Reveling in Constraints

The Google Web Toolkit is an end-run around Web development obstacles.

Bruce Johnson, Google

The Web's trajectory toward interactivity, which began with humble snippets of JavaScript used to validate HTML forms, has really started to accelerate of late. A new breed of Web applications is starting to emerge that sports increasingly interactive user interfaces based on direct manipulations of the browser DOM (document object model) via ever-increasing amounts of JavaScript. Google Wave, publicly demonstrated for the first time in May 2009 at the Google I/O Developer Conference in San Francisco, exemplifies this new style of Web application. Instead of being implemented as a sequence of individual HTML "pages" rendered by the server, Wave might be described as a client/server application in which the client is a browser executing a JavaScript application, while the server is "the cloud."

The key browser technologies responsible for enabling this new generation of Web applications are not especially new: JavaScript runs within the browser to manipulate the browser DOM as a means for actually rendering the UI and responding to user events; CSS (cascading style sheets) are used to control the visual style of the UI; and the XHR (XmlHttpRequest) subsystem allows JavaScript application code to communicate asynchronously with a Web server without requiring a full-page refresh, thus making incremental UI updates possible. There are many more browser technologies that read like alphabet soup: XML, VML, SVG, JSON, XHTML, DTD... the list goes on.

Curiously, these browser technologies have been available for many years, yet it has taken until now for mainstream developers to cobble them together to create compellingly interactive Web applications. Why? The opinion of the Google Web Toolkit team—a perspective that can, of course, be endlessly debated—is that the primary obstacle is literally the implementation details. It is simply too hard to code them all to work together in a way that provides quick and reliable performance on the wide range of browsers available.

Our response was to design GWT (Google Web Toolkit), which allows developers to spend most of their time writing and debugging application code using the Java language rather than JavaScript. Working in Java means developers can leverage the productivity of Java IDEs (integrated development environments). Then, once they're happy with their Java code, developers can use GWT's cross-compiler to convert Java source code into functionally equivalent, and optimized, JavaScript. The idea of cross-compilation tends to raise eyebrows, and we've heard more than our fair share of incredulity about this, so let's take a step back to describe how we decided on this approach—and how things have actually worked out.

GWT began life as a prototype that Google software engineer Joel Webber and I produced as a way to address what might best be described as the over-constrained problem of Web development. Thanks largely to the success of Google Maps and Gmail, several points had simultaneously become clear to us:

- End users really liked and wanted browser-based applications.
- Rich client-side interactivity (for example, Maps and Gmail) made such applications much more

responsive and usable than typical Web 1.0 page-at-a-time applications and thus much more compelling.

- Each major browser was technically capable of enabling such interactive applications. They could all run JavaScript and support dynamic HTML, but bugs, inconsistencies, and proprietary APIs and behaviors prevented any single JavaScript program from working consistently and efficiently on the majority of the modern browsers.
- Support for the JavaScript language itself—that is, for the “pure” language syntax and core JS libraries, excluding DOM APIs—was, to our surprise, quite consistent and reliable across browsers.

In other words, the browser—in particular, XHR, JavaScript, and the DOM—presented a capable, albeit frustrating, platform for delivering applications.

JAVASCRIPT RESERVATIONS

At the same time, we had questions about whether JavaScript was a good language in which to write business-critical applications. On the one hand, JavaScript is a flexible, dynamically typed language that makes certain types of code easy to write and pleasantly succinct. On the other hand, that same flexibility can make JavaScript harder to use within a team environment because there is no easy way to enforce the use of consistent conventions automatically across an entire codebase. It is true that, with significant extra work, a JavaScript team could insist that all script be augmented with extra metadata (JSDoc, for example) and then use additional tools to verify that all script complies with the agreed-upon conventions. This would also necessarily restrict the developers to a statically analyzable subset of JavaScript, since some of JavaScript’s most dynamic constructs—`eval()` and the `with` statement are good examples—thoroughly defeat static analysis. All this extra stuff—the metadata and verification tools—seemed an awful lot like an ad-hoc static type system and a compiler front end.

Furthermore, we badly wanted an IDE. Our experience had thoroughly convinced us that IDEs are a boon to productivity, quality, and maintenance. Features that are status quo in modern Java IDEs such as code completion, debugging, integrated unit testing, refactoring, and syntax-aware search were virtually nonexistent for JavaScript. The reason for this, again, is related to the dynamism of JavaScript. For example, it isn’t possible to provide sound code completion in a JavaScript editor in the general case because different runtime code paths can produce different meanings for the same symbols. Consider this legal JavaScript:

```
function foo(m) {
  alert("You called foo with the message: " + m);
}

if (dayOfWeek() == "Wednesday") {
  foo = 3;
}

foo("Hello?");    // [1]
```

At [1], it’s impossible to tell statically whether `foo` is a function or a variable, so IDE code

completion can only provide “potentially correct” suggestions, which is an optimistic way of saying that you must double-check the IDE’s code completion suggestions, which in turn is likely to diminish much of the would-be productivity gain to be realized from a JavaScript IDE. For similar reasons, automated refactoring tools for JavaScript are rarely seen, even while such tools are ubiquitous in the Java world. These observations made JavaScript seem less attractive as a language in which to write large applications.

