DeloreanJS: Un Debugger en el Tiempo para JavaScript

Paul Leger
Escuela de Ingeniería
Universidad Católica del Norte
Coquimbo, Chile
pleger@ucn.cl

AAAA BBBB
Escuela de Ingeniería
Universidad Católica del Norte
Coquimbo, Chile
aaaa.bbbb@alumnos.ucn.cl

XXXX YYYY
Escuela de Ingeniería
Universidad Católica del Norte
Coquimbo, Chile
xxxx.yyyy@alumnos.ucn.cl

***Resumen*—Aplicaciones Web, usando JavaScript, son desarrolladas cada vez con mayor frecuencia. Como en la mayoría de los entornos de desarrollos, una aplicación Web puede adquirir defectos de software (conocido como *bugs*), cuyos síntomas se aprecian durante este desarrollo e incluso, siendo peor, en producción. Para ello, el uso de debuggers son sumamente útiles para detectar estos bugs. Lamentablemente, los actuales debuggers solamente avanzan hacia adelante en la ejecución para detectar un bug y no permiten retornar hacia un punto anterior en la ejecución para tomar acciones asociadas al bug detectado. Por ejemplo, probar si el mismo bug podría gatillarse con otro valor de una variable. Usando el concepto de continuaciones, este artículo presenta un debugger, llamado DeloreanJS, para JavaScript que permite devolverse en el tiempo con el fin que un programador pueda volver a verificar y probar el contexto alrededor de un bug. Este debugger ha sido implementado como una extensión para Google Chrome con el fin que la comunidad de desarrolladores pueda usarlo.**

***Index Terms*—DeloreanJS, JavaScript, Debugger, Web application, Continuations**

I. Introducción

Para construir aplicaciones Web, uno de los lenguajes más usados es JavaScript, cuya presencia en el entorno de la Web es de alrededor del 95 % [1]. La industria del software se mueve fuertemente hacia el desarrollo de este tipo de aplicaciones, siendo testigo un gran número de migraciones de aplicaciones *standalone* a la Web; los ejemplos van desde convertir un documento Word a un formato PDF [2] hasta un sistema ERP (*Enterprise Resource Planning*) [3]. Por ello, cada vez estas aplicaciones se han vuelto más complejas y con mayor riesgo de introducir defectos de software (*a.k.a. bugs*).

Detectar y reparar bugs representa una de las tareas más costosa en el proceso de desarrollo de software, y las aplicaciones Web no son la excepción. Para aliviar el proceso de tratamiento de bugs, un conjunto de debuggers se han propuestos, desde el que incluye un navegador en su distribución (*e.g.,* Mozilla Firefox developer tools) hasta debuggers omniscientes que pueden recorrer a través de la traza de ejecución para encontrar el origen de un bug [4], [5]. Considerando estos últimos debuggers, ellos son *post-morter*, y es decir, solo permiten mostrar la ocurrencia de un bug sin entregar la posibilidad de retroceder *en el tiempo* para que un programador pueda repararlo mientras se ejecuta o manipular el estado de las variables alrededor del bug para mejorar la comprensión de su causa.

Este artículo científico presenta DeloreanJS, un debugger que permite al desarrollador en una aplicación Web escrita en JavaScript fijar *timepoints* (en cambio de *breakpoints*) para entregar la posibilidad retornar en el tiempo a esos puntos y modificar valores de variables para continuar la ejecución del programa. Este novedoso enfoque de DeloreanJS permite:

1. Modificar los valores de las variables asociado a un bug encontrado para mejorar la comprensión de éste. Lo anterior puede ahorrar un gran número de ejecuciones usando un contexto similar para descubrir la verdadera razón del bug.
2. Probar escenarios hipotéticos de la ejecución de una aplicación Web usando un timepoint de DeloreanJS. Esto permite explorar diversas evoluciones de la ejecuación de la aplicación cambiando valores de algunas de sus variables.
3. Mantener una aplicación Web funcionando usando los timepoints para potencialmente reparar un bug durante la ejecución. Lo anterior significa que no es siempre necesario detener la ejecución de la aplicación con el fin de realizar un analísis post-morter, implicando perder el contexto de ejecución asociado al bug.

