The technology of ClearCommit

Zach Allaun *2013 November 13*

This is a guest post by Zach Allaun; see also the previous ClearCommit post.

ClearCommit is a multi-language, semantic diff tool for code bases built in 48 hours by Ryan Lucas, Kevin Lynagh, and Zach Allaun for the 2013 Node Knockout. You can check out the live app and see what it can do on our NKO team page.

At a high level, ClearCommit analyzes and summarizes codebases at specific git commits. ClearCommit extracts from each commit the codebase elements: functions, classes, CSS selectors, etc. By comparing two summaries you can extract useful information about what's changed in a codebase. For instance, ClearCommit can be used to see what a GitHub pull request would change, answering questions like, "How should I bump the semantic version of my project if I accept this?"

We set out to build something sophisticated in a short amount of time, so we brought out the big guns.

CLOJURESCRIPT

ClojureScript is Clojure compiled to JavaScript, and we chose it for a number of reasons. Unlike many other popular compile-to-JS languages like CoffeeScript,

CLJS offers significant semantic improvements over vanilla JS: functional programming, immutable data, safe extension with protocols, and program transformations à la core.async and core.match. Furthermore, these improvements don't sacrifice host-platform interop: there's no need to reinvent the wheel, because JS libraries can be easily wrapped or called directly. (Examples of this further down.)

We used CLJS both on the frontend and backend, coupled with Angular and Node. ClearCommit's architecture is entirely service-oriented; the backend analyzes code and exposes the results through a JSON API, and the frontend (a JavaScript app served by nginx) handles presentation. This decoupling was great for remote development under pressure; there was no stepping on each other's toes.

Kevin has already written about building an app with Angular and ClojureScript, so I'll instead focus on running ClojureScript on Node.

ClojureScript on Node

As I mentioned earlier, JS interop is a key feature of ClojureScript, and that definitely proved to be the case with Node. For instance, we used the request module from ClojureScript as a simple HTTP client.

```
(def ^:private js-request (nodejs/require "request"))

(defn request [url-or-opts callback]
  (js-request (clj->js url-or-opts) callback))

(request {:method "GET" :uri "http://www.google.com"}
         (fn [error response body]
           (when (and (not error) (= 200 (.-statusCode response)))
             (println body))))
```

This is a straightforward wrapping which allows us to pass in Clojure data structures instead of JavaScript ones. However, this is still very similar to

callback-based JavaScript. The bigger win is using core.async to change the semantics altogether:

```
(defn request [url-or-opts]
  (let [c-resp (async/chan 1)]
    (js-request (clj->js)
                (fn [error response body]
                  (async/put! c-resp
                              (if error
                                {:error error}
                                {:response response :body body})
                              #(async/close! c-resp))))
    c-resp))
```

Now, instead of passing a callback to `request`, a core.async channel will be returned, ultimately containing either a response or an error. (For more about core.async, see David Nolen's Communicating Sequential Processes.) This technique works great combined with a library like core.match, which gives us pattern-matching:

```
(async/go
  (match [(<! (request "http://www.google.com"))]
    [{:error error}] (println "fail")
    [{:body body}]   (println body)))
```

Core.async's `go` does a whole-program transformation, turning what looks like synchronous code into asyncronous, callback-based code. This is quite a lot of power given to us essentially for free.

Core.async also provides quite elegant solutions to common, tricky problems in asyncronous callback-based code. One such problem is the need to execute a number of IO operations in parallel and then synchronize on the result once they all finish. We accomplished this using a (perhaps poorly named) helper function, `parallel-chan`.

```
(defn parallel-chan
  "Takes one value from all vals of `map-of-chans`,
   returning a map with the same keys and those values.
   All takes are done in parallel via `alts!`."
  [map-of-chans]
  (go-loop [res {}
            chans (map-invert map-of-chans)]
    (if (empty? chans)
      res
      (let [[val c] (alts! (keys chans))]
        (recur (assoc res (get chans c) val)
               (dissoc chans c))))))
```

This allowed us to pass in a map of names to channels, and recieve a channel that eventually contains a map of names to results:

```
(go
  (let [results (<! (parallel-chan {:web1 (web-request-1)
                                    :web2 (web-request 2)}))]
    ;; ^^ this parks the go-block until both :web1 and :web2 have finished
    (println (:body (:web1 results)))
    (println (:body (:web2 results)))))
```

We found it much easier to write straight-forward code using these methods than it otherwise would be using callbacks. It was also trivial to wrap necessary portions of Node's async API, returning core.async channels instead of requiring callbacks.

It wasn't all roses, though. Core.async seems to be a huge semantic win over other solutions like callbacks, but debugging can be a right pain. Not only are you frequently looking at stack traces in terms of emitted JS instead of CLJS source, but errors thrown in core.async state machines show up there in traces, instead of in the code that originally constructed them. Thankfully, this won't be a permanent issue, as ClojureScript source maps have come a long way and

seem to be getting better by the day. I wasn't able to get them to work with Node on such short notice, there is light at the end of the tunnel.

ANALYSIS OVERVIEW

When ClearCommit recieves a request (either via a @clearcommit mention on a GitHub comment or a plain HTTP request), the following needs to happen:

1. Retrieve all of the necessary Git data from GitHub. This means commit metadata, trees, and blobs.
2. Parse blobs into abstract syntax trees. We used existing parsers for JavaScript and CSS, but hacked together our own for Clojure/Script using the built-in reader.
3. Transform ASTs into something more standardized. This is where we identify the elements of the code base – functions in the case of JavaScript and Clojure/Script, and selectors in the case of CSS. The output of this phase is the same format regardless of language.
4. Merge the analyzed ASTs into a single "analyzed commit" containing the elements from every file in the repo. This is the end of the line if someone's just looking for analyzed commits.
5. If someone's requested a semantic diff, diff the data structure returned from the commit analysis phase. Because these analyzed commits have a standard structure, we get to use the same diffing code for all languages.

The diff algorithm itself is quite simple, but thus far seems relatively effective. We identify which elements have been added and removed, and then which elements have been changed. For the elements that have been changed, we figure out which sub-components have been changed. These diffed elements will ultimately be a part of the JSON response, and look like this:

```
{name: ["foo" "function"], params: {before: ["x", "y"], after: ["x"]}}
```

(Our decision to standardize on the format of an analyzed commit made it possible to add support for diffing CSS in 30 minutes, just a couple of hours before the competition ended. We didn't want to have to change our copy, which already included CSS as a supported language =).)

One nice property of building something on top of Git data is that Git data is immutable. This means we can aggressively cache at every stage of the pipeline, and we do. Every stage returns JSON, and it's all written to the disk based on a SHA. The locations on disk match our JSON API, and we were able to configure nginx to serve the files directly if they exist, and dispatch to Node if they don't. This means that most repeated requests don't hit our Node server at all!

WRAPPING UP

Node is already a cool platform, and Node+ClojureScript turned out to be a great match. You don't often have the ability to so easily pull together libraries from two disjoint ecosystems. The combination of Node's well-thought-out async APIs and core.async turned out to be particularly powerful, and the ability to add something as fundamental as pattern matching to a language by specifying a dependency is an incredible convenience. Under a tight deadline, we were able to build something sophisticated using simple parts. ClearCommit isn't where it could be, but it should give you an idea of what's possible.

For now, Node Knockout judging continues, so get diffing and consider voting for us!

Like this article? Subscribe to Kevin's mailing list.