Supporting Dynamic, Third-Party Code Customizations in JavaScript Using Aspects

Benjamin S. Lerner *
University of Washington
blerner@cs.washington.edu

Herman Venter
Microsoft Research
hermanv@microsoft.com

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

Web sites and web browsers have recently evolved into platforms on top of which entire applications are delivered dynamically, mostly as JavaScript source code. This delivery format has sparked extremely enthusiastic efforts to customize both individual web sites and entire browsers in ways the original authors never expected or accommodated. Such customizations take the form of yet more script dynamically injected into the application, and the current idioms to do so exploit arcane JavaScript features and are extremely brittle. In this work, we accept the popularity of extensions and seek better linguistic mechanisms to support them.

We suggest adding to JavaScript *aspect-oriented* features that allow straightforward and declarative ways for customization code to modify the targeted application. Compared to most prior aspect-related research, our work has a different motivation and a different target programming environment, both of which lead to novel design and implementation techniques. Our aspect weaving is entirely integrated into a new dynamic JIT compiler, which lets us properly handle advice to first-class functions in the presence of arbitrary aliasing, without resorting to whole-program code transformations. Our prototype demonstrates that an aspect-oriented approach to web-application customization is often more efficient than current idioms while simplifying the entire process.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Design

Keywords Aspects, extensions, JavaScript

* Work done while at Microsoft Research

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

1. Introduction

Web browsers and web applications have become indispensable software for end-users. Yet the software design, development, and deployment model for client-side web code is, for better or worse, very different than the model for traditional desktop applications. Code is delivered in source form, at run-time, from multiple sources, and in a language (JavaScript, abbreviated JS) that is highly dynamic and encourages run-time creation and evaluation of more code.

Consider just the JS code that runs when a user visits a typical "Web 2.0" site. The main page will use JS to provide an interactive experience, probably incorporating popular third-party JS libraries. Ads on the side will contain separately developed scripts. User-installed browser extensions or userscripts include yet more JS to affect browser functionality and how pages are displayed. In fact, some browsers, most notably Firefox, now have large parts of the browsers themselves written in or modified by JS. In short, the code running for a page is a run-time conglomeration of scripts from multiple sources with multiple purposes.

Matters get worse: The entire purpose of some JS (e.g., in a browser extension) is to change the behavior of other JS (e.g., on a popular web page) in ways the affected code never expected. To do so, programmers (ab)use JS features such as the redefinition of top-level functions and the ability to retrieve a function's source code at run-time using its `toString()` method, rewrite it in some manner, and evaluate the result. While it is easy to dismiss such shenanigans as unprincipled hacks unworthy of serious programming-languages study, in this paper we choose instead to accept this reality. Third-party modifications are extremely popular, with tens of thousands of extensions run by millions of users every day. We simply cannot expect web programmers to modify the behavior of web applications only in ways the application writers allow and prepare for.

1.1 Aspects for JavaScript

We propose adding to JS features that make third-party code modifications straightforward, with clear semantics that avoid the pitfalls and awkwardness of current practice. Our approach adapts and extends mechanisms from aspect-oriented

programming to the world of web applications and JS. Our motivation differs from the conventional motivation for aspects: rather than better modularizing code by separating cross-cutting concerns, we use aspects to specify code modifications by third parties concisely and accurately.

Our approach is the first JS just-in-time compiler with support for aspects, taking advantage of the ideas behind recent advances in JIT compilation for JS [13]. Implementing aspects within the JIT offers better performance and completeness than prior approaches. Most aspect systems statically compile aspects into the code they modify. But the web domain has too much code arriving and being generated dynamically for this to make sense: aspect weaving at runtime lets us advise all JS code, regardless of its source. Moreover, aspects can be implemented more efficiently via runtime support than via source-to-source transformation.

JS web code is imperative and event-driven, with first-class functions and prototype-based object hierarchies. These features, combined with the unusual scoping and evaluation-order rules of JS, form a very different substrate on which to define aspects than more well-known efforts for object-oriented or functional languages. We also demonstrate that, despite efforts to the contrary, aspects cannot be properly provided as a JS library; support from the JS implementation is needed.

To evaluate our work, we have taken several popular third-party extensions and rewritten them using our aspects. The resulting code is shorter, simpler, and faster than existing idioms. To justify the features present in our language, we have examined twenty popular Firefox extensions, measuring the fraction of their code involved in aspect-like behavior and how frequently each type of advice occurs. Our language supports nearly every idiom we encountered, and every feature of our design is used significantly often.

While our work required solving some technical issues peculiar to JS, many of our contributions transcend this language. Once one accepts that web applications will be changed and rewritten at run-time in unexpected ways by third-party scripts, one needs linguistic mechanisms to support such change in (relatively) robust ways. Our research focuses on JS because it is the predominant language for today's web applications.

1.2 Outline

Section 2 presents two examples of extensions—one user-script, one browser-level—and the idiomatic hooks they use to install their code. Section 3 introduces the key concepts of aspect-oriented programming and revises the examples to use aspects instead. Section 4 lays out the key properties that an aspect system for JS should support, then presents our aspect language in full. Section 5 describes our implementation. Section 6 evaluates effectiveness and efficiency. Section 7 discusses related work. Section 8 discusses future work. Section 9 concludes.

```
var oldP = unsafeWindow.P;
unsafeWindow.P = function(iframe, data) {
  if (data[0] == "mb")
    data[1] = format(data[1]);
  return oldP.apply(iframe, arguments);
}
```

Figure 1. Central hook used to install a text formatter into Gmail. Note: The common DOM `window` object is available via the `unsafeWindow` alias.

2. Extensible Web-Programming Examples

To understand the nature of code that modifies web applications, we consider two examples. The first modifies a web page running Gmail, Google's webmail client, to reformat the display of emails. The second modifies the Firefox browser to change the behavior of opening a new tab. Section 3 revisits these examples to show how aspects provide cleaner and more robust solutions.

2.1 Reformatting Messages in Gmail

Many email clients detect `/italic/`, `_underlined_` and `*bold*` text using punctuation and format the text appropriately. This functionality is not present in Gmail, so a *user-script*[1] was written to add this feature, replacing, e.g., `/text/` with ⟨**i**⟩text⟨/**i**⟩. A userscript defines code and a set of pages on which it should be run. When a page finishes loading, all relevant userscripts are appended to the page and executed. For instance, this userscript runs for URLs matching `http://mail.google.com/mail/*`. Browsers may support userscript loading directly (as in Chrome) or via an extension (as in Greasemonkey for Firefox).

If all the email data were already present in the page's DOM, it would be simple to identify messages and format them. But like all Ajax applications, Gmail fetches data lazily. All processing of that data begins with a function at global scope and so the userscript hooks into this function to preprocess the data. That crucial hook is shown in Figure 1.

The basic idea is to replace Gmail's `unsafeWindow.P` function with a new function that `formats` the data when a message body (`"mb"`) arrives, and then proceeds to call the original code bound to `P` with the modified data. This technique is known as *wrapping*. The use of JS's `apply` method is necessary so that the implicit `this` parameter of `P` is bound properly. This unintuitive idiom is common, but our approach using aspects makes it largely unnecessary.

2.2 SpeedDial: Customizing New Tabs in Firefox

The Opera and Safari browsers offer a home page showing a grid of the user's most frequently visited sites. This feature is not built into Firefox, but two extensions have been written to add it. We investigated one[2] that takes care to interact with

[1] `http://userscripts.org/scripts/show/8178`

[2] `https://addons.mozilla.org/en-US/firefox/addon/4810`

```
SpeedDial.init = function () {
  ...
   eval("getBrowser().removeTab ="+
     getBrowser().removeTab.toString().replace(
      'this.addTab("about:blank");',
      'if (SpeedDial.loadInLastTab) {this.addTab('
     +'"chrome://speeddial/content/speeddial.xul"'
     +')} else { this.addTab("about:blank")}'
     ));
  ...
  if (SpeedDial.clearURLBarOnLoad) {
    if (!SpeedDial.isFirefox3) {
      ...
    } else {
      var newLocationChange =
         window.URLBarSetURI.toString().replace(
           /aURI.spec == \"about:blank\"/g,
           'aURI.spec == "about:blank" || '
          +'(aURI.spec.indexOf("chrome://'
          +'speeddial/content") == 0)');
      eval('window.URLBarSetURI = '
           + newLocationChange + ';');
    }
  }
}
```

Figure 2. Central hook used to modify the Firefox blank tab.

other features of the browser: when a blank new tab appears, a page of thumbnails of frequently visited sites should appear instead, but the browser's address bar should optionally (depending on user preference) remain blank (as it does for normal blank tabs). Figure 2 shows a simplified version of the main hook that installs this change, though understanding the details of this code is difficult and unnecessary.

