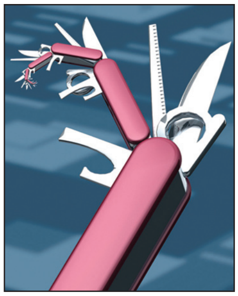
Editor: Steve Vinoski • *vinoski@ieee.org*

ClojureScript: Functional Programming for JavaScript Platforms



Mark McGranaghan • *Heroku*

Functional programming is a productive approach to writing software that can be successfully applied to Web development. Functional Web development has historically focused on the server side with languages such as Erlang, Haskell, Scala, and Clojure. However, full-stack Web applications increasingly rely on sophisticated client-side components that must execute in the browser's JavaScript runtime environment.

Clojure (http://clojure.org) is a dynamic, Lisp-like programming language that originally targeted the Java Virtual Machine (JVM). It's been successful on the JVM platform because of its combination of expressiveness, performance, and host interoperability. ClojureScript brings these qualities to JavaScript platforms with a ClojureScript-to-JavaScript compiler and associated tool chain based on the Google Closure suite (http://code.google.com/closure/).

Here, I introduce the ClojureScript language and its Google Closure substrate, demonstrate how to use ClojureScript in dynamic client-side Web applications, and discuss ClojureScript's unique approach to JavaScript compilation.

Getting Started

Like Clojure, ClojureScript is a Lisp that you can explore interactively at the *read-eval-print* loop (REPL). ClojureScript's developers have paid particular attention to being able to get such a REPL up and running quickly. Let's take advantage of that to try ClojureScript.

First, download and enter the ClojureScript project:

```
$ git clone \
git://github.com/clojure/clojurescript
$ cd clojurescript
```

Bootstrap your ClojureScript installation by downloading Clojure and the Google Closure tools:

```
$ script/bootstrap
```

Now, you're set to explore the ClojureScript REPL:

```
$ script/repljs
```

This REPL will be familiar to users of Clojure and other dynamic languages; the prompt reads the typed expressions, evaluates them on the fly, and prints the results:

```
ClojureScript:cljs.user> (+ 1 2 3)
6
```

Here, we use the typical Clojure list-based functional call syntax and take advantage of the variadic + function defined in ClojureScript. In general, the syntax and semantics of ClojureScript's core language features are similar to those of Clojure, even though ClojureScript executes in a JavaScript environment, whereas Clojure executes on a JVM.

The standard Clojure data types and structures, and their associated manipulation

1089-7801/11/$26.00 © 2011 IEEE
Published by the IEEE Computer Society

```
(ns demo
 (:require
  [clojure.set :as set]
  [clojure.string :as string]))

(def healthy
 #{"lettuce" "fish" "apples" "carrots"})

(def tasty
 #{"apples" "cake" "candy" "fish"})

(defn ^:export talk []
 (js/alert
  (str
   "Some healthy and tasty foods are: "
    (string/join ", " (set/intersection healthy tasty)))))
```

Figure 1. Set and string manipulation with ClojureScript (demo.cljs).

functions, are also available in ClojureScript:

```
ClojureScript:cljs.user>
   (def person
  (-> {:name "Bob"} (assoc
    :occupation "Programmer")))
{:name "Bob", :occupation
   "Programmer"}

ClojureScript:cljs.user>
   (def attrs (keys person))
(:name :occupation)

ClojureScript:cljs.user>
   (conj (set attrs) :height)
#{:height :occupation :name}
```

This short series of expressions demonstrates several features that ClojureScript brings to the JavaScript environment. In this first expression, we take a literal map `{:name "Bob"}` with a keyword key `:name` and use the `->` macro in combination with the `assoc` function to produce a new, updated version of that map. As in Clojure, this update is functional and doesn't mutate the original map. In the second expression, we extract the sequence of keys from that resulting map, and in the third, we transform that sequence into a set and add a third element to that set.

These latter two examples illustrate two specific data abstractions — the sequence and the set — that ClojureScript brings to the JavaScript environment. Indeed, JavaScript itself offers only one data structure — an associative array with string keys — but the ClojureScript compiler and runtime library transparently provide a full suite of functional data structures on top of this JavaScript primitive.