Para construir DeloreanJS, utilizamos la abstracción *continuación* [6] de los lenguajes de programación funcional como Scheme [7]. Una continuación permite al programador capturar y guardar, como valor del lenguaje, un momento en la ejecución (*program counter* y *stack*) de una aplicación. Extendiendo esta abstracción para lenguajes no funcionales como JavaScript, nosotros habilitamos a DeloreanJS para que ofrezca la posibilidad a un programador de expresar la insercción de timepoints a una aplicación Web, y así poder volver esos momentos de ejecución y modificarlos cuando es requerido.

El artículo está organizado como sigue. La sección II presenta diferentes ejemplos de aplicación de DeloreanJS, donde se puede apreciar la interfaz gráfica de nuestra propuesta. Luego, se presenta DeloreanJS, detallando cómo se usa el concepto de continuaciones. En la sección IV se discuten

herramientas similares. Finalmente, sección V concluye y describe lineamientos sobre el trabajo futuro de nuestra propuesta.

Disponibilidad. El código fuente de la implementación de DeloreanJS se encuentra en `http://github.com/fruizrob/delorean` y una prueba de concepto de su aplicación se puede obtener, como extensión de Google Chrome, en `http://xxx.com`.

II. Un Tour por DeloreanJS

PL *Cada ejemplo debe tener el nombre de lo que hace* □

II-A. Ejemplo 1

PL *Explicar con imagénes el ejemplo 1* □

II-B. Ejemplo 2

PL *Explicar con imagénes el ejemplo 2* □

II-C. Ejemplo 3

PL *Explicar con imagénes el ejemplo 3* □

III. DeloreanJS

Dado que nuestra propuesta trabaja con un enfoque relativamente diferente a los existentes debuggers, esta sección describe cómo funciona DeloreanJS y los conceptos necesarios para construirla.

La figura 1 muestra las multiples trazas de ejecución, llamadas *timelines*, que pueden ser producidas por los timepoints en la ejecución de una aplicación Web. Cuando un programador inserta un timepoint, se puede volver a ese timepoint durante la ejecución y así crear una nueva timeline. Cada timeline representa una traza de ejecución distinta y a su vez cada timeline puede generar otras timelines debido a los timepoints insertados. Como se puede apreciar, la abstracción de un timepoint es crucial porque captura y almacena la ejecución en términos del *program counter*, *stack*, y *heap* (Figura 2).

III-A. Continuaciones

This section briefly introduces continuations and then explains how Sync/cc uses them. Programming languages like Scheme provide continuations [6], which capture and store the current program control state as a first-class value. If this value is a function, it can be called and the current continuation will be replaced with the stored continuation. Unwinder [8] is a JavaScript library that captures the current continuation with a built-in function, named callCC. We illustrate continuations in Unwinder through a piece of code that captures the execution of a function adding two numbers (Listing 1):

```
1  var kont;
2
3  function add(x,y) {
4    return x + (function() {
5                  kont = callCC(cont => cont);
6                  return typeof(kont) == "number"? kont:y;})();
7  }
8
9  show(add(5,1));                              //show 6
10 if (typeof(kont) == "function") kont(20); //show 25
```

Listing 1. Use of continuations in the Unwinder library.

Figure 3 shows a flowchart of the piece of code above. In Line 5, the piece of code captures and stores the continuation in kont before adding y. This capture takes place in Line 9 when add is called. The result of add is passed to show, and then the number 6 is displayed. Line 10 executes the continuation bound to kont with 20 as parameter, which replaces the return of the anonymous function of Line 6 (*i.e.,* return 1 → return 20). As a result, 25 is displayed (25 = (x = 5) + (y = 20)). Note that the *if expression* (Line 6) and *if statement* (Line 10) are used to differentiate when a continuation is created from being called. A continuation is a function (Line 9) if this continuation has been created but not called, otherwise the continuation is bound to the value passed by parameter when it is called (*i.e.,* 20 in this piece of code).