As in the Gmail example, the code is redefining a method. However, it is doing so by rewriting the source-code string of the original method—using regular expressions to find and replace text—and then calling `eval` on the resulting string. This technique is error-prone and brittle, and it produces code that is difficult to analyze or maintain. Yet this precise sequence—retrieve the source-code string, manipulate it, call `eval`—is so common that it has a name in web programming: *monkey-patching*. We know of dozens of browser extensions that use it. Our goal is to design an aspect system expressive enough to obviate this idiom.

2.3 Discussion

The two examples above are typical representatives of extensions. They interact with the structure of the page or browser but need to patch the underlying scripts to enable their behaviors. Such patches are more the rule than the exception: this is "how things are done" on the web. There are nearly 40,000 userscripts and over 6,000 Firefox extensions available, with millions of daily users: this development model is successful and growing daily.

Sometimes, patches such as these can drive application revision. Thanks to the popularity of userscripts, Google has revised Gmail's code to provide some APIs for userscripts to use. Similarly, some exceptionally popular extensions for Firefox have been merged into the application's core as built-in features, and new APIs have been introduced to streamline ungainly workarounds used by many extensions. But not all web applications will be so accommodating, nor will users wait for such support. In general, the underlying programs cannot be expected to plan for such diverse extensions, nor can extension authors be expected to know about and plan for all other extensions. Instead, we have the opportunity to provide a more principled framework for writing and maintaining extensions.

Perhaps the most distasteful facet of monkey-patches is their rampant violation of function abstractions through `replace` and `eval`. In an ideal world, such tricks would be unnecessary. However, when mainline programs do not expose a particular value through a convenient abstraction boundary (as with the `aURI.spec` variable in the SpeedDial example in Figure 2), extension authors must resort to abstraction-violating patches to implement their features. Our aim in this paper is not to eliminate such hacks (which would be futile), but rather to recognize and give structure to commonly-used idioms, preserving abstraction boundaries where possible.

Wrapping and monkey-patching also make it difficult to analyze the impact extensions may have on each other or the mainline program: due to aliasing, code paths that were identical may now diverge, and of course all analyses are greatly complicated by the use of `eval`. By contrast, our aspects can eliminate almost all uses of these idioms and so they may make such analyses feasible.

3. Using aspects for extensions

An extension is a self-contained unit of code that needs to insert itself into various places in the underlying application to implement its functionality. An extension is similar to an *aspect*, though "extension-oriented" programming and aspect-oriented programming are very different in motivation. Aspects have traditionally been used for crosscutting concerns such as security monitors or loggers. We propose that the mechanisms of aspects are well-suited to the use case of extending web applications.

We give a brief summary of the terms and concepts of aspect-oriented programming, then introduce our aspect language for JS by rewriting the examples in the previous section. The examples here rely on an informal description of our language constructs, with full and complete definitions following in Section 4.

3.1 Key aspect-oriented concepts

An *aspect* inserts new code into an existing *mainline* program to run at specific times during that program's execution. The new code is called *advice*. Each specific time is called a *joinpoint*, and sets of joinpoints are called *pointcuts*. There are several kinds of pointcuts: for example, one might be "when

function `foo` is called," while another might be "when field `bar` is accessed on object `X`." Aspects also need to specify the type of advice: for instance, *before* or *after* or *around* the call to a function. Sometimes a pointcut might be too broad, describing too many moments during execution. A pointcut can therefore specify a *filter* to restrict the joinpoint: for instance, "all calls to function `foo`, when called by function `bar`." Integrating advice into mainline code is called *weaving*: it incorporates the advice into the mainline such that the advice is triggered at its specified pointcut. Multiple aspects may advise the same pointcut; the weaver uses precedence rules to order the aspects. Advice should rarely call the advised code directly, but rather use a mechanism, usually called `proceed`, to refer to the next installed advice or the underlying mainline, as determined by the weaver.

3.2 Advice surrounding functions

Examining the text-formatting example in Section 2.1, we see it is exactly what *before* advice accomplishes. The new version of `unsafeWindow.P` inserts its preprocessing *before executing* the original function. We can rewrite this example more concisely as

```
at pointcut(callee(unsafeWindow.P))
before(iframe, data){
  if (data[0] == "mb")
    data[1] = format(data[1]);
}
```

An aspect must define a pointcut and advice. Here, our pointcut is `callee(unsafeWindow.P)`, and our advice is the `before { }` block. The name "callee" emphasizes that the advice applies to the code inside `unsafeWindow.P`, rather than inside its caller. Unlike in the userscript, which had to save a reference to and then manually call the original version of `P`, such explicit plumbing is unneeded. Instead, the advice examines and modifies the actual arguments to the function: essentially, the advice is *inlined* into the body of `P`. Traditionally, advice takes a parameter giving reflective access to the arguments of the advised code; our design avoids the indirection and specifies arguments explicitly.

3.3 Advice within functions

The SpeedDial extension in Section 2.2 inserted two hooks into Firefox code, so we define two aspects. Examining the first portion of the code, we see it is executing a particular statement only in certain conditions. There are several ways to translate this intent. Most literally, we might say:

```
at pointcut(
    statement_containing(this.addTab("about:blank"))
    && within(getBrowser().removeTab)) around(...) {
  if (SpeedDial.loadInLastTab)
    this.addTab(
       "chrome://speeddial/content/speeddial.xul");
  else
    proceed;
}
```

This example shows a new type of pointcut, describing the nearest enclosing statement containing a particular subexpression. We surround that statement with advice that in some cases calls an entirely separate function, and in other cases `proceed`s with the original call. Since we do not want to rewrite every statement containing the specified expression, we use a filter to constrain it `within` the lexical scope of the function `getBrowser().removeTab`.

Perhaps the intention was more general: advise *all* calls to `addTab` to open the SpeedDial page instead:

```
at pointcut(callee(this.addTab)) before(url) {
  if (url == "about:blank" &&
      SpeedDial.loadInLastTab)
    url="chrome://speeddial/content/speeddial.xul";
}
```

The extension code, as currently written, gives no indication which of these meanings (or others) were intended.

The second half of the SpeedDial code is actually broken: the mainline code no longer contains the expression `(aURI.spec == "about:blank")`. That means the `replace()` call becomes a no-op, and the monkey patch silently fails. However, the mainline code does set a variable `isBlank`, which likely is the original intent of this extension. We can express the patch as:

```
at pointcut(field(isBlank) &&
            within(unsafeWindow.URLBarSetURI))
wrap {
  set { isBlank = (proceed ||
          (uri.spec.indexOf(
             "chrome://speeddial/content") == 0)) }
}
```

This aspect uses the `field` pointcut to denote accessing variables or fields of objects, again filtered `within` the lexical scope of a particular function. The advice itself `wrap`s the variable, in this case only when it is being `set`. Depending on the situation, we might write `get` advice as well.

Note that the `statement_containing` and `field` advice are fragile: they depend on the syntax of the method being advised. If a subexpression is no longer used or a variable is renamed (which is what broke the SpeedDial code), our advice will fail to apply. We cannot prevent that brittleness—when extensions need to modify the internal logic of functions, there may be no simpler alternative. However, unlike monkey patching, using our aspects will result in a weaving warning indicating that no joinpoints were found.

4. Aspects as a new JS primitive

In Section 4.1, we propose a more reasonable semantics for aspect weaving in JS than is possible with wrapping or monkey-patching. Our approach ensures that functions may be advised regardless of when they are defined, and regardless of how they are referenced. Implementing this relies on two key features: weaving should occur at runtime rather than compile-time, and should apply to closures rather

than variables. These two together give us a third key feature, the ability to disable or re-enable installed advice dynamically. In Section 4.2, we show that neither wrapping, monkey-patching, nor any other idiom within the JS language can provide these guarantees. Section 4.3 describes our full language extensions to JS precisely.

4.1 Key features of an aspect primitive

Dynamic weaving Aspect weaving depends on naming functions as targets for advice. Weaving can happen at compile time or at runtime.[3] We argue that dynamic weaving is the only appropriate method for JS: code frequently defines anonymous functions, or creates closures at runtime. A static weaver would be unable to advise either of these types of functions, leading to an artificial and unintuitive split between advisable and non-advisable functions. A dynamic weaver has no trouble with dynamically created functions: instead of being triggered based on the static name of the function, it is triggered by the *contents of variables at runtime*.