We finally realized that we wanted to *develop* our source code in the Java language, yet *deploy* it as pure JavaScript. By choosing the Java language as the origination language, we could immediately leverage the great ecosystem of Java tools, especially the great Java IDEs out there. The only question was how to produce JavaScript from the Java source input. Our answer was to build a Java-to-JavaScript compiler—an optimizing compiler, in fact, because we figured that since we were going to the trouble of writing a compiler anyway, why not make sure it produced small, efficient JavaScript? Furthermore, we discovered that because Java has a static type system, it allowed for many compile-time optimizations that JavaScript—being dynamically typed—would not.

As an example of this, consider the interaction between inlining and devirtualization (that is, the removal of polymorphism in a method invocation). In JavaScript, developers often simulate object-oriented constructs such as polymorphism. For example, if you wanted to have a simple Shape hierarchy in JavaScript, you might write this:

```
function Shape() { }
Shape.prototype.getArea = function() { }

function Circle(radius) { this.radius = radius; }
Circle.prototype = new Shape();
Circle.prototype.getArea = function() { return this.radius * this.radius * Math.PI; }

function Square(length) { this.length = length; }
Square.prototype = new Shape();
Square.prototype.getArea = function() { return this.length * this.length; }

function displayArea(shape) { alert("The area is " + shape.getArea()); }

function runTest() {
    var shape1 = new Circle(3);
    var shape2 = new Square(2);
    displayArea(shape1);
    displayArea(shape2);
}
```

The same thing written in the Java language for use with GWT might look like this:

```

abstract class Shape {
    public abstract double getArea();
}

class Circle extends Shape {
    private final double radius;
    public Circle(double radius) { this.radius = radius; }
    @Override public double getArea() { return radius * radius * Math.PI; }
}

class Square extends Shape {
    private final double length;
    public Square(double length) { this.length = length; }
    @Override public double getArea() { return length * length; }
}

static void displayArea(Shape shape) { Window.alert("The area is " + shape.getArea()); }

static void runTest() {
    Shape shape1 = new Circle(3);
    Shape shape2 = new Square(2);
    displayArea(shape1);
    displayArea(shape2);
}

```

The source for the two examples looks nearly identical, except for minor syntax differences, the use of `@Override` (which is useful for helping to prevent bugs), and the presence of explicit type names sprinkled on fields, methods, and local variables.

Because of the extra type information, the GWT compiler is able to perform some optimizations. The unobfuscated version of the GWT compiler output looks approximately like this:

```

function runTest() {
    var shape1, shape2;
    shape1 = $Circle(new Circle(), 3);
    shape2 = $Square(new Square(), 2);
    alert('The area is ' + shape1.radius * shape1.radius * 3.141592653589793); // [1]
    alert('The area is ' + shape2.length * shape2.length); // [2]
}

```

Note that in [1] and [2] above, a cascade of optimizations was made.

First, the compiler inlined both calls to the `displayArea()` method. This proved helpful because it removed the need to generate code for that method. Indeed, `displayArea()` is completely absent from the compiled script, resulting in a minor size reduction. Even better, the inlined code is amenable

to being further optimized in a usage-specific context where more information is available to the optimizer.

Next, the optimizer noticed that the types of `shape1` and `shape2` could be “tightened” to types more specific than their original declaration. In other words, although `shape1` was declared to be a `Shape`, the compiler saw that it was actually a `Circle`. Similarly, the type of `shape2` was tightened to `Square`. Consequently, the calls to `getArea()` in [1] and [2] were made more specific. The former became a call to `Circle`’s `getArea()`, and the latter became a call to `Square`’s `getArea()`. Thus, all the method calls were statically bound, and all polymorphism was removed.

Finally, with all polymorphism removed, the optimizer inlined `Circle`’s `getArea()` into [1] and `Square`’s `getArea()` into [2]. Both `getArea()` methods are absent from the compiled script, having been inlined away. `Math.PI` is a compile-time constant and was also trivially inlined into [1].

The benefit of all these optimizations is speed. The script produced by the GWT compiler executes more quickly because it eliminates multiple levels of function calls.

For obvious reasons, large codebases tend to be written with an emphasis on clarity and maintainability rather than just on sheer performance. When it comes to maintainability, abstraction, reuse, and modularity are absolute cornerstones. Yet, in the previous example, maintainability and performance come into direct conflict: the inlined code is faster, yet no software engineer would write it that way. The “maintainability vs. performance” dichotomy isn’t unique to Java code, of course. It is equally true that writing modular, maintainable JavaScript tends to produce slower, larger script than one would prefer. Thus, all developers building complex Web applications have to face the reality of this trade-off. The pivotal question would seem to be just how amenable your codebase is to optimization once you’ve written it. In that regard, the Java type system provides great leverage, and that is how the GWT compiler is able to include many optimizations similar to the ones shown here to help mitigate the “abstraction penalty” you might otherwise end up having to pay in a well-designed object-oriented codebase.

