ARTÍCULO SELECCIONADO DEL CONAIISI

Plataforma de meta-programación para Javascript

Alexis Ferreyra[1]
Néstor Navarro[2]
Ricardo Medel[3]
Emanuel Ravera[4]

[1]E-mail: alexis.ferreyra@gmail.com
[2]E-mail: nestornav@gmail.com
[3]E-mail: ricardo.h.medel@gmail.com
[4]E-mail: ravera.emanuel@gmail.com

Universidad Tecnológica Nacional
Facultad Regional Córdoba

RESUMEN

El auge de los dispositivos móviles y la expansión de la web han permitido que JavaScript se convierta en uno de los lenguajes de mayor crecimiento en los últimos años. Aunque se ha mejorado constantemente el lenguaje y se han desarrollado herramientas para aumentar su productividad, eficiencia y seguridad, en este trabajo presentamos un enfoque que permite una aplicación rápida a ideas de investigación y desarrollo. A fin de utilizar la meta-programación, que permite la reescritura e introspección de código, desarrollamos la plataforma PumaScript. En este artículo explicamos cómo está construida dicha plataforma y mostramos su aplicación a la resolución de problemas de eficiencia y seguridad, dos aspectos claves en las aplicaciones web.

Palabras clave: JavaScript, meta-programación, reescritura de código.

INTRODUCCIÓN

Debido al crecimiento de la web y sus aplicaciones, durante estos últimos años se ha incrementado la importancia del lenguaje JavaScript manifestándose en múltiples avances en los runtimes [1, 2, 3, 4, 5] y extensiones propuestas para mejorar la performance [6, 7, 8, 9]. Sin embargo, a pesar de la constante experimentación e innovación en el lenguaje, no existe una plataforma para JavaScript que permita a los investigadores y desarrolladores experimentar rápidamente con nuevos conceptos.

Aunque existen esfuerzos dispersos como dialectos con macros programables [10], parsers y generadores de código [11, 12], ninguno de estos proyectos incorpora una infraestructura genérica y flexible de meta-programación para JavaScript. La ventaja de un entorno flexible de meta-programación es que permite a investigadores y desarrolladores

la rápida experimentación de conceptos sin requerir el gran esfuerzo que significa construir una variante de un lenguaje y su runtime por completo. Si bien se está trabajando en el siguiente estándar del lenguaje [13] en el mismo no se planean incorporar características de meta-programación, tales como introspección y reescritura de código.

Es por ello que nuestro grupo desarrolló PumaScript, una plataforma genérica de meta-programación que extiende JavaScript con capacidades de introspección de código fuente y reescritura. En el presente trabajo demostramos el diseño e implementación de la plataforma en la sección DISEÑO E IMPLEMENTACIÓN DE PUMASCRIPT y luego mostramos su utilidad a través de su aplicación a dos aspectos críticos del software en general y de las aplicaciones web en particular: su eficiencia y seguridad.

En la sección APLICACIONES DE LA PLATAFORMA se aplica PumaScript a la detección de patrones de código de baja eficiencia y, a través de la reescritura de código, se hace al software más eficiente. Asimismo, se aplica la misma técnica para identificar el uso de APIs inseguras y reescribir las porciones de código vulnerables con código de similar comportamiento pero más seguro.

En la sección TRABAJOS RELACIONADOS describimos y analizamos trabajos de otros grupos con objetivos similares pero que utilizan técnicas diferentes. Para finalizar, en la sección CONCLUSIONES Y DIRECCIONES FUTURAS elaboramos las conclusiones de nuestro trabajo y el trabajo que nos proponemos realizar como continuación de éste.

DISEÑO E IMPLEMENTACIÓN DE PUMASCRIPT

A fin de explorar la aplicación de técnicas de meta-programación e introspección de código en JavaScript desarrollamos PumaScript, un superconjunto del lenguaje y una plataforma asociada que automatiza el proceso de reescritura de código. PumaScript se caracteriza por la posibilidad de definir meta-funciones que permiten manipular el Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés) para analizar y mejorar el código fuente.