Weaving into closures Dynamic weaving exposes the distinction between variables and their values. Consider the following snippet:

```
var f = function(x) { return x*x; };
var g = f;
Install Advice: before executing f, print(x)
var h = f;
f(4); g(4); h(4);
```

Suppose the last four lines were in separate ⟨**script**/⟩ sources with an unknown loading order (e.g., as subfiles in a library, ads delivered to a page, or scripts in an application) following the first line. The intent is for `f`, `g`, and `h` to be aliases of one another. Reasonably, one would expect that all three calls in the last line *ought to trigger the advice*: clearly calling `f` should, and `g` and `h` are "the same function." The fact that `g` was defined before the advice was installed should be hidden from the user since the loading order is likely unknown. Hence advice should apply to the *underlying closure*, and not to a *variable* bound to the closure. In JS (or any higher-order language), functions may not have unique names, and indeed frequently do have more than one (for instance, by defining a function and subsequently installing it as an event handler in a page). Usually, the developer who uses aspects intends that such aliased functions be advised consistently. Similar conclusions have been reached by others [9].

The ability to advise closures marks our key departure from what is possible within JS: modifying closures cannot be expressed within JS. When this behavior is not desired (i.e., the intended behavior *does* depend on particular aliases), the wrapping idiom is appropriate and still available.

[3] The expert JS programmer will note that "compile-time" is ambiguous, encompassing parsing time and function hoisting time as distinct phases before executing the top-level script statements; moreover this repeats for each ⟨**script**/⟩ on the page. All of these suffer from the same inability to advise anonymous functions, so we consider them all "compile-time" here.

Dynamic disablement Disabling advice arises naturally in the extension setting as extensions provide multiple, mutually-exclusive features that can be selected at runtime by the user. Contrast this form of predicating the execution of advice with the filters mentioned earlier: filters restrict advice to particular lexical or dynamic scopes; disablement restricts advice based on arbitrary runtime decisions.

Disabling advice can be implemented manually by wrapping all advice code in `if`-tests, but this is tedious, and the examples presented earlier only partially implement it: the userscript makes no effort to do so, while the SpeedDial extension is inconsistent, inserting a guard around one piece of advice but not the other. Our aspects are *expressions* in the language that appear as objects, and we equip them with a mutable `disabled` field that selectively disables or re-enables individual advice. This field may be used easily and consistently to "turn off" woven advice.

4.2 Aspects cannot be implemented as a library

Both wrapping and monkey-patching rely on replacing the closure bound to a given variable; nothing else is expressible with variables within JS. Hence they are incorrect in the presence of aliasing. They also suffer additional, distinct problems.

Because wrapping eventually calls the underlying function, it is limited to adding code before and after the function—it cannot modify the internal control flow of the function. Extensions frequently require this ability, making wrapping unsuitable for their needs. (For instance, the SpeedDial code cannot be written as a wrapper, as it must modify code in the middle of the target function.)

Monkey patching fails in two other critical ways. The essential difference between wrapping and monkey-patching is that the former calls the original closure, while the latter discards it. Consequently, the new function does *not* close over the same environment as the original function—the `eval` happens in a different context. Consider this example:

```
function makeAdder(x)
  { return function(y){return x+y;}; }
var addFive = makeAdder(5);
// addFive.toString() == "function(y){return x+y;}"
```

The closure `addFive` makes use of closed-over variables, but by `eval`ing its source code we lose the closure environment: if we were to monkey-patch `addFive`, the new function would use the global value for `x`, if it existed, or fail otherwise. It is impossible, using monkey patching, to determine if a function closed over local variables or not, and so this technique is fatally flawed.

While monkey patching can modify the middle of functions, it is challenging to write precisely the correct replacement operations solely in terms of their textual representation (as opposed to their more structured abstract syntax). The code invariably is obscure, needing fairly long pattern matches to find the right joinpoints, and inserting poorly-

$$
\begin{array}{rrcl}
e \in & \textsc{Expr} & ::= & \ldots \mid a \\
& & \mid & \texttt{retval} \mid \texttt{proceed} \\
a \in & \textsc{AspectExp} & ::= & \texttt{at pointcut}(p)\ ad \\
p \in & \textsc{FilteredPC} & ::= & b\ [\&\&\ f]^* \mid p||p \\
b \in & \textsc{BasePC} & ::= & \texttt{callee}(e) \mid \texttt{field}(e) \\
& & \mid & \texttt{statement\_containing}(e) \\
f \in & \textsc{Filter} & ::= & \texttt{stack}(sd) \mid \texttt{within}(e) \\
sd \in & \textsc{StackDesc} & ::= & e[,\ sd] \mid \texttt{!}e[,\ sd] \\
ad \in & \textsc{Advice} & ::= & \texttt{before}(params)\{s\} \\
& & \mid & \texttt{after}(params)\{s\} \\
& & \mid & \texttt{around}(params)\{s\} \\
& & \mid & \texttt{wrap}\{[\texttt{get}\{e\}][\texttt{set}\{e\}]\} \\
params \in & \textsc{Params} & ::= & [ident[,\ ident]^*]
\end{array}
$$

Figure 3. Aspect syntax for JS

formatted and potentially malformed strings as replacement code.[4] Moreover, if the `replace` fails to match, it returns the original code unchanged, which means that monkey patches *fail silently*, making them devilishly hard to debug.

4.3 Language Semantics

Our language extensions to JS are shown in Figure 3. We explain the language in stages, focusing on the semantics of each construct. Section 5 then shows how we implement these features efficiently, avoiding extra work or code blowup that a naïve implementation of the semantics might incur.

4.3.1 Advising functions: `at pointcut(callee(`e`))`

Operationally, the simplest form of advice applies to closures: in the grammar above, these are `at pointcut(callee(`e`))` ad `{` s `}` expressions, where ad is one of `before`, `after` or `around`. Such advice is *inlined* into the advised closure. When encountered, each pointcut evaluates e to a (reference to) a closure c, which we represent here as $\langle env, \lambda(x_1, \ldots, x_n)\{s\}\rangle$ where env is the environment when the closure was created. (If e does not evaluate to a closure, abort with a runtime error.) We first define weaving one aspect before explaining how full weaving is defined for all the aspects for a closure.

Given a closure $\langle env, \lambda(y_1, \ldots, y_n)\{s_1\}\rangle$ and advice `before(`$x_1, \ldots, x_n$`) {` s_2 `}`, define the new statement $s_2' = s_2[y_1/x_1, \ldots, y_n/x_n]$. *Mutate* the closure, replacing it with $\langle env, \lambda(y_1, \ldots, y_n)\{s_2'; s_1\}\rangle$. (`After` advice is analogous; `around` advice requires slightly more effort.) Notice that the updated closure uses the original environment, which avoids the rebinding-of-variables problem associated with monkey patching. Moreover, because we update the closure in place, we resolve the aliasing problem: all references to that closure now contain the advice. This neatly includes recursive calls to e: if env contained an entry pointing to this closure, it will subsequently see the now-mutated version.

[4] The expert JS programmer will note that poor formatting is not merely aesthetically problematic: problems may arise due to the interaction of patch code containing newlines and the rules for semicolon insertion.

`Proceed` ***and*** `retval` Generalizing `before` and `after`, `around` advice surrounds the mainline code. To "call" the mainline code, `around` advice uses the `proceed` keyword. Semantically, we again inline: given `around` advice s_2 and mainline s_1 as above, we define $s_2' = s_2[y_1/x_1, \ldots, y_n/x_n]$. The woven code is then $s_2'[s_1/\texttt{proceed}]$.

It is common for `after` or `around` advice to refer to the return value of the mainline code. In the body of such advice, we bind the return value to the name `retval`, which then can be read or modified as needed. (Outside advice, `retval` is not a reserved keyword.) Any `return` statements become jumps to subsequent advice; the "last" one sets the final return value, and the caller resumes control only after all advice have run.

Weaving order Overall, weaving for a particular closure c is defined in terms of *all* pointcuts for which the expression e evaluates to c. We take the *original, unadvised* closure $\langle env, \lambda(y_1, \ldots, y_n)\{s\}\rangle$ and ordered lists for before $(b_1, ..., b_i)$, after $(a_1, ..., a_j)$, and around $(r_1, ..., r_k)$ advice. (For simplicity, assume all aspects use the same parameter names as the mainline function; in general we perform the same substitution on parameter names as above.) Each list is ordered by when the pointcut was encountered during program execution. We then apply each advice one at a time, starting with b_i in order, then a_i in reverse order, and then surrounding them by r_i in order, similar to AspectJ's weaving order [14]. The woven result is then

$$
\begin{array}{l}
r_1\,[\cdots[r_k\,[\\
\quad \{b_1; \cdots; b_i; s; a_j; \cdots; a_1\}/\texttt{proceed}]\cdots]\,/\texttt{proceed}]
\end{array}
$$

This ordering semantics means that weaving cannot be performed eagerly when dynamically evaluating each aspect expression. Instead, our JIT (see Section 5.3) stores the aspects for each closure with the closure and weaves while JITing when the closure is invoked. When the collection of aspects for a closure changes, weaving/JITing is redone.

