How do web pages run?

- **Web pages =**
  - HTML (structure)
  - CSS (style)
  - JS (behavior)

- **Extensions =**
  - New JS inserted into the page

```html
<html>
  ...
<body>
  <script>
    function msg() {
      return "Salutations";
    }
    body.onclick = "alert(msg());";
  </script>
  ...
</body>
</html>
```
Outline

- Motivation
  - Userscripts
  - Browser extensions

- Techniques and semantic flaws
  - Wrapping
  - Monkey-patching

- Language approach: weaving mechanism
  - Functions
Outline

- Motivation
  - Userscripts
  - Browser extensions

- Techniques and semantic flaws
  - Wrapping
  - Monkey-patching

- Language approach: weaving mechanism
  - Functions
Motivation: Userscripts

- Lightweight extensions to individual web pages
- Add or change features of the site in ways the site designer never anticipated
Key features of Userscripts

- **Execution**
  - Userscripts are appended to the page
  - Once added, they behave identically to page scripts

- **Popularity**
  - 60K scripts
  - 10M+ users
Motivation: Web-browser Extensions

- Downloadable code that customizes a browser
How do these extensions work?

- Can’t only *append new code to the page*
  - It won’t get called

- Need to *replace existing code too*

- Only two techniques available within JS:
  - Wrapping
  - Monkey patching
Wrapping

- “This function doesn’t quite do what I want; let me replace it”

```javascript
function P(iframe, data) {
    ...
}

function P(iframe, data) {
    if (data[0] == "mb")
        data[1] = format(data[1]);
    ...
}
```

- How?

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

- “This function doesn’t quite do what I want; let me tweak it”

Create a new closure and bind to existing name

foo

A closure

A closure’s `toString()` returns its source code

String-level search & replace
function XULBrowserWindow.setOverLink(link) {
    ...
}

eval("XULBrowserWindow.setOverLink = " +
    XULBrowserWindow.setOverLink.toString().replace(/\{/, "$\& link = Fission.setOverLink(link);");");

Idiom: $\&$ inserts whatever was matched

When does this code run?

What is link?

Idiom: the first { is always the start of the function

function XULBrowserWindow.setOverLink(link) {
    link = Fission.setOverLink(link);
    ...
}
Drawbacks of these approaches

- Incorrect for aliases
  - All other aliases are unmodified

```javascript
function foo(x) { return x*x; }

var bar = foo;

eval("foo = " + foo.toString().replace("x*x", "42");

> foo(5) == bar(5);
> false
```
So, don’t alias functions…?

- Function aliases are everywhere in web JS code
  - Installing event handlers creates aliases

```javascript
function onLoad(evt) { window.alert("hello"); }
window.addEventListener("load", onLoad, ...);
eval("onLoad = " + onLoad.toString.replace('hello', 'hi there'));
> ...loading the page...
> Alert: “hello”
```

- Needs a solution that works with existing web code
Drawbacks of these approaches

- Incorrect for closures
  - They are new closures that have the wrong environment

```javascript
function makeAdder(x) { return function(y){ return x+y; }; }
var addFive = makeAdder(5);
eval("addFive = " + addFive.toString.replace('y', 'z'));
> addFive(3)
> error: ‘x’ is undefined
```
Outline

- Motivation
  - Userscripts
  - Browser extensions

- Techniques and semantic flaws
  - Wrapping
  - Monkey-patching

- Language approach: weaving mechanism
  - Functions
Goal: combine extensions & mainline

- Extensions need to:
  - Define *what* new code to run
  - *When* it needs to run
  - *How* it interacts with existing code

- Sounds a lot like dynamic aspect weaving!
  - …Unless you’d rather we not call it “aspects”
  - These aren’t traditional “cross-cutting concerns”

- We *use the same mechanism, not the same motivation*
Aspects

- Aspects = Advice + Pointcuts

- Advice defines what new code to run
- Pointcuts define when to trigger it

```
at pointcut(callee(square)) before (x) {
    print("x is ", x);
}
```
Key features of our aspects

```java
at pointcut(callee(square)) before (x) {
    print("x is ", x);
}
```

This cannot be done in JS
Kinds of aspects

- Function advice:
  - Before, around, after calls to functions
  - Before, around, after bodies of functions

- Field advice:
  - Around getting, setting fields

- Statement advice:
  - Before, after, around statements within functions

- ...others?
Implementation: function advice

- Targeting a JIT compiler
- Key idea:
  weaving = inlining advice + invalidating JITed closure

- Inlining advice:
  - Avoids function-call overhead
  - Ensures advice has access to local variables

- Invalidating JITed closure:
  - Ensures next calls to function get the advice
  - Amortizes weaving cost across all calls to function
Conclusions

- Extensions have strange behavior
  - Existing techniques within JS are inadequate
  - But we can’t simply outlaw all extensions

- Introduced *dynamic aspect weaving* as new JS language feature
  - Provides cleaner semantics
  - Provides better performance
  - Provides sufficient expressive power for real extensions
  - Win-win!