e \mid e_t.id = e
\end{array}$$

Figure 4: Subset of the JavaScript syntax.

The transformation is defined by the syntax-driven rewriting function $[\![\cdot]\!]$, presented in Figure 5. The first four rules deal with statements. Rule 1 recursively triggers the transformation of the expression being used to initialize the declared variable. A function declaration is rewritten into a var declaration, binding the function name to the (transformation of) the anonymous function definition (rule 2). Rule 3 groups functions declared in a block, and moves them to the beginning of that block. This reordering is necessary because of rule 2: in JavaScript a variable needs to be *explicitly assigned* to a value in order to be used, whereas a function declaration makes the function name available globally in its declaring block. Finally, rule 4 rewrites the expression part of the statement.

[9]http://mxr.mozilla.org/mozilla/source/js/narcissus/

[10]AspectScript per se does not extend the *syntax* of JavaScript; as illustrated in previous sections, it is provided as a library.

Statements

$$\llbracket \text{var } id \text{ = } e \rrbracket = \text{var } id \text{ = } \llbracket e \rrbracket \tag{1}$$
$$\llbracket \text{fun } id_f(\overline{id})\ \{\overline{s}\} \rrbracket = \text{var } id_f \text{ = } \llbracket \text{fun } (\overline{id})\ \{\overline{s}\} \rrbracket \tag{2}$$
$$\llbracket \{\overline{s}\} \rrbracket = \{ \llbracket \overline{s}_{fun-decls} \rrbracket \llbracket \overline{s}_{other-decls} \rrbracket \} \tag{3}$$
$$\llbracket e; \rrbracket = \llbracket e \rrbracket; \tag{4}$$

Object Creation

$$\llbracket \text{new } e_c(\overline{e}) \rrbracket = \rho_{new}(\text{fun}(k,\overline{a})\{\text{return new } k(\overline{a})\}, \llbracket e_c \rrbracket, [\llbracket \overline{e} \rrbracket]) \tag{5}$$
$$\llbracket [\overline{e}] \rrbracket = \rho_{new}(\text{fun}(k,\overline{a})\{\text{return } [\overline{a}]\}, \text{Array}, [\llbracket \overline{e} \rrbracket]) \tag{6}$$
$$\llbracket \text{fun}(\overline{id})\ \{\overline{s}\} \rrbracket = \rho_{wrap}(\text{fun}()\{\text{return fun}(\overline{id}) \llbracket \{\overline{s}\} \rrbracket) \tag{7}$$
$$\llbracket \{\overline{id : e}\} \rrbracket = \rho_{lit}(\text{fun}()\{\overline{\text{this}.id = \llbracket e \rrbracket}\}) \tag{8}$$

Function Invocation

$$\llbracket e_f(\overline{e}) \rrbracket = \rho_{call}(\text{AS.globalObject}, \llbracket e_f \rrbracket, [\llbracket \overline{e} \rrbracket]) \tag{9}$$
$$\llbracket e_t.id(\overline{e}) \rrbracket = \rho_{call}(\llbracket e_t \rrbracket, \llbracket e_t.id \rrbracket, [\llbracket \overline{e} \rrbracket]) \tag{10}$$

Property Access

$$\llbracket e.id \rrbracket = \rho_{read_p}(\llbracket e \rrbracket, \text{"}id\text{"}) \tag{13}$$
$$\llbracket e_t.id \text{ = } e \rrbracket = \rho_{assign_p}(\llbracket e_t \rrbracket, \text{"}id\text{"}, \llbracket e \rrbracket) \tag{14}$$

Figure 5: Rewriting function.

The next rules rewrite expressions. The general scheme is the same: all the information required to generate a join point is gathered and passed as parameters to the appropriate reifier, which then triggers weaving (Section 4.2). Object creation (rule 5) is rewritten into an invocation of the ρ_{new} reifier, passing as argument a first-class anonymous function that encapsulates the actual instantiation (parametrized by the constructor and actual arguments). This function is then used at runtime for the proceed method of the constructed join point. The same approach is used for array creation (rule 6). The reifier also receives the constructor and the arguments. In order to support generation of function execution join points, functions are wrapped; thus, rule 7 uses a different reifier ρ_{wrap}. Finally, literal object creation (rule 8) uses yet another reifier, ρ_{lit}. This reifier receives as argument a first-class anonymous function that encapsulates the initialization of the properties specified in a given literal object creation.

