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 el desarrollo e incluso, siendo peor, en producción. Para ello, el uso de debuggers es sumamente útil para detectar 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 para JavaScript, llamado DeloreanJS, que permite a un programador volver en el tiempo en la ejecución con el fin pueda volver y probar el contexto alrededor de un bug. Este debugger ha sido implementado como una prueba de concepto en una extensión para Google Chrome.**

***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]. Asimismo, 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 online 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 han sido 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 de una aplicación para encontrar las causas 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* las veces necesarias para que un programador pueda repararlo mientras la aplicación se ejecuta o manipular los valores de las variables alrededor del bug para mejorar la comprensión de su causa.

Este artículo científico presenta DeloreanJS, un debugger que permite al desarrollador de 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 reiteradamente 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 ejecución 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.

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 (*contador de programa* y *pila*) 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 en 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 y extiende continuaciones, y la integración con un navegador. En la sección IV se discuten debuggers 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 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 diferente a los existentes debuggers, esta sección describe cómo funciona DeloreanJS.

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 en la figura 2, la abstracción de un timepoint es crucial porque captura y almacena la ejecución en términos del *program counter* (contador de programa), *stack* (pila), y *heap* (datos). El program counter y stack es capturado usando continuaciones [6] y el heap es capturado a través de un análisis estático usando Babel [8].

III-A. Usando Continuaciones

Esta sección brevemente introduce continuaciones y explica cómo son usadas por DeloreanJS. Los lenguajes de programación funcionales como Scheme [7], los cuales se caracterizan por no tener variables *mutables*, proveen la abstracción de una continuación. Esta abstración captura el estado actual (program counter y stack) de un programa funcional y lo guarda como un *valor de primera clase*, es decir, un valor que soporta operaciones como asignación e invocación (ej. funciones en JavaScript). Cuando una continuación es creada y guardada, ésta puede ser llamada y reemplazará la actual ejecución por la guardada continuación. Unwinder [9] es una librería en JavaScript soporta continuación a través de una función llamada callCC. Nosotros ejemplificamos continuaciones con Unwinder con pieza de código que captura la ejecución de una función que suma dos números (pieza de código 1):

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

Listing 1. Uso de continuaciones con la librería Unwinder.

La figura 3 muestra el diagrama de flujo de la pieza de código 1. En la línea 5, una continuación kont es creada antes de sumar la variable y. Esta captura toma lugar en la línea 9 cuando la función sumar es llamada. El resultado de sumar es pasado a sumar, y entonces el número 6 es mostrado. La línea 10 invoca la continuación asociada a kont con el parámetro 20, produciendo que el valor retornado de la función anónima de la línea 6 (es decir, return 1 → return 20). Como resultado, se muestra 25 (25 = (x = 5) + (y = 20)). Notar que el *if expression* (línea 6) y *if statement* (línea 10) son usadas para diferenciar cuando una continuación es creada y llamada. Una continuación es una función (línea 9) es creada pero no llamada, en otro caso la continuación es enlazada al valor pasada como parámetro cuando es llamada (ej. el valor 20 en esta pieza de código).

***************HERE***************

III-B. Capturando el Heap

PL *Explicar su forma de capturar estados* □

III-C. Integrando en un 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 [10]–[14] y desarrollo [8], [15]–[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, estos tipos de debuggers son *post-morter*, significando que no es posible volver a un punto en la historia de esta ejecución

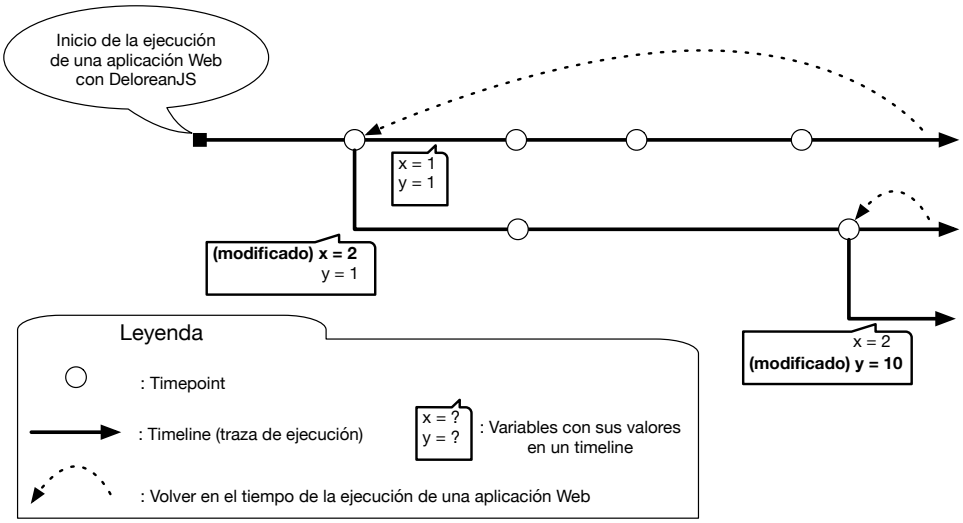

Figura 1. Multiples trazas de ejecuciones de la ejecución de una aplicación con DeloreanJS.

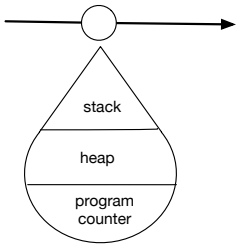

Figura 2. Composición de un timepoint.

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.

[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].

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:

Capturar completamente el heap. A diferencia del stack que se captura 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

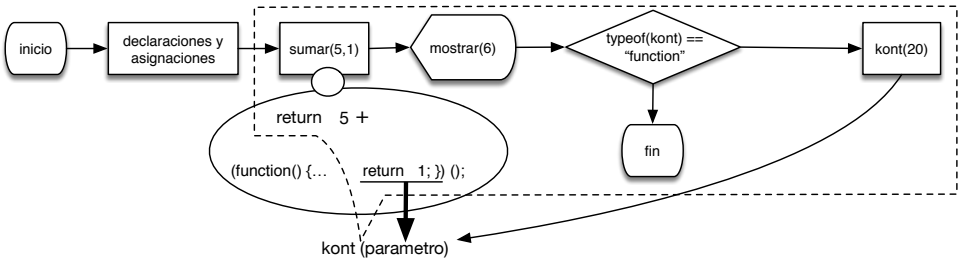

Figura 3. El diagrama de flujo de la pieza código 1, el cual usa una continuación kont. La invocación de kont usa un parametro para reemplazar el retornado valor de la función anónima dentro de sumar.

la hora actual). Sin embargo, esta limitación se encuentra también presenta en los existentes debuggers.

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.

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

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

[10] 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.

[11] 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

[12] 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.

[13] 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.

[14] 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.

[15] 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/

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