These examples also highlight a subtle but important aspect of ClojureScript: it's a semantics-altering compiler as opposed to a syntactic layer above JavaScript. CoffeeScript and several other LISP-to-JavaScript compilers take the latter approach, but ClojureScripts's semantics-level approach is ultimately what lets it bring the robustness of functional programming to the browser.

Although a ClojureScript application benefits from access to Clojure-like language and runtime facilities, it can still easily participate in the host JavaScript environment. For example,

```
ClojureScript:cljs.user>
   (.toFixed 0.9876 2)
"0.99"
```

Here, we use the `.toFixed` function that's defined on JavaScript numbers to convert a float to a string. As in Clojure itself, access to host features and libraries is designed to be efficient in terms of both syntax and runtime execution. This first-class JavaScript host access is important for ClojureScript when it's interoperating with browser facilities and pure-JavaScript libraries.

Compilation and Deployment

Although the ClojureScript REPL is useful for exploration, ClojureScript is designed to be compiled for efficient deployment to Web browsers and other client environments. Let's try a simple Web-based ClojureScript example to demonstrate this.

Figure 1 shows the basic ClojureScript file for our application.

When we invoke the `talk` function defined in this ClojureScript, it will use ClojureScript set and string manipulation and the JavaScript `alert` function to render a list of recommended foods to the user.

To run this code, create a webpage in which to host the compiled ClojureScript (see Figure 2).

Now, we need to compile the ClojureScript to the target location expected by the `script` tag. ClojureScript ships with a command-line tool `cljsc` that serves as a simple bridge to a Clojure-based compilation library. We'll use this tool to compile our demo ClojureScript source file:

```
$ bin/cljsc demo.cljs \
{:optimizations :advanced} >\
demo.js
```

Open `index.html` in a Web browser and click on the "what to eat?" button; an alert triggered by the compiled ClojureScript code should pop up showing you some options.

Now that we've looked at some basic examples of ClojureScript, let's explore a more sophisticated application and the ClojureScript compilation model itself.

An Example ClojureScript Application

A more complete ClojureScript example application can help us better understand what ClojureScript offers, how it works, and how it interacts with its JavaScript host.

This example app will implement an interactive ClojureScript-to-HTML renderer. It will work by reading text input from the user, parsing that into ClojureScript data, rendering the data into HTML text, and then displaying that text in real time on the page. For example, the app might read the following ClojureScript data structure:

```
[:div {:id "demo"}
  "hello world!"]
```

and render it to this HTML snippet, which would then be displayed in the browser:

```
<div id="demo">hello
  world!</div>
```

The app will use the ClojureScript standard libraries along with the Google Closure DOM and browser event libraries, and will execute entirely client-side in the user's browser. All this client-side code will pass through the ClojureScript/Closure optimizing compiler tool chain for delivery to the running app.

Figure 3 demonstrates the app's user-facing shape with the static HTML component.

This file is a simple HTML skeleton for the app's layout and a hook for the dynamic ClojureScript piece of the application. Note that the app includes three fields: a text area where users can enter their Clojure input, an output area for the compiled HTML, and an output area for the rendered HTML. Figure 3 also includes some sample input text to demonstrate how the tool works and provide initial test data for the renderer.

The ClojureScript component will update the page whenever you change the input text. These updates will be implemented with a combination of core ClojureScript libraries as well as browser libraries from Google Closure. We'll `require` these libraries into a renderer ClojureScript namespace definition (see Figure 4).