If developers need to stop a handler execution, a continuation in the end of the top-level script should be created. For example, Listing 2:

```
function showMessageHandler() {
  show("This message will be shown");
  stop();
  show("This message will never be shown");
}

var stop = callCC(cont => cont);
```

Listing 2. Stopping a JavaScript script with continuations.

Sync/cc uses two continuations: the first one stops a handler execution just after an asynchronous operation invocation, and the second one resumes the handler execution when the asynchronous operation response is available.

III-B. Capturando el heap

PL *Explicar su forma de capturar estados* □

III-C. Extensión en el Navegador

PL *Explicar la aplicación en Google Chrome* □

IV. Trabajos Relacionados

Dado que JavaScript es un lenguaje muy ampliamente usado, existe un interés constante en crear avances en términos de investigación [9]–[13] y desarrollo [14]–[17]. En estos avances, varios tipos de propuestas de debuggers son posible encontrar en la literatura. Algunos de ellos [18]–[20] ofrecen un amplio conjunto de características como modificar los valores de variables mientras la aplicación se está ejecutando (ej. FireBug [18]). Aunque, en lo mejor de nuestro conocimiento, no es encontrar debuggers que sigan un enfoque similar a DeloreanJS, existen algunos que consideran la historia de ejecución de una aplicación Web:

Debuggers omniscientes. Estos tipos de debuggers se encargan de almacenar cada evento que ocurre en la ejecución de un programa, creando un historial de la traza de ejecución. Estos debuggers han sido implementados en lenguajes como en Java [21] y xDSMLs [22]. Con respecto a JavaScript, podemos encontrar PECCit [4] y JARDIS [5], los cuales almacenan la traza de ejecución y ofrecen diferentes interfaces de usuario para navegar a través de esta traza. A diferencia de DeloreanJS,

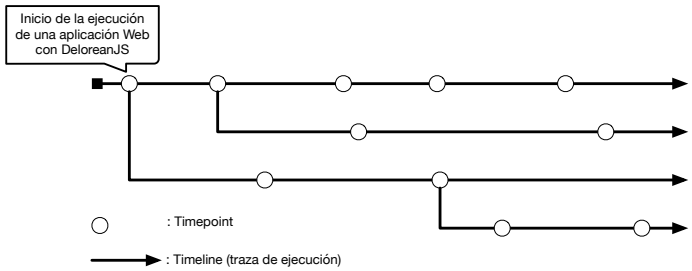

Figura 1. Multiples trazas de ejecuciones de a través de timepoints de DeloreanJS.

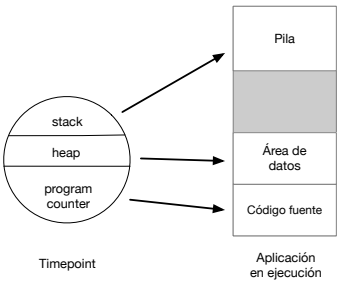

Figura 2. Composición de un timepoint.

estos tipos de debuggers son *post-morter*, significando que no es posible volver a un punto en la historia de esta ejecución y continuarla con valores de variables potencialmente modificados.

Debuggers remotos. JavaScript es generalmente[1] usado para construir aplicaciones Web que son usadas por usuarios con algún dispositivo que pertenece a un muy variado y amplio catalogo, pues lo único que se necesita es un navegador con acceso a internet. Por esta razón, cualquier configuración del dispositivo puede producir un potencial bug que los desarrolladores no podrían observar en un ambiente de desarrollo controlado. Para abordar esta dificultad, existen debuggers [25]–[27] que supervisan remotamente las ejecuciones de los usuarios. Similarmente a los debuggers omniscientes, estos registran y envían cada punto ejecución a un desarrollador por la red. Aunque estos debuggers permiten a los desarrolladores trazas de ejecuciones de diferentes usuarios en tiempo real, no ofrecen la posibilidad de retroceder en momentos de la ejecución a los usuarios.