Las denominadas meta-funciones trabajan de manera muy similar a un sistema de macros para lenguajes tipados. En particular, pueden desagregar expresiones de la función llamadora en la misma línea, lo que implica que se puede reescribir el árbol que retorna una meta-función en la misma línea en que se la invoca. Una meta-función toma como parámetro el árbol de sintaxis decorado de los argumentos actuales y retorna un árbol de sintaxis abstracto que se usa como reemplazo en la función llamadora. Debido a que no siempre es conveniente reemplazar el código original, las meta-funciones de PumaScript pueden decidir no expandir cierta ocurrencia de la función llamadora retornando "*null*" en lugar del árbol de sintaxis abstracta.

Una meta-función, además, puede ejecutar otras funciones normales o más meta-funciones. Adicionalmente, las meta-funciones tienen acceso a funciones especiales intrínsecas provistas por la plataforma PumaScript que les permiten realizar la introspección y la reescritura de código de cualquier porción del árbol de sintaxis abstracta.

La Figura 1 muestra un programa de ejemplo muy sencillo incluyendo la declaración de una meta-función y su invocación. El propósito de la meta-función *"helloWorld"* es reescribir la expresión llamadora retornando una operación de suma con los argumentos invertidos.

```
1  /* @meta */
2  function helloWorld(a, b){
3    return pumaAst($b + $a);
4  }
5
6  helloWorld("World", "Hello");
```

Figura 1. Programa PumaScript de ejemplo con una declaración de meta-función y su invocación.

La Figura 2 muestra el resultado de ejecutar el programa donde se puede apreciar cómo el programa se re-escribió en la concatenación de las dos, pero invertidas, provistas como

argumentos al invocar la meta-función en la última instrucción del programa.

```
1  'Hello' + 'World';
```

Figura 2. Resultado de aplicar la metafunción helloWorld.

La meta-función del ejemplo es muy simple, pero nos permitirá introducir el funcionamiento general de PumaScript. El runtime de PumaScript procesa el programa de entrada de la misma forma que JavaScript estándar. El texto del programa de entrada es parseado y convertido en una estructura de Árbol de Sintaxis Abstracta (AST). Luego cada nodo del AST es interpretado en el orden en el cual se presenta en el texto del programa original siguiendo las reglas de JavaScript. Por ejemplo, identificando las declaraciones en cada contexto, antes de ejecutar el nuevo contexto como el contexto global del programa o una nueva función invocada. PumaScript se diferencia de JavaScript, fundamentalmente, en la incorporación de meta-funciones y la interacción de estas con el resto de la semántica estándar de JavaScript.

En el ejemplo de la Figura 1 se declara una meta-función *"helloWorld"*, la cual toma como argumento dos expresiones. La línea 6 del programa invoca la meta-función, la cual pasa como argumento dos AST triviales con los nodos literales *"World"* y *"Hello"* respectivamente. La línea 3 de la meta-función realiza dos operaciones, en primer lugar, crea un nuevo AST conteniendo una expresión binaria de suma con los AST provistos como argumentos en cada operador, finalmente retorna dicho AST como resultado de invocar la meta-función. Al terminar de invocar la meta-función, el runtime de PumaScript reescribe la línea 6 del programa en la Figura 1 por el AST retornado por la función *"helloWorld"*. Al finalizar la ejecución de todas las instrucciones del programa, PumaScript genera como salida el programa reescrito de la Figura 2, donde se puede apreciar que la meta-función fue eliminada del programa de entrada.

En la siguiente sección se explica en más detalle las pasadas de ejecución de PumaScript.

PASOS DE EJECUCIÓN DE PUMASCRIPT

El flujo de ejecución de un programa en nuestra plataforma consta de cuatro pasos, tomando como entrada un programa PumaScript (es decir, el programa JavaScript original que se quiere analizar y mejorar, aumentado con meta-funciones) y devolviendo como salida un programa en el lenguaje JavaScript estándar, con el mismo comportamiento que el original pero con las mejoras implementadas. Los cuatro pasos se muestran gráficamente en la Figura 3 y se describen a continuación.