ClojureScript namespaces are similar to Clojure namespaces: they explicitly define the dependencies for their portion of the application

```
<html>
 <head>
  <script type="text/javascript" src="demo.js">
  </script>
 </head>
 <body>
  <input type="button" value="what to eat?"
   onClick="demo.talk()">
 </body>
</html>
```

Figure 2. HTML skeleton for the demo ClojureScript application (demo.html).

```
<html>
 <head>
  <title>Renderer</title>
 </head>
 <body>
  <h1>Renderer</h1>
  <h2>Input</h2>
  <textarea id="input-text" rows="6" cols="82">
[:div {:id "demo"} [:h3 {} "Hello ClojureScript!"]]
  </textarea>
  <h2>Output - Compiled</h2>
  <div id="output-compiled"></div>
  <h2>Output - Rendered</h2>
  <div id="output-rendered"></div>
  <script type="text/javascript" src="renderer.js">
  </script>
 </body>
</html>
```

Figure 3. Skeleton, JavaScript hook, and example input (renderer.html).

```
(ns renderer
 (:require
  [clojure.string :as string]
  [cljs.reader :as reader]
  [goog.dom :as dom]
  [goog.events :as events]
  [goog.events.EventType :as event-type]))
```

Figure 4. Declaring the renderer namespace and its dependencies (renderer.cljs).

```
(defn attrs-props [attrs]
 (string/join " "
  (map
   (fn [[k v]]
    (str " " (name k) "=\"" v "\""))
   attrs)))

(defn closing-tag [tag attrs]
 (str "<" tag (attrs-props attrs)" />"))

(defn wrapping-tag [tag attrs inner]
 (str "<" tag (attrs-props attrs)">"
    inner
    "</" tag ">"))

(defn compile-form [form]
 (cond
  (vector? form)
   (let [[tag attrs & body] form]
    (if (seq body)
     (wrapping-tag (name tag) attrs
      (apply str (map compile-form body)))
     (closing-tag (name tag) attrs)))
  (seq? form)
   (apply str (map compile-form form))
  :else
   (str form)))
```

Figure 5. Simple HTML compiler implementation (renderer.cljs).

```
(defn render [& _]
 (let [input-text (.value (dom/getElement "input-text"))
    input-form (reader/read-string input-text)
    output-compiled(compile-form input-form)]
  (dom/setTextContent
   (dom/getElement "output-compiled")
    output-compiled)
  (set!
   (.innerHTML (dom/getElement "output-rendered"))
   output-compiled)))

(defn init []
 (events/listen (dom/getElement "input-text")
  event-type/KEYUP render)
 (render))

(init)
```

Figure 6. Wiring it up. We tie the HTML compiler into the actual webpage using a browser event listener implemented by the Google Closure library.

and define aliases with which we can easily address these dependent namespaces. ClojureScript provides the `clojure.*` and `cljs.*` namespaces, while those under `goog.*` come from the Google Closure library.

At the core of this ClojureScript application is the actual HTML compiler. This compiler will take as input a ClojureScript data structure and return as output HTML text. This transformation is seen in Clojure libraries such as clj-html and hiccup; Figure 5 shows a simple definition to demonstrate ClojureScript usage.

Finally, we tie the HTML compiler into the actual webpage using a browser event listener implemented by the Google Closure library (see Figure 6).

Next, we define the core render loop in the `render` function. This function will extract the user input area's contents, use the ClojureScript `read-string` function to map that input into a ClojureScript data structure, pass that form into the compiler described previously, and then render that output as both a raw HTML string and actual content on the webpage.

The `events/listen` call in the `init` function uses the Closure browser events library to register a callback on text changes in the input field. Making changes to this field invokes the `render` function, causing the full rendering sequence to execute.

The `(init)` call at the bottom of the file will be invoked once the browser has loaded all the JavaScript; this function will register the event listener and execute an initial rendering.

Compile this complete ClojureScript file the same way you did the demo app:

```
$ bin/cljsc renderer.cljs \
{:optimizations :advanced} > \
renderer.js
```

Now you should be able to open the `renderer.html` file in your browser, type more code into the input field, and see the real-time HTML rendering in action.

The ClojureScript Compilation Model

We've seen that ClojureScript can execute a program with Clojure-like source code in a JavaScript-based browser environment. This execution is facilitated by the ClojureScript/Closure compiler tool chain. Let's look at that compilation process and see why ClojureScript's unique approach to compilation is essential to the language's practical application.

The ClojureScript compiler implements the initial phase of compilation. The compiler is written in Clojure as a recursive descent parser/analyzer/emitter. It's relatively simple because the Clojure semantics that it's compiling for are simple, and because its JavaScript target is itself a rich language (as compared to, for example, the JVM bytecode that Clojure itself targets).