The translation above deliberately does not deal explicitly with exceptional control flow. If any code (mainline or advice) throws an exception, all subsequent code is skipped. However, because our advice is truly inlined into the function, `around` advice may surround calls to `proceed` with a `try-catch` statement, and control flow will work properly. (In AspectJ terms, all `after` advice is really `after returning`, and we do not support `after throwing` advice.) Similarly, if `before` advice `returns` early, no mainline code or subsequent advice runs, unless `around` advice uses a `try-finally` statement. This modifying of control flow by advice is surprisingly common behavior by real-world extensions (c.f. Figure 7 and the usages of `statement_containing` advice).

Runtime representation of aspects and dynamic disablement Aspects are expressions in our language, and when executed they evaluate to native objects (much like built-in objects, e.g., `Math` or `Array`): `aObj = at pointcut...{...};`. These objects have a mutable `disabled` field that "turns off" the advice. If the program sets `aObj.disabled = true`, then effectively `aObj` is removed from the installed-advice list in the closure, and the advice is rewoven. When the program resets `aObj.disabled = false`, `aObj` is restored to its original position in that list. Our implementation does not actually reweave or re-JIT, but still provides these semantics.

Named parameters Note that in the example above, and in the abstract syntax for advice, we give names to the parameters of the function. Advice parameters are resolved by position, and do not have to match the parameter names defined by the mainline function. This has the benefit of letting aspects name parameters for native methods (such as `Math.sin`), which do not have explicit JS names for their parameters. By permitting the aspect author to name parameters, we enable a more natural coding style for advice.[5] Traditional aspect style would use a reflective parameters array; JS already includes this via the `arguments` array-like object. However, for technical reasons, *any* usage of the reflective `arguments` object prevents the JS engine from applying certain useful optimizations. Referencing the `arguments` object requires an additional object creation and initialization per function call, and requires an extra indirection when referencing all parameters in that function.

4.3.2 Stack Filters

Extensions frequently need to advise utility or library functions so as to change behaviors of the program. But perhaps not *every* call to those function needs advice; only ones with a particular call stack should be advised. To achieve this, we let developers specify *filters* to constrain which joinpoints match a pointcut: `callee(`e`) && stack(`s`)`. In our language, the `stack` filter generalizes AspectJ's `cflow` pointcut. It permits specifying multiple stack frames that must, or must not, be present when the advice is triggered.

For example, a filter `stack(a, !b, c, d)` states that functions `a`, `c` and `d` must be on the stack in that order, and that `b` must not be on the stack between `a` and `c`. Our design is greedy—`a` matches the deepest (i.e., oldest) `a` on the stack—and permits arbitrary intervening stack frames between specified frames. Thus, the pattern above will match the stack `main, a, a, c, b, d`, but will not match the stack `main, a, b, a, c, d`. This semantics fits well with extensions' requirements: supporting unspecified intervening frames lets stack patterns continue to match even if other extensions insert themselves into the call stack, while the greediness of negative assertions permits defensively avoiding specific other extensions.

To define the semantics for stack filters more precisely, let $S = e_1 :: \cdots :: e_m :: \top :: []$ be a stack, where e_1 is the deepest stack frame and $\top$ is added after e_m at the young end of the stack. Let `F = f`$_1$`::`$\cdots$`::f`$_n$`::`$\top$`::[]` be a stack filter (again $\top$ is implicitly added). Inductively, `F` matches S in the following cases:

1. `F` $= []$
2. `F = !f`$_1$`::`$\cdots$`::!f`$_n$`::h::F'`, $S = e_1 :: \cdots :: e_m :: g :: S'$, $e_i \neq$ `f`$_j$, $e_i \neq$ `h`, $g =$ `h` and `F'` matches S'
3. `F = f::F'`, $S = e :: S'$, $e \neq$ `f` and `F` matches S'
4. `F = f::F'`, $S = e :: S'$, $e =$ `f` and `F'` matches S'

Note that negative assertions are not quite symmetric with positive assertions: they accumulate until a positive assertion discharges them. Case 2 therefore has to skip over (zero or more) stack frames e_i until finding the first one (g) that matches the next positive assertion (`h`), and check that none of e_i match any of the accumulated `f`$_j$.

Weaving with filters Naturally we have to change our weaving definitions to accommodate filters. For a set of stack filters $f_1, \ldots, f_n$ guarding some aspect, and a current stack S, we must check that *all* filters match the stack S. For `before` advice b or `after` advice a, we define

$$b' = \texttt{if}(match(S, f_1) \,\&\&\, \cdots \,\&\&\, match(S, f_n))\ \{b\}$$
$$a' = \texttt{if}(match(S, f_1) \,\&\&\, \cdots \,\&\&\, match(S, f_n))\ \{a\}$$

For `around` advice r, we must be sure to call subsequent advice:

$$r' = \texttt{if}(match(S, f_1) \,\&\&\, \cdots \,\&\&\, match(S, f_n))\ \{r\}\ \texttt{else}\{\texttt{proceed}\}$$

Weaving multiple pieces of advice proceeds as before.

4.3.3 Advising multiple functions simultaneously

The construction `at pointcut(`$p_1||p_2$`)` ad is roughly syntactic sugar for `at pointcut(`p_1`)` ad`; at pointcut(`p_2`)` ad, and is evaluated in the obvious manner. This construct may nest arbitrarily: $p_1||p_2||\cdots||p_n$. The sole distinction between the parallel and desugared forms is that the former produces only *one* aspect object a while the latter produces several. Thus the parallel construction lets programs enable or disable the advice on all targets simultaneously. Each base pointcut p_i must be the same type of pointcut—all `callee`, all `field`, or all `statement_containing`.

4.3.4 Advising within function bodies

Our survey of popular extensions' code emphasized that not all desired program modifications fall neatly at function boundaries. We therefore support two additional pointcuts in our language, advising how variables are accessed within functions, and advising statements within function bodies.

[5] This is similar to an idiomatic usage of AspectJ's `args` pointcut, though we do not support `args` as a pointcut for JS: all functions take arguments of dynamic types and arities, which defeats the intention of `args`.

Advising variables As in AspectJ, it is useful to advise getting and setting variables and fields. Extension code often modifies local variables within a function, to influence its behavior. We therefore define the `field` pointcut and its corresponding `wrap` advice, which specifies a getter or setter (or both) to be used instead of the designated variable or field. Since variable names are commonly reused within a program, it is undesirable to advise all accesses to that name everywhere in the program. Therefore any aspects using the `field` pointcut also specify a `within` filter which (like the `callee` pointcut itself) accepts an expression that resolves to a closure at runtime; the advice is applied only to that closure.

The advice code must be an expression, just like the code it is replacing.[6] To accommodate weaving multiple advice onto the same variable or field, we use `proceed` to denote the next advice expression or the underlying expression as appropriate. Any expression of the form `a.b.c.d` can be used with the `field` pointcut, rather than just variables. We do not support array indexing (e.g., `field(a[`e`].f)`), as the replacement advice may evaluate e at different (or potentially multiple) times, which may cause unintuitive side effects.

For example, the following code ensures that the `prefs` object never appears null during the `config` function:

```
at pointcut(field(prefs) && within(config))
wrap { get { prefs != null ? prefs : getPrefs() } }
```

while the following advice ensures that a variable containing a maximum value can never decrease:

```
at pointcut(field(maxVal) && within(computeStats))
wrap { set { maxVal = Math.max(maxVal, proceed) } }
```

Note that in JS, assignments are expressions, so the code above assigns the correct value to `maxVal`, and then returns it to any surrounding code.

The semantics of `field` advice are straightforward: given an aspect `at pointcut(field(x.y) && within(`f`)){ get { `g` } set { `s` } }`, evaluate f to some closure $\langle env, \lambda(y_1, \ldots, y_n)\{b\}\rangle$. Replace all assignments `x.y` $= e$ in b with $s[e/$`proceed`$]$. Replace all other occurrences of `x.y` with $g[$`x.y`$/$`proceed`$]$. The extension to multiple advice for the same `x.y` is analogous to `callee` advice. Note that the joinpoints are specified only by the mainline code b—later advice does not apply to code introduced by earlier advice.