The remaining rules are straightforward. Function invocations (rules 9 and 10) are transformed into invocations of the ρ_{call} reifier, passing as parameters the target of the call, the function being invoked, and its arguments. Rule 9 uses the global object as a target because the JavaScript semantics specifies that when no target is specified when invoking a function, the global object must be used. Rules for property access (13 and 14) transform reads and writes of properties into invocations of the ρ_{read_p} and ρ_{write_p} reifiers, passing as parameter the name of the property being accessed, and in the case of a property write, the value being assigned. The owner of the property is passed as the first argument to both reifiers.

4.2 Runtime Weaving

At runtime, the calls to the reifiers inserted by rewriting are evaluated. Apart from creating the corresponding join point, the reifiers also trigger the weaving process by invoking the weave function of AspectScript. To illustrate reifiers, below is the (simplified) implementation of the ρ_{call} reifier:

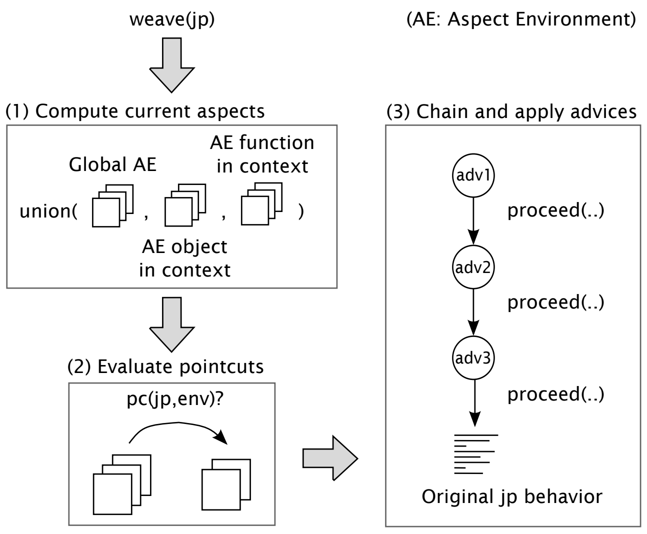

Figure 6: Weaving process.

```
function r_call(obj, fun, args){
  var jp = new CallJP(obj, fun, args, currentJoinPoint);
  return weave(jp);
}
```

The r_call function receives as arguments the target of the call (obj), the function being invoked (fun), and the arguments to that function (args). This is the JavaScript code executed to weave a call expression, corresponding to the transformation rules 9 and 10 in Figure 5.

The weave function weaves the join point it receives as argument. Figure 6 depicts the weaving process initiated by weave, and its simplified definition is as follows:

```
function weave(jp){
  // 1. compute current aspects
  var currentAspects =
     union(globalAspects,aspectsIn(ctxObj),aspectsIn(ctxFun));
  // 2. evaluate pointcuts
  var advices = match(jp, currentAspects);
  // 3. chain and apply advices
  return chainAndApply(advices);
}
```

The first step is to determine the set of aspects that may potentially apply. Recall from Section 3.3 that AspectScript supports different deployment mechanisms: global, per-function, and per-object[11]. In consequence, at a given join point, the list of aspects that may apply is the *union* of the aspects that are deployed in several aspect environments (step 1). First, the *global* aspect environment contains all aspects deployed using deploy (either deployed globally or with dynamic scope). In addition, aspects may have been deployed with deployOn, therefore each object and function has its own aspect environment. At a given join point, aspects in the currently-executing function (ctxFun) and the currently-execution object (ctxObj) have to be considered (accessed with aspectsIn in the code above).

Once the list of current aspects has been determined, pointcuts of all these aspects are evaluated against the current join point (step 2). The advices of aspects that matched the current join point are then chained together, each one nesting the next one (step 3). Executing proceed in one advice triggers the following advice; in the last advice in the chain, proceed runs the original base computation.

[11]Scoping strategies are an extension to AspectScript (Section 5.2).

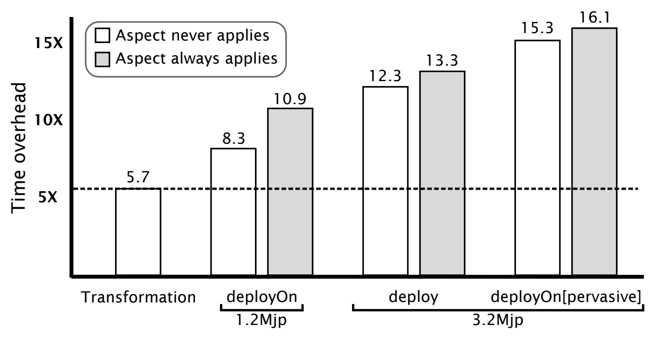

Figure 7: Performance overhead of AspectScript for CPU-intensive tests in the JQuery test suite.