As an example of this first level of ClojureScript compilation, consider this simple ClojureScript namespace:

```
(ns compiler)

(defn add-two [n1 n2]
 (+ n1 n2))

(defn ^:export calc [a b c]
 (let [ab (add-two a b)]
  (add-two ab c)))
```

We can observe the first stage of compilation by explicitly avoiding optimization phases:

```
$ bin/cljsc compiler.cljs \
{:optimizations false \
:pretty-print true} > \
compiler.js
```

This unoptimized compilation produces several JavaScript files, one of which contains a snippet like the one in Figure 7.

```
goog.provide('compiler');
goog.require('cljs.core');
compiler.add_two = (function add_two(n1,n2){
 return cljs.core._PLUS_.call(null,n1,n2);
});
compiler.calc = (function calc(a,b,c){
 var ab__1977 = compiler.add_two.call(null,a,b);
 return compiler.add_two.call(null,ab__1977,c);
});
goog.exportSymbol('compiler.calc', compiler.calc);
```

Figure 7. Snippet of ClojureScript compiler output.

The mapping between the namespace and function definitions in our original ClojureScript source is quite clear. This straightforward compilation of Clojure to JavaScript would be useful in itself, but it does have problems for production applications. In particular, if we're compiling a substantial application that depends on the large Closure and ClojureScript libraries, the number and size of the resulting JavaScript files will be too large for fast delivery to bandwidth-constrained Web clients.

To address this problem, ClojureScript leverages Closure's sophisticated whole-program JavaScript optimizer. The optimization process starts when ClojureScript generates Closure-compatible JavaScript source code. Such code explicitly declares its namespace imports and exports so that the Closure compiler understands the program's structure. We see such declarations in Figure 7, with `goog.require('cljs.core')` and `goog.exportSymbol('compiler.calc', compiler.calc)`.

The ClojureScript build program can then feed the output of the ClojureScript compiler for a given application, along with the compiled ClojureScript core library and the Google Closure JavaScript library, into the Closure compiler. This compiler uses the dependency metadata available from the specially formatted JavaScript source files to build a whole-program dependency tree, eliminate all function definitions that aren't reachable by specific applications, rewrite variable names and eliminate comments and whitespace to reduce code size, and emit a single file containing all the resulting JavaScript.

Executing this multistep compilation process by hand would be arduous, but ClojureScript provides a simple interface to this Closure tool chain. Indeed, we used it earlier for our demo apps. To better see how this optimizing compiler works, let's compile our simple test code from earlier:

```
$ bin/cljsc compiler.cljs \
{:optimizations :advanced} > \
compiler.js
```

If you examine the compilation output in `compiler.js`, you'll see JavaScript with very short variable names, no comments, and with almost all whitespace eliminated. It might not even be clear that this code corresponds to the original application source or the ClojureScript library. However, if you search this source for `"compiler.calc"`, you'll find the compiled code corresponding to our original compiler namespace. It will look something like this (whitespace re-introduced for clarity):

```
function nc(a,c) {
 return pb.call(f,a,c)
}
function oc(a,c,d) {
 a = nc.call(f,a,c);
 return nc.call(f,a,d)
}
var pc = "compiler.calc".
  split(".")
```

As you can see, the compiled output is semantically similar to the nonoptimized output, but variable references have been renamed and inlined to minimize code size while preserving application behavior.

Also note that the `compiler.js` file is itself only 34 Kbytes. This is many times smaller than just the ClojureScript core library source code size; we achieve this reduction in code size via the aggressive whole-program optimization and minification the Closure compiler performs.

ClojureScript is a young language and ecosystem, but its future as a tool for JavaScript platforms is promising. JavaScript has massive reach, and indeed is now a required target for any comprehensive Web or mobile application project. As the client-side portions of Web and mobile applications become more sophisticated, ClojureScript is well positioned to bring the robustness of functional programming and the elegance of Clojure to these JavaScript environments.

Mark McGranaghan is an engineer at Heroku. Contact him at mark@heroku.com.