Advising statements The remaining type of advice is a new way to insert code into the body of a function, inserting new statements before, after, or around existing statements within function code. Other than systems that expose explicit labels for joinpoints, we are unaware of any aspect system that allows this flexibility. The challenge is isolating a sufficiently expressive pointcut to identify individual statements. We define `statement_containing(`e`)` as the smallest statement containing expression e, as matched by abstract syntax. This has some subtleties: in the statement

```
while (x + 1 > 0) { if (x < 5 && x > 0) { x--; } }
```

the pointcut `statement_containing(x)` matches each of the `while` loop, the `if` statement, and the decrement statement, because each one contains an instance of `x` not contained in any smaller statement. Conversely, it matches the `if` statement only once, despite the repeated usages of `x`, so that advice is not woven multiple times at the same point.

Once the pointcut selects statements, we can easily apply before, after, or around advice to them, replacing statements with statements. The `proceed` keyword executes the original statement or subsequent advice. This pointcut is particularly useful with extensions where the original code handled a certain set of behaviors and the extension adds a new, unexpected one. Suppose a mainline function assumes its argument is a member of some enumeration `{A,B,C,D}`, and throws an error otherwise. If an extension adds a new member `E` to that enumeration, it must also add code that handles `E` and avoids raising the error. Such code can be inserted before each `statement_containing` that argument.

4.3.5 Discussion

This section considers additional aspect constructs one might contemplate adding and discusses how our aspects language may interact with new features added in ECMAScript 5 [10].

All our pointcuts are constructed to select exactly one target via `callee` or `within`; the parallel $p||p$ construction permits specifying multiple pointcuts per advice. However, many aspect systems admit more free-ranging pointcuts, either with wildcards, catchalls, or arbitrary pointcut-designator functions, that select an arbitrary number of targets. In the web-extension context, this flexibility is unused, either because it is truly unneeded or because no extant JS idiom can express it. In the twenty examples we examined in depth (see Section 6.2), each wrapper was uniquely applied to a single function, as was nearly every monkey-patch. The few exceptions were either applied a fixed number of times—essentially $p||p$—or applied, one at a time, to an ad-hoc array of functions: it is unclear from the extensions' code what broader pointcut the developer might have intended that would select precisely those targets and no others.

Missing from our set of pointcuts is `caller`, which inlines code at call sites rather than within the callee. There are technical and pragmatic reasons for this. The technical reason stems from the dynamic nature of JS: We cannot know until runtime when a particular function is about to be called—at which point, the caller has already been compiled, which prevents us from inlining the advice at the call site. To overcome this, we would have to insert conditional tests at every function call site, which would introduce significant overhead to the common, unadvised case. Pragmatically, we have not seen advice-like idioms that try to emulate a `caller` pointcut. In the code we have examined, those usages that

[6] If necessary, the programmer may wrap the advice in the `(function(){...})()` idiom to use statements and `return` instead.

pick out particular call sites (e.g., advice that uses a `stack` filter) modify the caller's code only in ways that are better expressed using our `field` pointcut.

A larger survey of extensions might indicate further pointcuts and filters. Extensions may want to select only `if` statements, or only the n^{th} occurrence of some statement, or use wildcards to select any `statement_containing(_ > 0)`). Supporting these poses no inherent challenges.

The ECMAScript 5 proposal includes ways to define getter and setter functions for fields. Once these abilities are implemented in JS engines, advising a field's accessor functions using `callee` advice might seem to subsume `field` advice. However, accessor functions are neither required nor implicitly defined; retaining the explicit `field` pointcut lets users advise fields whether or not they have accessors.

JS is primarily hosted within the web setting, which may also inspire new constructs. With appropriate implementation hooks, our approach is already capable of advising DOM methods (e.g., `Element.appendChild`). New pointcuts would be required to advise event dispatch or other moments in the lifecycle of a webpage [11]. New filters might be useful to restrict advice to some subtree of the current page.

5. Implementation of advice weaving

We implemented our extensions in the Microsoft Research JScript compiler[4]. This compiler is a JIT that lazily compiles target function bodies at call time, specialized for the dynamic types of the arguments at that call-site. Our compilation strategy leverages this behavior: when calling an advised function, we must JIT the function body anyway, so we weave the advice into the body just before JITing, which then compiles the woven result. Additionally, since weaving happens entirely at runtime, our weaver can produce special syntactic forms that are not available to concrete syntax: these special forms are crucial to our efficiency. Section 5.1 explains how our JIT compiles regular, unadvised code. Section 5.2 explains the changes needed to compile aspect expressions. Section 5.3 explains the weaving process itself.

5.1 Compiling unadvised code

When we encounter a JS function for the first time, we need to generate intermediate code that we can then execute. Because our JIT is lazy, we instead create a *code generator*, an object that encapsulates all the information needed (e.g., the current environment) to eventually compile the source JS. The runtime representation of a closure contains a pointer to its associated code generator.

When we encounter a function call expression $f(e)$, we:

1. Evaluate f to a closure c_f and let $cg_f := c_f.\textit{codeGen}$.
2. Ask cg_f to compile f, specialized to e's runtime type.
3. Jump to and execute the compiled code.

The specialization in step 2 is straightforward to do at runtime, and is key to efficiency. Since JITing is expensive, code generators memoize the compiled, specialized function bodies in a table $\textit{cache} : \langle \textit{list of argument types} \rangle \rightarrow \textit{CompiledCode}$. Further, code generators are shared among all closures with the same source (e.g., as with higher-order functions): all closures sharing a code generator run identical code. The effect is to compile identical functions as few times as possible.

5.2 Compiling aspect expressions

Aspects rewrite their advised closures, which means we must erase the closures' code generators' memoized compiled code. Normal JIT mechanisms will then recompile the closure when needed, at which point we can weave the advice. To achieve this invalidation, we add a timestamp field to both closures and code generators: if a closure's timestamp is newer than its code generator's, then the memo table is out of date. To make code generators aware of aspects, we add a list of installed aspect expressions to each closure, which can then be used during weaving. In short, when a code generator compiles an aspect expression a, it generates code that:

4. Evaluates the pointcut p to a closure c_p.
5. Adds a pointer to a into c_p's list of aspects.
6. Updates c_p's timestamp.

When we encounter a function call $f(e)$, we change step 2 above, since it now needs to account for the installed advice. First, we must effectively change the memotable to be of type $\langle \textit{list of aspects} \rangle \rightarrow \langle \textit{list of argument types} \rangle \rightarrow \textit{CompiledCode}$. Second, we must check timestamps: if the code generator is out of date, we clear the cache and update the timestamp. But clearing this entire cache is too coarse: other closures may share the code generator, but may not share the same advice. We only need to clear the part of the cache keyed by the current installed aspects. To do so, we leave the memotable as it was, and instead make a new code generator for "the closure plus current advice". We want to maintain sharing when multiple closures with identical code generators are advised by the same aspect. We introduce a second table $\textit{aspectCache} : \textit{aspect} \rightarrow \textit{CodeGenerator}$. To use this table, we continue after step 6:

7. Let $cg_p := c_p.\textit{codeGen}$.
8. If $cg_p.\textit{aspectCache}[a] \neq \textit{null}$, then set $c_p.\textit{codeGen} := cg_p.\textit{aspectCache}[a]$.
9. Otherwise, set $c_p.\textit{codeGen} := \textit{new CodeGenerator}$ and $cg_p.\textit{aspectCache}[a] := c_p.\textit{codeGen}$.

Suppose we have three closures c_x, c_y and c_z sharing code generator cg_1. When c_x is advised by aspect a, we create a new code generator cg_2 for it to use; cg_1 (with its cache) is still valid for c_y and c_z. If c_y is later advised by a, it finds $cg_2 = cg_1.\textit{aspectCache}[a]$, and so reuses cg_2 and its cache. Again, c_z continues using cg_1 and its cache.

This sequence of operations is minimally invasive on the critical path of function dispatch: in programs with no aspects, we add only a single branch (comparing timestamps). More-

over, it is maximally sharing, creating the fewest number of distinct code generators. The worst case behavior involves repeatedly advising a function and calling it once, which completely defeats any caching behavior. Even in this unrealistic, pathological example, our compilation overhead is lower than that of other techniques (see Section 6.1).

5.3 Weaving advice

We now describe the actual weaving process. Our weaving mechanism is essentially a function of type `ASTNode` $\times$ `AspectList` $\rightarrow$ `ASTNode`, though we make use of `ASTNode` types that cannot be generated via concrete syntax. These synthetic nodes let us temporarily alias function parameters to match advice parameter names or manipulate the return address of the function without strange syntactic contortions.

5.3.1 Weaving `callee` advice

Consider a function `f` with body `B`, and lists $\vec{b}$, $\vec{a}$ and $\vec{r}$ (of lengths l, m, and n) of installed before, after, and around advice. Assume for now that each advice came from a `callee(f)` pointcut with no filters. Figure 4 shows a simplified resulting woven body `B'`. Before and around advice can be concatenated with the body (BA). Around advice is more complicated, as it may call `proceed` multiple times; we implement around advice by inlining the next around advice (R_{i+1}) into the current one. Labels L and L_{a_i} mark potential targets for `return` statements. The net result is the outermost R_1, along with some bookkeeping described below.