Note that following AspectScheme [11], before and after advices are only syntactic sugar for around advices. Aspects are deployed in the reverse order of their deployment, like in AspectScheme.

4.3 Runtime Performance

The primary design goal of AspectScript is expressiveness. When conceiving the language, we have not sacrificed any potentially valuable feature on the basis of its expected cost. Still, we are interested in making AspectScript a practical solution for aspect-oriented programming in JavaScript; it therefore makes sense to evaluate its overhead. In order to do so, we selected a subset of the jQuery [15] test suite (+1300 tests, around 90% of the total number of tests) that includes only CPU-intensive tests, as this constitutes a worst-case scenario for AspectScript. For this experiment, we used jQuery 1.3.2 (118KB of source code) on an Intel Core 2 Duo, 2.66 GHz PC with 2GB of RAM running Ubuntu 9.04 (kernel 2.6.28) and Firefox 3.5.2. Running the selected tests takes approximately 20s in this setting, without AspectScript.

We then transformed the jQuery library as described in Section 4.1 in order to support aspects. The transformed code ended up weighing 326KB (a 2.8x factor). Just running the test suite with the transformed library, without any aspect weaving at all, is 5.7 times slower. This overhead includes the generation of approximately 3.4 million join points (Mjp). Considering this baseline, Figure 7 describes the overhead introduced by an aspect following different deployment scenarios: *(a)* a globally-deployed aspect, *(b)* an aspect deployed with the pervasive scoping strategy we used in Section 2.4, and *(c)* an aspect deployed on an object (the $ object, entry point of the jQuery library). In both cases *(a)* and *(b)*, the deployed aspect sees all 3.4Mjp; in case *(c)*, due to lexical scoping, the aspect only sees 1.2Mjp. The advice of the aspect only calls proceed on the join point. For each case, we measured the overhead with both a pointcut that never matches and a pointcut that always matches, thus giving a lower and an upper bound for the overhead of the presence of the aspect. We did experiments where the pointcut does a function comparison (using reference equality) instead of just returning a boolean, without any noticeable difference.

The overhead of a global aspect is between 12.3x (never match) and 13.3x (always match). Activating scoping strategies and deploying a pervasive aspect implies a relative overhead of 20% only (15.3x-16.1x). Using a per-object aspect (which results in the aspect seeing only 1.2Mjp) reduces the overhead down to 8.3x-10.9x. The results are overall encouraging, especially considering the low amount of time we spent working on optimizing the implementation so far; our focus has rather been to get the semantics right and working. We believe there are many venues for optimization to be explored, both in the runtime itself and by adding the possibility for the programmer to statically declare certain aspects and restrictions on the general dynamicity of AspectScript. Finally, raw overhead in a CPU-intensive scenario does not really reflect the fact that JavaScript is more widely used for interactive applications, that may even include remote communication. For instance, we have tested AspectScript (global aspect always matching) on a JavaScript Tetris game, without any noticeable difference[12].

5. EXTENSIBILITY

In addition to being a practical aspect language, AspectScript is also a useful medium for experimenting with new aspect-oriented constructs. While as of now AspectScript does not provide any particular support for modular language extension per se, the conciseness of its core implementation (1700 loc, comments included) makes it easy to localize extension points. We now report on our experiment in extending our initial implementation of AspectScript in order to support both custom join points as in Ptolemy [26] (Section 5.1), and scoping strategies [30, 31] (Section 5.2). Our motivation to integrate both features is eventually to gather empirical evidence about their use in practical scenarios.

5.1 Custom Join Points

As illustrated previously, the generation of custom join points is done by invoking AS.event passing as parameters the type of the custom join point (just a string), the arguments it exposes, and the block of code to which the join point corresponds (Section 3.1). Therefore, AS.event plays the role of an explicit reifier, similar to those introduced in Section 4.1. Rather than being introduced through code transformation, the programmer explicitly inserts them where needed. Not surprisingly, the definition of event is very similar to that of a reifier:

```
AS.event = function(type, ctx, block){
  return weave(new CustomJP(type, ctx, block));
};
```

Just like any other join point, CustomJP needs to implement the proceed method. In the case of custom join points, proceed must execute the specified block:

```
function CustomJP(type, ctx, block){
   ...
   this.proceed = function(){
      return block();
   };
}
```

The proceed function invokes the block of code the join point was created with. The code presented above is the essence of the custom join points extension. The real implementation is the same except for a few performance optimizations. The complete code of the extension is only 56 lines of code.