1. La librería Esprima [12] realiza el parsing de un programa escrito en PumaScript y obtiene como resultado su árbol de sintaxis abstracta (AST).
2. El runtime ejecuta el AST siguiendo la semántica de PumaScript, esto es, la semántica JavaScript aumentada con meta-funciones. Durante la ejecución, además, se tiene acceso a funciones intrínsecas, las que permiten realizar operaciones sobre el AST, tales como inspeccionar, cambiar, agregar o remover nodos.
3. Una vez ejecutado el programa, se eliminan las declaraciones de meta-funciones y se obtiene un AST compatible con el lenguaje JavaScript estándar.
4. El árbol de sintaxis (AST) resultante es procesado por la librería Escodegen [11] para obtener un programa escrito en lenguaje JavaScript estándar como salida.

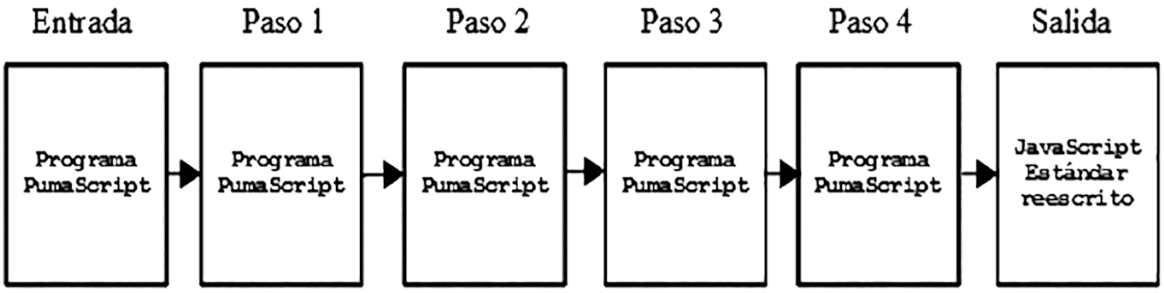

Figura 3. Flujo de procesamiento de un programa PumaScript.

META-FUNCIONES DE PUMASCRIPT

Las meta-funciones de PumaScript que se escribieron para un programa juegan un papel clave durante el proceso de su ejecución. Dado que la sintaxis de estas meta-funciones utiliza el lenguaje JavaScript, agregando el comentario *@meta* antes de su declaración, no se introducen nuevas estructuras sintácticas, lo que permite que PumaScript sea completamente compatible con JavaScript. Las principales diferencias con las funciones de JavaScript se enumeran a continuación.

a) Todos los parámetros en una meta-función se evalúan en referencia al árbol de sintaxis decorado al momento de la ejecución. Por ejemplo, si la meta-función *"foo(a, b)"* es invocada con la expresión *"foo(2 * x, 3 * z)"*, al momento de la ejecución los parámetros a y b tomarán los valores del árbol de sintaxis que corresponde a las expresiones pasadas como argumentos: *"2 * x" y "3 * z"*, respectivamente.

b) Todas las meta-funciones deben retornar un árbol de sintaxis válido o el valor *"null"*. Si se retorna el valor *"null"* entonces la función llamadora no será reescrita. En cambio, si el valor retornado es un árbol de sintaxis válido entonces la expresión de la función llamadora será reemplazada por éste.

c) Todas las meta-funciones tienen acceso a funciones y objetos intrínsecos que pueden ser utilizados para realizar la introspección de código o la reescritura de cualquier porción del programa. Algunos ejemplos de éstas son *"pumaAst"*, la cual genera un árbol de sintaxis correcto, y *"pumaFindByType"*, que permite realizar búsquedas dentro del árbol de sintaxis del programa en ejecución.

A fin de comprender mejor el concepto y aplicación de las meta-funciones de PumaScript, las Figuras 4 y 5 muestran ejemplos más complejos que el presentado previamente.