Named parameters To implement the renaming of function parameters, we must ensure that any names introduced as parameters in one piece of advice must be scoped only to that advice. Rather than rewrite advice s explicitly to substitute names, we temporarily introduce the named parameters as *aliases* for the parameters of the function, scoped only to the body of the advice. We surround the advice by a pair of directives `RenameParams` and `UnRenameParams`. These have no runtime effect (they do not appear in the compiled code), but temporarily change how the code generator compiles those identifiers: instead of local variables, they resolve to the appropriately positioned arguments on the stack. Assignments to these parameters via these aliases are visible to subsequent advice and to the mainline code. The current implementation requires that the arities of the function and advice must match; this is not a fundamental requirement and can be relaxed later if desirable.

Retval and exceptions Because we inline our advice into the callee, we have to pay careful attention to return values and targets. The calling conventions dictate where a function's return value is located; the AST node `retval` (used by `around` or `after` advice) provides a mutable way to describe that location. Advice can therefore change the return value by either assigning to `retval` or simply `return`ing a new value. The calling conventions also dictate that `return` statements branch to the function epilogue before resuming the caller.

```
Let BA =
  For each Before advice b_i (1 ≤ i ≤ l) in install-order
          RenameParams(b_i.params)
          b_i.body
          UnRenameParams(b_i.params)
  InstallReturnLabel(L_{a_m})
  B
  For each After advice a_i (1 ≤ i ≤ m) in reverse order
  L_{a_i}:  InstallReturnLabel(L_{a_{i-1}})
          RenameParams(a_i.params)
          a_i.body
          UnRenameParams(a_i.params)
Let R_{n+1} = BA
For each Around advice r_i (1 ≤ i ≤ n) in install-order
Let R_i =
  RenameParams(r_i.params)
  r_i.body[Let L be a fresh label in
          InstallReturnLabel(L)
          UnRenameParams(r_i.params)
          R_{i+1}
          RenameParams(r_i.params)
          L
    /proceed]
  UnRenameParams(r_i.params)
Let B' =
  R_1
  UninstallReturnLabels()
  return retval
```

Figure 4. Weaving of `callee` aspects

To recapture control flow for `after` and `around` advice, we must change which label the compiler thinks identifies the epilogue: the `InstallReturnLabel` directive does precisely this, changing the return label from the epilogue to the next applicable advice. The original return label is reinstated by the `UninstallReturnLabels` directive.

`Proceed` It is possible that multiple `around` aspects that each call `proceed` multiple times could lead to exponential code blowup. In practice this is unlikely; none of the extensions we examined made use of such constructions. To avoid the blowup, we can compile `proceed` to a pair of jumps, similar to our construction for `retval`.

5.3.2 Weaving `stack` filters

Implementing our design for `stack` filters is particularly efficient: though it appears we need to walk the stack when the advised function is called, we can instead achieve the same effect using only $O(1)$ work per function named by the filter.[7] A `stack` filter is a simple state machine whose state must be updated as each relevant function runs. To evaluate the filter in the aspect `at pointcut(callee(`e`) && stack(`$f_1, \ldots, f_m$`)) before {`s`}`, we need to add code to the entries and exits of closures f_i to update the state of the stack filter. Specifically, let st be a runtime representation of a stack filter. Then for each $1 \leq i \leq m$, evaluate `at pointcut(callee(`f_i`)) around {` $enter(st, f_i)$`; try { proceed; } finally {` $exit(st, f_i)$ `} }`. Functions

[7] We are not the first to recognize that stack inspection can be achieved more efficiently; Wallach et al. [22] use a similar technique for Java.

$enter$ and $exit$ update the state of st to reflect how far the stack pattern matches the current call stack, based solely on the current stack frame f_i; using `finally` ensures that $exit$ will run regardless of the mainline control flow.

Consider a filter `stack(a, !b, c)`. When `a` is called, the filter's state advances from "Start" to "Expecting `c`". If `c` is called (i.e., the stack is `a::···::c`), the state advances to "Success" and the enabled-filter counter is incremented. As `c` exits, the counter is decremented and the state reverts to "Expecting `c`". However, if `b` is called before `c` (i.e., `a::···::b::···::c`), the filter state is "Fail" until `b` exits.

The remaining details elided from Figure 4 support `stack` filters. Since aspects can have multiple `stack` filters, we equip them with a counter of currently-enabled filters, maintained by the stack advice above. We surround advice with an enablement check (shown here for `after` advice):

```
L_{a_i}: if (isEnabled(a_i)) {
           InstallReturnLabel(L_{a_{i-1}})
           RenameParams(a_i.params)
           a_i.body
           UnRenameParams(a_i.params)
         }
```

The `isEnabled` helper checks that the current count equals the installed count. If the test passes, the advice runs as before; otherwise, control falls through to the next installed advice. (Around advice cannot simply fall through; we instead generate an `else { proceed }` branch.)

Dynamic disablement To check whether the program has disabled an aspect at runtime (via `aObj.disabled = true`), we add that check to the $isEnabled$ function. The weaving remains unchanged.

Avoiding redundant overhead If an aspect has no `stack` filters, or if the aspect expression is used as a statement (and therefore the aspect object is ignored), we can simplify or eliminate the $isEnabled$ tests, leading to more efficient woven code (see Section 6.1).

5.3.3 Weaving `wrap` and `statement_containing`

Compiling `wrap` or `statement_containing` advice is done by preprocessing the body before applying `callee` advice. During preprocessing, field advice is rewritten before statement advice. Recall that neither pointcut designates a function in which the rewriting should occur: this is specified by the mandatory `within` filter which, like the `callee` pointcut, evaluates its argument at runtime to a closure, and installs the advice onto it. Such advice is therefore somewhat of a hybrid, as it is a syntactic (static) transformation on a runtime-specified (dynamic) closure.

The current implementation has the small limitation that it is impossible to advise global variables while in the global scope. This is a minor restriction on expressiveness, as there is relatively little code at global scope. In practice, this has not been a stumbling block. (Resolving this would change how we compile global code and penalize the performance of unadvised code; we do not feel there is enough interesting code in global scope to warrant this change.)

6. Evaluation

The MSR JScript compiler is written entirely in C# and consists of a JS front-end and JIT code generators targeting either a specialized bytecode or the .NET Common Intermediate Language (MSIL); the MSIL in turn is compiled by the CLR JIT. Our extensions were easily confined to the front end and the code generators; the CLR JIT was unchanged. Because the backends target .NET, and because the runtime environments are implemented in .NET, the CLR JIT can easily optimize JS code together with the runtime.

We evaluate our framework on performance (using the MSIL backend) and on expressiveness.

6.1 Performance

Recent work has seen enormous improvements in the performance of JS engines [13], so it is important that new constructs not undo this progress. Our compiler is work in progress, so absolute performance numbers are preliminary. Instead, we measure the relative performance of 1) an unwoven base program for reference, 2) simple advice defined using our aspects, along with equivalent monkeypatched, wrapped, and manually-woven versions, and 3) advice with a `stack` filter defined by aspects, and monkeypatched, wrapped and manually-woven versions. We choose a trivial baseline function and minimal advice, to pessimistically maximize the ratio of weaving overhead to useful runtime work: the aspect version of our test program is shown in Figure 5. The seemingly-extraneous function surrounding the advice is included in the manual and monkey-patched versions too, to force them to execute as many closures while weaving in advice as the equivalent wrapper idiom, eliminating one (large) source of runtime differences. This hurts the non-wrapper versions equally, slowing them to the weaving speed of wrapping, without impacting the performance of the advised code. In practice, real code would not be written this way.

To account for inter-run and system variability, we report the minimum time achieved for each technique: whereas system noise can slow down a test run, the minimum time represents the fastest time actually achieved by any approach. In our setting, measuring performance must distinguish two levels of JITing/caching: .Net JITing our compiler and our compiler JITing the JS program. Once the .Net cache warmed up (the first 10-20, out of 200 runs), variability was insignificant: the averages and minima were nearly equal. Every test iteration started from a cold cache for our compiler. Our tests were run on a 2.8GHz Windows XP machine with 4GB of memory running .NET 3.5 SP1. Despite caching and JITing, no test had a working set greater than 35MB of memory.