5.2 Scoping Strategies

Adding scoping strategies to AspectScript was a significant extension, though compact to implement. This extension shows how to better control the scope of deployed aspects; and to do so, a deployed aspect requires extra attributes. Recall from Section 3.3 that a scoping strategy specifies the propagation of an aspect by means of both a call stack propagation and delayed evaluation functions (resp. c and d), and its activation by means of a filter function (f). Adding these three properties to a deployed aspect object is trivial in a dynamic prototype-based language like JavaScript:

[12]Both versions of the Tetris game can be tested online on the AspectScript website [17].

```
function deploy(aspect, fun, ss){
  ss = normalize(ss); //turn boolean exprs. into functions
  aspect.c = ss[0]; aspect.d = ss[1]; aspect.f = ss[2];
  //... as in Section 3.3
}
```

To implement the scoping strategies semantics, we use two functions: enterSS and exitSS, invoked at the beginning and at the end of weave, respectively. Both take as parameter the list of current applicable aspects (Figure 6) and the current join point; they may update the list to reflect the scoping specifications.

Let us first look at how call stack propagation is supported. Before weaving, enterSS checks if the current join point is a call, and if so, it checks which aspects should propagate, by evaluating their c propagation function:

```
var removedAspects = ...; //stack for nested calls
function enterSS(asps, jp){
  if(jp.isCall()){
    var removed = [];
    for(var i = 0; i < asps.length; ++i){
      if(!asps[i].c(jp)){
        asps.remove(asps[i]); //remove from current aspects
        removed.add(asps[i]); //but remember it
    } }
    removedAspects.push(removed);
} }
```

If an aspect does not propagate, it is removed from the list; all removed aspects are kept on a stack so as to be reinstalled when the evaluation represented by the join point returns (the removedAspects stack enables the support for nested calls), in exitSS:

```
function exitSS(asps, currentJP){
  if(currentJP.isCall()){
    var removed = removedAspects.pop();
    asps.add(removed);
} }
```

Taking into account delayed evaluation propagation is simpler; it is only necessary to add the following condition to enterSS:

```
function enterSS(asps,currentJP){
 //... as above
 if(currentJoinPoint.isInit()){
    for(var i = 0; i < asps.length; ++i){
      if(asps[i].d(currentJP)){
        AS.deployOn(asps[i], currentJP.target);
      }
} } }
```

If the current join point is an init, the d function of each current aspect is evaluated. Whenever d evaluates to true, the corresponding aspect is deployed on the object being created, using deployOn.

The code presented above is the essence of the scoping strategies extension. The actual code is only 65 lines in total, including the implementation of filtering (the last component of a deployment strategy), not included here for space reasons.

6. RELATED WORK

Modularization of crosscutting concerns has long been considered in Web technologies [28]. An example of this is the separation of an HTML document in different sources, such as CSS files for presentation style and JavaScript files for functionality. However, within these sources, crosscutting concerns are still present. For this reason, diverse tools to modularize these concerns are available in the Web. For instance, the jQuery library [15] allows programmers to separate crosscutting concerns in the DOM of a Web page, *e.g.* for adding borders to all tables. We now review AOP frameworks for JavaScript, as well as AOP proposals for other higher-order procedural languages.

AOP for JavaScript. A large number of lightweight AOP frameworks for JavaScript have been made available on the Web by programmers, that rely on function wrappers. In the simplest case, there is no quantification at all, and one explicitly has to give both a function and the advice function that should be added to it with wrapping [25]. Some frameworks make it possible to wrap methods of an object by specifying the names of the methods to wrap [4, 5, 14], and others support regular expressions for describing method names to wrap [1, 7].