V. Conclusiones

No es un misterio que la industria del software se esté orientando a construir aplicaciones Web cada vez más grandes y complejas, implicando que exista una mayor probabilidad que aparezcan bugs. Para construir estas aplicaciones, el lenguaje JavaScript es ampliamente usado y una amplia cantidad de debuggers se encuentra disponible en la Web [4], [5], [18]–[20], [25]–[27]. Sin embargo, a diferencia de DeloreanJS, ninguno de estos debuggers considera el enfoque de *retroceder en el tiempo* a momento en la ejeucución una aplicación escrita en JavaScript. Sin este enfoque, hay un conjunto de oportunidades que un programador pierde, por ejemplo, probar distintos valores de variables en un mismo contexto de ejecución con el objetivo de mejorar la comprensión del bug o explorar escenarios hipotéticos de la evolución de la ejecución dado un mismo contexto inicial. Para alcanzar una adopción por parte de la comunidad, DeloreanJS tiene aún algunos desafíos:

Guardar el heap. A diferencia del stack que se guarda automáticamente en una aplicación en ejecución, un programador debe ahora especificar qué variables del heap se guardarán en los timepoints. Resolver este desafío implica almacenar copias de las variables contenidas en el heap por cada timepoint.

Integrar a un existente debugger. Actualmente DeloreanJS ofrece el nuevo enfoque de retroceder en el tiempo de una ejecución. Integrar este enfoque con las características de los existentes debuggers podría significativamente la adopción de nuestra propuesta. Por ejemplo, debuggers omniscientes integrado con DeloreanJS permitiría analizar multiple trazas de ejecuciones partiendo desde un timepoint.

Como limitación podemos mencionar que DeloreanJS realiza la promesa de retornar en el tiempo de una ejecución, la cual podría no cumplirse a totalidad si aplicación depende de los resultados de eventos externos (ej. servicio Web que entrega la hora actual). Sin embargo, esta limitación se encuentra también presenta en los existentes debuggers.

[1]Aparte de la Web, JavaScript actualmente es usado en varios entornos de desarrollos, por ejemplo, es usado en el lado del servidor en una aplicación Web [23] y en administradores de ventas de sistemas operativos basados en Linux [24].

Referencias

[1] "Usage of client-side programming languages," https://w3techs.com/technologies/history_overview/client_side_language/all, accessed: 2019-03-14.

[2] "Smallpdf: Word to pdf," 2019. [Online]. Available: https://smallpdf.com/word-to-pdf

[3] "Oracle erp cloud," 2019. [Online]. Available: http://www.oracle.com/ERP

[4] Z. Azar, "Peccit: An omniscient debugger for web development," Master's thesis, University of Denver, Denver, United States, 2016.

[5] E. Barr, M. Marron, E. Maurer, D. Moseley, , and G. Seth, "Time-travel debugging for javascript/node.js," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, WA, USA, Nov. 2016, pp. 1003–1007.

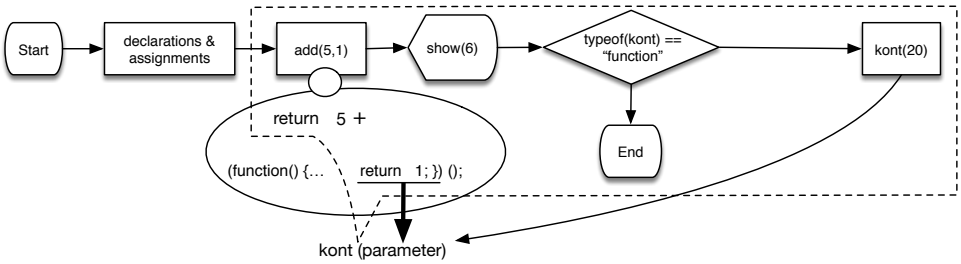

Figura 3. The flowchart of Listing 1, which uses the continuation kont. The invocation of kont uses a parameter to replace the return value of the anonymous function inside of add (*i.e.,* "return 1").