Results are shown in Figure 6, presented as ratios of the total runtimes of each test versus that of the unadvised code.

```
square = function(v) { return v*v; };
callSquare = function(v) {
  var ret = square(v);
  return ret;
};

(function() {
  at pointcut(call(square) && stack(callSquare) )
  before(v) { v++; };
})();

for (test = 0, incr = 1; test < N; test++, incr++)
  if (callSquare(test) != incr*incr)
    print("Test failed for test ", test);
```

Figure 5. Test microbenchmarks, without and with `stack` filters (boxed), written using advice. N is an integer literal that varies across the x-axis in Figure 6.

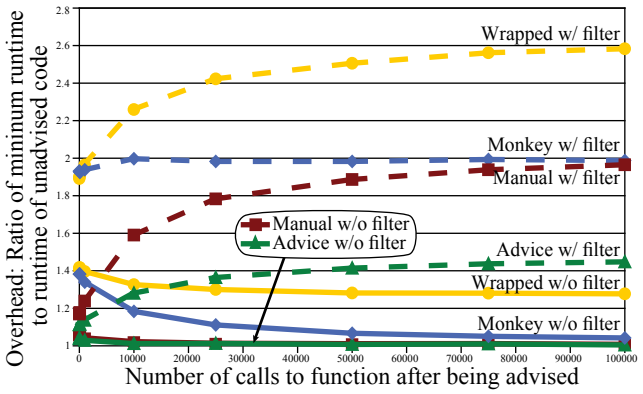

Figure 6. Overhead comparison for test in Figure 5, with (dashed) and without (solid) stack filters. Lower is better.

Solid lines show the unfiltered test; dashed lines show the filtered one. The x-axis counts how often the test function is called after being advised (the constant N): calling it only once effectively measures the overhead of the weaving process; calling it many times asymptotically approaches the overhead of the woven code. For some techniques, the relative cost of weaving may be cheaper than that of the woven advice, leading to some curves *rising* for larger N.

In short, our weaving mechanism runs *as fast* as the unfiltered manually-woven version, and *5–27% faster* than the `stack`-filtered manually-woven version. Contrast this with wrapping (31–61% slower than the filtered manual version), or with monkey-patching (1–63% slower). Wrapping's performance is badly hampered by the extra function calls within the advice (recall we already accounted for the extra function calls within the weaving process itself). Monkey patching generates code identical to the manual version, but pays a large initial cost for `eval`. Our performance parity in the non-filtered case comes from simple optimizations that eliminate vacuous filter-enablement checks. Our performance gains in the filtered case come from implementing the stack filter inside the runtime, rather than with state variables in JS itself. Runtime weaving as implemented here provides higher performance *and* stronger semantics than existing idioms—a win-win situation.

6.2 Expressiveness

We designed our aspect language to support extensions, but we deliberately did not examine a large selection of extensions in advance, to prevent over-fitting our language to awkward coding idioms or legacy code. Instead, after designing the language, we thoroughly examined the scripts of twenty other popular Firefox extensions. Note that because our engine is not embedded in Firefox, we have not yet implemented an aspect-enabled Firefox, but rather we show that extensions could easily be rephrased to use aspects instead. Our twenty extensions were drawn from a snapshot of the top 50 add-ons in each category from Mozilla's `http://addons.mozilla.org`, as of October 2008; the versions of these twenty extensions tested here all support Firefox 3.0. The snapshot contains 350 distinct extensions, of which roughly 10% use monkey-patching; our twenty extensions were drawn from these. As such our sample may conservatively undercount the prevalence of wrapping in extension code.

Our results are summarized in Figure 7. The first three columns measure the amount of JS code in each extension, and the amount of code directly attributable to either monkey-patches or wrapping. We count as a monkeypatch any lines of code that call `eval` and `replace` on code, including the string arguments as they are the contents of the patch. It is more difficult to precisely count the lines of code needed in wrapping, as sometimes a function is simply wholly replaced, rather than precisely fitting the wrapping idiom. In such cases, we choose to undercount and include only the single line that binds the new function to the old name. In total, these twenty extensions contain 98,700 lines of JS code (LOC), of which 2,713 LOC (2.8%) is a monkey-patch or wrapper.

The extensions range in size from 300–10,000 LOC, and subjectively range widely in their effects on the browser. The two are not correlated: Tree Style Tab requires far more lines of monkey-patching code ($894/6786 = 13\%$) than any other extension, even though it is less than half the size of TabMixPlus and even though subjectively it modifies the browser less than SplitBrowser. Conversely, All-in-one Sidebar is nearly the same size overall, but has 5% the amount of patching code, while NoScript is twice as large and profoundly impacts the browsing experience, yet needs merely 11 lines of patches. Regardless of size, patches are key to extensions' behavior, and so simplifying them is helpful.

The remaining columns of the table count and categorize each hook (both monkeypatch and wrapping) as one or more of our advice types, or as "other" if we could not express it using our advice language. We were carefully literal-minded in these classifications: if a patch was not precisely expressible as an aspect, even if a semantically equivalent patch could be so expressed, we counted this patch as a failure. For example, Tree Style Tab includes seven patches that insert

Name	JS LOC	Monkeypatch LOC	Wrapping LOC	Function	Field	Stack	Stmtcont	Other
Fission 1.0	367	5		5		2		
TabRenamizer 0.8.11	536	3					3	
Compact Menu 2-2.2.0	586	7		1	1	1		1
Multirow Bookmarks Toolbar 2.9	587	53		2		2	2	
Redirect Remover 2.5.5	926	20		2	2	2	1	
Multiple Tab Handler 0.3.2008101801	2048	83	5	10	2	3	1	1
Img Like Opera 0.6.17	2287	45		4	1	1	6	1
FireGestures 1.1.5.1	2455	6	4	2	1			
Gladder 2.0.3.1	2558	49		4	1			
All-in-one Gestures 0.19.1	4056	4					1	
Split Browser 0.5.2008101801	4519	333	269	71	48	20	14	
Session Manager 0.6.2.4	4697	6	3	3	2		1	
TabKit 0.4.3	5232	63	16	12				
SpeedDial 0.7.2.5	5641	20	22	3	2	1		
Tree Style Tab 0.7.2008101801	6786	894	5	45	32	13	26	7
All-in-one Sidebar 0.7.6	7079	38	23	22	9	3		
Gmarks 0.9.9	7700	3	1	2		1		
TorButton 1.2.0	10560	2	139	139				
NoScript 1.8.3.3	10809	10	1	4		3		
TabMixPlus 0.3.7.3	14278	551	30	90	21	51	23	5

Figure 7. Comparing 20 Firefox extensions, showing code size, patch size, and counts of how many patches by advice types.

a new statement after the opening brace of an `if`-statement. None of our existing advice forms support this. However, in these cases the guards of the `if`s are pure, and so using `before statement_containing` advice that repeated the `if`-test would have the same semantic effect. Despite being so stringent, our language as presented can express 621 out of 636 (97.6%) observed hooks. Of the remaining 15: seven add a statement to the beginning of an `if` block; four change the condition of an `if` test; one changes a statement in the middle of an `if` body; one inserts a statement at the end of one case of a `switch` statement; one changes a `while` loop into a `for` loop; and one is unknown and appears broken.

Evident from the table is that all forms of advice are actually used: function advice is by far the most common, but statement advice accounts for over 10% of the advice. Of the 103 `stack` filters, while most filters were only one function deep, several instances used multiple patches to manually implement two- or three-function-deep filters; these patches could all be subsumed into a single `stack` filter.

7. Related Work

7.1 Aspects for object-oriented languages

AspectJ [18, 19] is probably the most well-known aspect-oriented language. AspectJ was designed to support both static and load-time weaving. In Java all methods have statically known names. Compiling AspectJ efficiently poses several challenges, particularly for `cflow` [3]. However, since Java does not support first-class closures, aliasing issues simply do not arise.

AspectJ employs complicated heuristics to define how pointcuts match in the presence of inheritance, overridden methods, and interfaces. The rules are designed so that advising a method on some superclass will trigger that advice on all subclasses, regardless of whether they override the method. But JS is a prototype-based object-oriented language, making heuristics designed for class hierarchies unnatural. Our design of advising closures reflects that distinction: by triggering advice through all aliases to a closure, we ensure that all objects of a given prototype share advice applied to the prototype. However, if a new object overrides a method from its prototype, it is a different kind of object, and advice does not implicitly attach to it.

Some of the compelling power of aspect-oriented programming derives from its freedom to tamper with almost any part of the code. Aldrich [2] proposed an explicit "open module" approach to curtailing that freedom. Many of these ideas may be directly applicable to aspects in JS: for instance, functions computing trusted values (encryption, cookies, passwords, etc.) might be sealed from advice. Indeed, the object-hardening proposals for ECMAScript 5 [10] effectively permit sealing functions from wrapping or monkey patching; our aspect system would need to support or integrate the same ability. Currently, our proposal focuses on expressiveness; *restricting* expressiveness is left to future work.