AOJS [36] is a more mature AOP framework that takes quantification more seriously, and relies on code transformation. However, it does not embrace the features of JavaScript as AspectScript does. Aspects (with before and after advice only) are specified in a separate XML file; therefore, aspects cannot enjoy the full power of higher-order programming. In addition, function identification is name-based, rather than identity-based. As discussed in Section 3.2, this leads to fragile pointcuts, that may (not) match when expected, and excludes the execution of anonymous functions. The join point model of AspectScript is also considerably richer, and extensible. AOJS does not support dynamic deployment of aspects. While this is beneficial in terms of performance, it is limitating in terms of expressiveness. We plan to improve performance of AspectScript in the future by supporting static specifications to restrict full dynamicity to a certain extent. Finally, aspect reentrancy in AOJS is avoided by simply deactivating weaving during pointcut and advice evaluation, an insufficient solution (Section 3.4).

AOP for higher-order procedural languages. AspectScript draws significantly on previous work on AOP extensions of higher-order procedural languages like Scheme, Standard ML and Caml[13].

AspectScheme [11] is an aspect-oriented extension of Scheme, which is, like JavaScript, dynamically typed. In order to support the full power of higher-order programming, pointcuts and advices in AspectScheme are standard functions, aspects are dynamically deployed (either with dynamic or lexical scope), and function identification is identity-based rather than name-based. AspectScript imitates AspectScheme in all these dimensions. This said, AspectScript supports a richer join point model, including custom join points. Context exposure in AspectScript is more powerful, allowing pointcuts in a composition to use bindings exposed by prior pointcuts. Finally, AspectScript supports more expressive aspect deployment (deployOn) and scoping (scoping strategies).

AspectML [9] is an AO extension of Standard ML [24]. Like AspectScript, pointcuts are first-class values, but advices are not. Aspects are deployed with lexical scope only. AspectML does not use function identity to match pointcuts; instead, pointcuts use function names and argument types to match functions that are currently in lexical scope. Anonymous functions cannot be advised (although an as-yet-unsupported any keyword is discussed). As discussed before, relying on function names in a language where functions are first-class values easily leads to both false positives and false negatives.

Aspectual Caml [20] extends Caml [18] with aspect-oriented constructs. Like AspectML, Aspectual Caml integrates well with the type system of the host language. In addition, Aspectual Caml takes into account the fact that Caml supports object-oriented programming. In essence, the aspect-oriented features of Aspectual Caml are very similar to AspectML, except for the fact that Aspectual Caml supports pattern matching over method names (in any scope), as well as advising anonymous functions. However pointcuts are not first-class values.

[13]We do not repeat here the limitations of the mechanisms provided by AspectScheme and AspectML to avoid reentrancy (Sect. 3.4).

7. CONCLUSION

Because JavaScript is being widely used in applications of ever-increasing complexity, it is particularly relevant to propose a powerful aspect-oriented extension. AspectScript is a first concrete, working step in this direction. AspectScript fully embraces the characteristic features of JavaScript, by supporting higher-order aspects, a full-fledged join point model, customizable quantified events, and dynamic aspect deployment with expressive scoping. Aspect reentrancy is avoided without causing any burden on programmers. This combination of features is unique in the current design space of aspect languages. We have illustrated the potential practical benefits of this language through a number of examples from the realm of client Web applications.

While several enhancements could be made to the language itself, like supporting customizable *implicit* join points [35], the major challenges for AspectScript are related to its practical adoption: portability, performance, and debugging support.

Browser integration. Currently, AspectScript has only been tested in Firefox. For AspectScript to be adopted in real-world applications, a version compatible with current browsers is necessary. We intuit that AspectScript can work on most existing browsers with minor modifications because its implementation is mostly based on the ECMA-262 specification[14].

Weaving optimization. Implementing aspect weaving in dynamic languages like JavaScript is hard to do efficiently [8]. The main issue is the inability to partially evaluate pointcuts, because extremely few conditions can be statically determined. Our initial benchmarks of AspectScript confirm the resulting cost. While many Web-based applications are highly interactive in nature and may not be so performance-sensitive, it is important to improve the current performance of AspectScript. Based on a preliminary study, we foresee three different lines for improvement: 1) supporting constructs that indicate to AspectScript that certain pointcuts do not need to be evaluated at every join point; 2) permitting the pre-declaration of aspects to allow AspectScript to partially evaluate them; 3) adding native support for aspects to the JavaScript engine itself, so as to avoid relying on code transformation.

Debugging. Currently, debugging an application instrumented by AspectScript is hard mainly for two reasons. First, because the code is rewritten and reordered by the transformation, the programmer ends up debugging a completely different application. Second, step-by-step examination of a running program needs to be aware of AspectScript in order to not jump to AspectScript internal code. We therefore plan to extend a popular debugging tool, like FireBug[15], to make it AspectScript-aware.