BRINGING TOGETHER TWO WORLDS

Of course, the creation of an environment that allows developers to build browser-based applications in Java addresses only one part of the development cycle. Like most developers, we do not produce perfect code, so we knew we would also have to address the issues involved in debugging GWT programs.

Upon first hearing about GWT, people often assume you use it in the following way:

1. Write Java source.
2. Compile to JavaScript with GWT’s compiler.
3. Run and debug the JavaScript in a browser.

In fact, that is not the way you work in GWT at all. You spend most of your time in GWT’s *hosted mode*, which allows you to run and debug your Java code in a normal Java debugger (for example, Eclipse), just as you’re accustomed to doing. Only when the application is written and debugged do you need actually to compile it into JavaScript. Thus, everyone’s reflexive fear of never being able to understand and debug the compiled JavaScript proves to be unfounded.

The secret to making hosted mode an effective debugging environment is that it does not merely *simulate* the behavior of a browser while debugging in Java. Hosted mode directly combines true Java debugging with a real browser UI and event system. Hosted mode is conceptually simple, and it executes in a single JVM (Java Virtual Machine) process:

1. Launch an instance of an actual browser, embedded in-process, that can be controlled by Java code via JNI (Java Native Interface). We call this the *hosted browser*.
2. Create a CCL (compiling class loader) to load the GWT module's entry-point classes.
3. Whenever the CCL is asked to fetch a class, it checks to see if the class has JSNI (JavaScript Native Interface) methods. If not, the class can be used directly. If native methods are found, the class gets compiled from source and the JSNI methods are rewritten.
4. Run the bytecode of the entry-point class, which will in turn request other classes be loaded by the CCL, which repeats the process from step 3.

Step 3, rewriting JSNI methods, is the really neat part here. JSNI is the way to implement native Java methods in handwritten JavaScript, such as:

```
// This is Java!
static native Element createDivElement() /*-{
    // This is JavaScript!
    return document.createElement("div");
}*/;
```

The CCL knows about JSNI and rewrites it to look something like this:

```
// A static initializer is introduced in the class.
static {
    hostedBrowser.injectFunc("createDivElement", "return document.createElement(\"div\");");
}
```

```
// The method becomes all-Java and is no longer native.
static Element createDivElement() {
    return hostedBrowser.invokeInjectedFunc(this, "createDivElement");
};
```

Thus, the hosted-mode CCL turns JSNI methods into thunks that redirect their calls into the hosted browser's JavaScript engine, which in turn drives the real browser DOM.

From the JVM's point of view, everything described here is pure Java bytecode and can therefore be debugged normally using a Java debugger. From the developer's point of view, he or she can see the true behavior of a real browser being driven by the Java source code without it first having been cross-compiled into pure JavaScript.

Which brings up perhaps the most exciting point about hosted mode: because it works dynamically with Java code and does not depend on invoking the GWT cross-compiler (which can be slow), hosted mode is really fast. This means developers get the same kind of run/tweak/refresh behavior they enjoy whenever working directly with JavaScript.

GWT thus manages to combine the benefits of a traditional optimizing compiler with the quick development turn-around of dynamic languages. Although the compilation technology may appear complex, it is actually fairly standard fare for optimizing compilers. The real technical problems

we encountered along the way revolved around our efforts to create UI libraries to simultaneously account for browser-specific quirks without compromising size or speed. In other words, we needed to supply many different implementations of UI functionality—version A for Firefox, version B for Safari, and so forth—without burdening the compiled application with the union of all the variations, thereby forcing each browser to download at least some amount of irrelevant code. Our solution is a unique mechanism we dubbed *deferred binding*, which arranges for the GWT compiler to produce not one output script, but an arbitrary number of them, each optimized for a particular set of circumstances.

Each compiled output is a combination of many different implementation choices, such that each script has exactly (and only) the amount of code it requires. It's worth mentioning that in addition to dealing with browser variations, deferred binding can specialize compilations along other axes as well. For example, deferred binding is used to create per-locale specializations (for example, why should a French user have to download strings localized for English, or vice versa?). In fact, deferred binding is completely open ended, so developers can add axes of specialization based on their needs.

This approach does create a large number of compiled scripts, but we reasoned it was a welcome trade-off: you end up spending cheap server disk space on many optimized scripts, and, as a result, applications download and run more quickly, making end users happier.

In any event, our experience in developing GWT has thoroughly convinced us that there's no need to give in to the typical constraints of Web development. That is, with a bit of creativity and some dedicated effort, we now know it is indeed possible to retain the richness of more familiar development environments without compromising the experience application users are ultimately to enjoy. ¶

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

BRUCE JOHNSON founded Google's engineering office in Atlanta, right next door to his alma mater Georgia Tech, with the goal of producing Google Web Toolkit and a number of related tools intended to make Web development more efficient, effective, and a whole lot more fun.

© 2009 ACM 1542-7730/09/0700 \$10.00