En la Figura 4 se describe la definición y uso de la meta-función *"firstLetter"*, que re-escribe a la función llamadora sólo si el argumento es un literal. En caso contrario, retorna el valor *"null"* y evita la reescritura de la función llamadora. A partir de la línea 20 se muestra el resultado de la ejecución del programa. En la línea 21 se muestra el resultado de la llamada realizada en la línea 14. En este caso se retorna la letra "H", la primera letra de la cadena pasada como argumento, ya que se invocó a la meta-función *"firstLetter"* con el literal "Hola Mundo". Por otra parte, en la línea 22 se verifica que la invocación de la función *"firstLetter"* en la línea 18 no activa su reescritura, ya que el argumento no es un literal, sino una concatenación de dos cadenas.

```
1  // Programa de PumaScript
2
3  /* @meta */
4  function firstLetter (valueExp){
5    var ast = null;
6    if(valueExp.type === "Literal"){
7       ast = valueExp;
8       ast.value = ast.value.substring(0 , 1) ;
9    }
10   return ast;
11 }
12
13 // Esta llamada se va a reescribir como "H";
14 firstLetter("Hola Mundo") ;
15
16 // Esta llamada no se va a reescribir
17 // Debido a que la expresion no es un literal
18 firstLetter("Hello" + "World");
19
20 // Salida después de la ejecución
21 "H" ;
22 firstLetter ("Hello" + "World");
```

Figura 4. Ejemplo de meta-función firstLetter.

Una meta-función de PumaScript que cuenta todas las ocurrencias de las sentencias for y escribe el resultado utilizando el objeto intrínseco de JavaScript console se muestra en la Figura 5. En este ejemplo se utiliza la función intrínseca pumaFindByType y el objeto intrínseco pumaProgram. Este objeto es la representación de todo el programa, mientras que la función pumaFindByType permite hacer consultas sobre todos los nodos de dicha representación del programa.

```
1  /** @meta */
2  function countForStatemets () {
3    var forStas = pumaFindByType(pumaProgram, "ForStatement");
4    console.log("For statements found : " + forStas.length);
5    return null;
6  }
```

Figura 5. Contador de ocurrencias de llamadas a la sentencia for.

APLICACIONES DE LA PLATAFORMA

El objetivo de la plataforma PumaScript es permitir a investigadores y desarrolladores del lenguaje explorar fácilmente distintas posibilidades de mejora o líneas de investigación en forma ágil y sin mayor esfuerzo. En esta sección describimos la aplicación de esta plataforma a dos temas muy relevantes para el desarrollo de aplicaciones web: eficiencia y seguridad.

El enfoque utilizado es descubrir patrones de código con baja performance o que contenga vulnerabilidades de seguridad y escribir meta-funciones que permitan reemplazar el código original con nuevo código más eficiente o más seguro, según sea el caso.

MEJORA DE EFICIENCIA A TRAVÉS DE REESCRITURA DE PATRONES LENTOS

Como parte de nuestro proyecto de I+D, se buscaron patrones de código que sea de uso común por los desarrolladores al escribir aplicaciones, pero de conocida baja eficiencia en tiempo de ejecución. Se hizo foco en el uso ineficiente de las API (Application Programming Interface) nativas que brinda el runtime del navegador y la popular librería jQuery [14]. Luego, para cada patrón de baja eficiencia, buscamos una implementación que conserve su semántica pero sea más eficiente.

La Tabla 1 muestra ejemplos de dos de los patrones identificados. La primera y segunda columnas indican simplemente un número y nombre identificatorio de cada ejemplo, respectivamente. La tercera columna muestra un ejemplo del código original, mientras que la cuarta columna muestra el resultante de código más eficiente y con igual semántica.

Tabla 1. Patrones de código ineficiente en JavaScript.

	Nombre	Código Original	Código Optimizado
1	querySelectorAll vs. getElementByClassName	var items = document.querySelectorAll(“.test”);	var items = document.getElementsByClassName(“test”);
2	QuerySelectorAll vs. GetElementByTagName	Var items = document.querySelectorAll(“test”);	Var items = getElementsByTagName(“test”);

En el primer patrón se propone reemplazar el uso de la API *querySelectorAll* por *getElementsByClassName*, mientras que en el segundo caso, ante su invocación con un argumento diferente, la misma API se reemplaza por *getElementsByTagName*. Una importante ventaja de PumaScript es que para resolver ambos casos sólo es necesario definir una única meta-función que se comportará de una u otra manera según el tipo de argumento con que sea invocada.