7.2 Aspects for functional languages

AspectML [8, 23] primarily focuses on the challenges in adapting an aspect system to a strongly-typed functional language. The authors choose not to permit advising anonymous functions or first-class functions. We permit advising these functions, as functions are frequently aliased to new variables or passed as arguments to functions. Our treatment of `stack` filters is inspired by AspectML's stack patterns.

AspectScheme [9] contends with the challenges of aliasing in an aspect system for a higher-order functional language. Their aspects are fully first-class: a pointcut is simply a boolean Scheme function on joinpoints. Additionally, they explore both static and dynamic scoping constructs for advice;

we focused solely on dynamic scoping. Their implementation depends heavily on Scheme's hygienic macros and continuation marks [6, 12], permitting a whole-program-transforming implementation (by redefining function application) within Scheme. While one could add continuation marks to a JS JIT [7], our approach exploits direct JIT integration instead.

Our handling of replacing variables with new getters and setters bears some resemblance to the `map-closure` operation [17]: like that work, we provide a first-class mechanism for JS to open closures and reveal and revise their code without disturbing their closed-over environments. The authors note in their discussion that aspect systems and `map-closure` are related, though their scopes are different: aspects are applied globally, while `map-closure` can be applied to a dynamic scope. This distinction is pragmatically eliminated by dynamic filters such as our `stack` filters.

7.3 Aspects within JavaScript

AOJS [24] is a recent prototype that implements weaving statically in a proxy server that modifies scripts before the browser sees them. This approach has several shortcomings, however, largely stemming from the choice to define weaving via preprocessing, rather than within JS. It supports two pointcuts—variable assignments and function calls, but not retrievals, `callee`, or filters. Pointcuts are implemented by replacing the variable or function name with calls to wrapper closures that in turn execute the advice and the original expressions. The approach relies explicitly on names and so does not avoid the aliasing problem, and it cannot handle anonymous or runtime-created functions.

AspectScript [21] is a concurrent and independent project designing aspects for JS. Like us, they advise closures rather than variables, to avoid the aliasing problem. They also explore different scoping strategies (similar to our `within` filter). However, their implementation is fundamentally different: weaving is implemented at runtime by parsing and rewriting scripts from within JS to wrap every potential joinpoint with a function. These "reifier" functions construct a context for the joinpoint and then call the weaver which in turn executes the relevant advice and the mainline code. All these rewritings and indirections come at cost: code is substantially larger and slower even when no aspects are deployed. For a browser whose interface is largely written in JS, and (despite many extensions) largely unmodified JS at that, such overhead is likely untenable. By contrast, our weaver causes zero code bloat and minimal runtime overhead. Additionally, introducing a second JS parser into a web browser is risky: it may not be bug-for-bug compatible with the underlying JS engine, leading to potentially incorrect results. Finally, AspectScript does not catch `eval`ed code or code that was loaded without being processed by their rewriter library: such code is not visible to their joinpoints and is not advisable. In our system, advice can use an arbitrary run-time expression to identify the closure to be advised and there are no restrictions on which closures are advisable.

7.4 Web Extension in Practice

Browser extensions are an exceptionally popular mechanism for users to customize and enhance their browser with features that are useful to some users, but not sufficiently so to warrant inclusion in the main product. We need to distinguish between *extensions* to the browser—such as custom search toolbars, themes, download managers—and *plugins* to the browser's supported content types—such as PDF renderers or movie codecs. Most plugins are implemented as DLLs. By contrast, extensions are a downloadable module of code that modifies the code of the underlying browser so as to implement a self-contained feature. We focus here on Firefox's extension model, as it is the most expressive and powerful for browsers to date. We have focused on JS, but extensions also include user interfaces written in the XML language XUL and styling written in CSS.

There are currently over six thousand Firefox extensions used by over thirty million people daily [16]. These extensions customize nearly every facet of the browser, changing tab handling, mouse interactions, file downloading behavior, etc. Many of these extensions replace existing functionality with new code, using the wrapping idiom where appropriate. Other extensions merely modify existing code slightly, using monkey-patching. Some actually modify *each other* to resolve compatibility issues. This last usage motivated our `field` and `statement_containing` pointcuts: for example, one extension (SplitBrowser[8]) modified another (All-in-One Tabs[9]) by replacing all accesses to a local variable with a field on a globally visible DOM node, so that it could see that interim state later as needed. Moreover, a reference to the DOM node was introduced as a new local variable for brevity. Clearly, these contortions are *ad hoc* and difficult to reason about without a more structured approach.

The userscript mechanism is similarly popular: nearly 40,000 scripts exist to tweak individual applications such as Gmail, YouTube, or Facebook, remove ads on popular pages, etc. The top five userscripts have each been installed over eleven million times. Like browser extensions, userscripts frequently interact with the structure or style of the DOM of the page, in addition to modifying its script content. Analogously, those latter modifications are better expressed and reasoned about using aspects.

8. Future work

JS aspects can also be used to good effect beyond extensions:

- Concurrent research by colleagues [15] examines using aspects to enforce security policies along the lines of Caja [20] or ADsafe [1]. Much of the challenge in freezing objects in Caja, for instance, requires interposing on any aliases to member functions in that object; our approach of advising the closure directly addresses this issue.

[8] `https://addons.mozilla.org/en-US/firefox/addon/4287`

[9] `https://addons.mozilla.org/en-US/firefox/addon/12`

- Libraries such as Script.aculo.us or Prototype build upon existing JS objects to provide convenient, common functionality for websites. Some of that functionality is fairly aspect-like; Prototype, for instance, defines functions to extend objects with getters and setters, and explicitly thread through `proceed` functions to call the next installed code. Aspects can help simplify their development while improving their performance.

The declarative nature of aspects opens up the possibility of new analyses. For example, recent work has focused on staging information flow analyses in the face of dynamic composition of scripts [5]. Precision is lost whenever code is `eval`ed; declarative aspects present more structure than `eval` strings, which may improve precision. For another, web-application and browser extensions are notoriously prone to breaking when certain other extensions are simultaneously installed, due in large part to the fragility of the code-injection idioms. Not only do aspects subsume all the weaving complexity of extensions, but they permit identifying when multiple extensions advise the same code and potentially conflict, which could provide useful warnings.

9. Conclusions

Web applications and browsers are growing ever more complex. Their broad, ad-hoc customizations highlight the need for an expressive mechanism by which to program them. In this work, we have implemented the first JIT for JS that supports aspects. We identified key linguistic requirements in the web-application space and described how they differ from prior aspect systems. Our aspect proposal for JS meets these requirements, and we have shown that it offers better performance and cleaner semantics, thereby improving the development of extensions.

References

[1] Adsafe, Nov. 2009. `http://www.adsafe.org/`.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, 2005.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.

[4] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *25th ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications*, 2010.

[5] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

[6] J. Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2005.

[7] J. Clements, A. Sundaram, and D. Herman. Implementing continuation marks in JavaScript. In *9th Scheme and Functional Programming Workshop*, 2008.

[8] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Trans. Program. Lang. Syst.*, 30(3):1–60, 2008.

[9] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63(3):207–239, 2006.

[10] ECMA International. ECMAScript language specification, 5th edition, Sept. 2009. `http://www.ecmascript.org/`.

[11] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *11th USENIX workshop on Hot topics in Operating Systems*, 2007.

[12] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding delimited and composable control to a production programming environment. In *12th ACM SIGPLAN International Conference on Functional Programming*, 2007.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, 2001.

[15] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[16] J. Scott. How many Firefox users use add-ons?, Aug. 2009. `http://blog.mozilla.com/addons/2009/08/11/how-many-firefox-users-use-add-ons/`.

[17] J. M. Siskind and B. A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *34th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2007.

[18] The AspectJ Team. The AspectJ programming guide, 2003. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[19] The AspectJ Team. The AspectJ 5 development kit developer's notebook, 2005. `http://www.eclipse.org/aspectj/doc/next/adk15notebook/`.

[20] The Caja Team. Caja, Nov. 2009. `http://code.google.com/p/google-caja/`.

[21] R. Toledo, P. Leger, and E. Tanter. AspectScript: expressive aspects for the web. In *8th International Conference on Aspect-Oriented Software Development*, 2010.

[22] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans.*

Softw. Eng. Methodol., 9(4):341–378, 2000.

[23] G. Washburn and S. Weirich. Good advice for type-directed programming: aspect-oriented programming and extensible generic functions. In *ACM SIGPLAN Workshop on Generic Programming*, 2006.

[24] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: Aspect-oriented JavaScript programming framework for web development. In *8th workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2009.