8. REFERENCES

[1] Ajaxpect. A JavaScript framework for aspect-oriented programming. http://code.google.com/p/ajaxpect/.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, pages 345–364, San Diego, California, USA, October 2005. ACM Press. ACM SIGPLAN Notices, 40(11).

[3] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[4] AspectJS. A function-call framework in JavaScript. http://www.aspectjs.com/.

[5] AspectJS. A JavaScript framework for aspect-oriented programming. http://zer0.free.fr/aspectjs/.

[6] Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition, 2006.

[7] Cerny. A javascript framework for method-call interception. http://www.cerny-online.com/cerny.js/.

[8] Thomas Cleenewerck, Kris Gybels, and Adriaan Peeters. Aspects in a Prototype-Based environment. In *Dynamic Aspects Workshop, AOSD 2004*, 2004.

[9] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

[10] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150, Lancaster, UK, March 2004. ACM Press.

[11] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.

[12] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[13] Jesse James Garrett. Ajax: A new approach to Web applications.

[14] Humax. A JavaScript framework for aspect-oriented programming. http://humax.sourceforge.net/.

[15] jQuery. A JavaScript library to manage event handling, animating, and Ajax interactions for the Web development. http://jquery.com/.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[17] Paul Leger, Rodolfo Toledo, and Éric Tanter. The AspectScript language. http://pleiad.cl/aspectscript, 2009.

[18] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Language Objective Caml: Caml supports functional, imperative, and object-oriented programming styles. http://caml.inria.fr/.

[14] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[15] http://getfirebug.com/

[19] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[20] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 320–330, Tallinn, Estonia, 2005. ACM.

[21] Nathan McEachen and Roger T. Alexander. Distributing classes with woven concerns – an exploration of potential fault scenarios. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 192–200, Chicago, Illinois, USA, March 2005. ACM Press.

[22] Leo Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2009)*, Orlando, Florida, USA, October 2009. ACM Press. To appear.

[23] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th Symposium on Foundations of Software Engineering (FSE-12)*, pages 127–136, Newport Beach, CA, USA, November 2004.

[24] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[25] Prototype. A JavaScript library that aims to ease development of dynamic Web applications. http://www.prototypejs.org/.

[26] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In Jan Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 155–179, Paphos, Cyprus, july 2008. Springer-Verlag.

[27] Hridesh Rajan and Kevin Sullivan. Eos: Instance-level aspects for integrated system design. In *Proceedings of ESEC/FSE 2003*, pages 297–306, Helsinki, Finland, September 2003.

[28] John Stamey, Bryan Saunders, and Simon Blanchard. The aspect-oriented Web. In *Proceedings of the 23rd annual international Conference on Design of Communication: documenting & designing for pervasive information*, pages 89–95, Coventry, United Kingdom, 2005. ACM.

[29] Éric Tanter. Controlling aspect reentrancy. *Journal of Universal Computer Science*, 14(21):3498–3516, 2008. Best Paper Award of the Brazilian Symposium on Programming Languages (SBLP 2008).

[30] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, April 2008. ACM Press.

[31] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*, pages 3–14, Orlando, FL, USA, October 2009. ACM Press.

[32] Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, March 2010. ACM Press.

[33] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 27–38, Charlottesville, Virginia, USA, March 2009. ACM Press.

[34] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In Mehmet Akşit, editor, *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 158–167, Boston, MA, USA, March 2003. ACM Press.

[35] Naoyasu Ubayashi, Hidehiko Masuhara, and Tetsuo Tamai. An AOP implementation framework for extending join point models. In Walter Cazzola, Shigeru Chiba, and Gunter Saake, editors, *Proceedings of the RAM-SE Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, pages 71–81. Fakultät für Informatik, Universität Magdeburg, 2004.

[36] Hironori Washizaki, Atsuto Kubo, Tomohiko Mizumachi, Kazuki Eguchi, Yoshiaki Fukazawa, Nobukazu Yoshioka, Hideyuki Kanuka, Toshihiro Kodaka, Nobuhide Sugimoto, Yoichi Nagai, and Rieko Yamamoto. AOJS: aspect-oriented JavaScript programming framework for web development. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 31–36, Charlottesville, Virginia, USA, 2009. ACM.