[6] D. P. Friedman and M. Wand, "Reification: Reflection without metaphysics," in *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, Aug. 1984, pp. 348–355.

[7] R. A. Kesley and J. A. Rees, "A tractable Scheme implementation," *Lisp and Symbolic Computation*, vol. 7, no. 4, pp. 315–335, 1995.

[8] J. Long, "Unwinder: A call/cc library," 2018. [Online]. Available: https://github.com/jlongster/unwinder

[9] H. Vázquez, A. Bergel, S. Vidal, A. Díaz, and C. Marcos, "Slimming javascript applications: An approach for removing unused functions from javascript libraries," *Information and Software Technology*, vol. 107, pp. 18–29, 2019.

[10] P. Leger, É. Tanter, and R. Douence, "Modular and flexible causality control on the web," *Science of Computer Programming*, vol. 78, no. 9, pp. 1538–1558, Sep. 2013. [Online]. Available: http://pleiad.cl/weca

[11] P. Leger, E. Tanter, and H. Fukuda, "An expressive stateful aspect language," *Science of Computer Programming*, vol. 102, no. 0, pp. 108–141, May 2015.

[12] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 11th International World Wide Web Conference (WWW 2011)*. Hyderabad, India: ACM Press, Mar. 2011, pp. 805–814.

[13] N. ten Veen, D. Harkes, and E. Visser, "JSExplain: A double debugger for javascript," in *Proceedings of the 2018 Web Conference Companion (WWW 2018)*. Lyon, France: ACM Press, Apr. 2018, pp. 691–699.

[14] jQuery Foundation, "jQuery: A JavaScript library to manage event handling, animating, and Ajax interactions for the Web development. http://jquery.com/." [Online]. Available: http://jquery.com/

[15] "Angular: A framework to build web applications," 2019. [Online]. Available: https://angular.io/

[16] S. McKenzie, "Babel: A compiler for writing ES6 and ES7 generation JavaScript," 2019. [Online]. Available: https://babeljs.io/

[17] "RxJS: Reactive extensions for javascript," 2018. [Online]. Available: https://rxjs.dev

[18] J. Barton and J. Odvarko, "Dynamic and graphical web page breakpoints," in *Proceedings of the 11th International World Wide Web Conference (WWW 2011)*. Hyderabad, India: ACM Press, Mar. 2011, pp. 81–90.

[19] "JsBin: A open source framework to develop and debug javascript applications," 2019. [Online]. Available: https://jsbin.com/

[20] "NodeJS-Inspector: A open source framework to develop and debug javascript applications," 2019. [Online]. Available: https://jsbin.com/

[21] G. Pothier, É. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, Montreal, Canada, Oct. 2007, pp. 535–552.

[22] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xdsmls," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, Pittsburgh, PA, USA, Oct. 2015, pp. 137–148.

[23] N. Foundation, "Node.js: is a javascript runtime built for sever side," 2018. [Online]. Available: https://nodejs.org

[24] "GJS: Javascript bindings for gnome," Mar. 2019. [Online]. Available: https://gitlab.gnome.org/GNOME/gjs/wikis/Home

[25] P. Vallet, "SessionStack: An online monitor of javascript applications." [Online]. Available: https://www.sessionstack.com/

[26] B. Wark, "Raygun: Real user monitoring remotely." [Online]. Available: https://raygun.com/

[27] T. LLC, "TrackJS: A JavaScript tracking error monitoring. https://trackjs.com/." [Online]. Available: https://trackjs.com/