A fin de realizar experimentos con estos y otros patrones, desarrollamos una herramienta de ejecución de benchmarks que nos permitió ejecutar segmentos de código un gran número de veces, pudiendo así medir valores confiables. En las Tablas 2 y 3 se muestran los resultados de mejora de performance al ejecutar 10.000 repeticiones, comparando la versión original de código con de la optimizada, sobre diferentes configuraciones de hardware y software.

Tabla 2. Resultados de ejecuciones de código sobre diferentes configuraciones de PC.

	PC – Crome – v. 36.0.1985.143	PC – Mozilla – v30.0
1	138,88x	364,27x
2	236,16x	393,29x

Tabla 3. Resultados de ejecuciones de código sobre diferentes configuraciones de tablets.

	Tablet Android – Browser Nativo – 4.4.2	Tablet Android – Chrome – 4.4.2 – v. 36.0.1985.135
1	915,23x	394,26x
2	213,69x	146,89x

La Tabla 2 muestra la mejora en los tiempos de ejecución del código optimizado ante el código original al ejecutarlos sobre una computadora personal (PC) con procesador Intel Core-i5-3320M y 4GB de RAM. Cada columna muestra la mejora en eficiencia al utilizar el código optimizado con distintos runtimes: Chrome y Mozilla. Por ejemplo, la celda (1,1) de la Tabla 2 indica que para el caso 1 de la Tabla 1 ejecutar el Código Optimizado en una PC utilizando Chrome es 138,88 veces más rápido que el Código Original ejecutando en la misma configuración.

Por su parte, la Tabla 3 muestra un experimento similar al anterior pero ejecutado sobre una tablet con procesador Intel Atom Z2560 dual-core de 1.6GHz y 2GB RAM.

Como puede verse, en todos los casos la ganancia en eficiencia de performance es realmente notable.

EVITANDO VULNERABILIDADES A TRAVÉS DE LA REESCRITURA DE APIS INSEGURAS

Un problema de seguridad habitual al que se enfrentan las aplicaciones web es el conocido como mXSS (Mutation-based Cross-Site-Scripting) [15]. Este tipo de ataque comienza con la preparación de código HTML malintencionado por parte de un atacante, quien luego lo inyecta en la aplicación web como una cadena de caracteres, la cual no es detectada como una amenaza, ni del lado del servidor ni en el navegador. Dentro de los posibles ataques generados por mXSS, nos concentramos en aquellos que son perpetrados a causa de la vulnerabilidad generada por la propiedad *innerHTML* y la función *eval*, propias del lenguaje JavaScript.

EVITANDO LA VULNERABILIDAD DE innerHTML

Dentro del runtime de JavaScript se encuentra definida a la propiedad *innerHTML* que nos permite crear código HTML a partir de una cadena de caracteres que es pasada como parámetro, obteniéndose como resultado el cambio del estado actual de los nodos del DOM (Document Object Model). Para darle un mayor contexto el Document Object Model provee una interfaz donde se define la estructura lógica de un documento HTML válido, así como los mecanismos que se debe implementar para ser accedido y/o manipulado.

Es por todo esto que cuando se recibe el parámetro en la propiedad *innerHTML* se desencadena un cambio en aplicación web en general insertando nuevos nodos al DOM.

Si bien dicha propiedad no se encuentra definida por ningún estándar, todos los navegadores modernos la soportan. Se puede utilizar esta propiedad con fines maliciosos porque tiene permisos de escritura para modificar el DOM actual y luego permiso de lectura a fin de interpretar los nodos modificados del DOM.

PumaScript permite la detección automática de la presencia de la propiedad *innerHTML* en

el código JavaScript original. Posteriormente proporciona la posibilidad de reescritura, implementando su uso con una forma análoga basada en funciones propias del DOM (tales como *createElement* y *appendChild*) que brinda una mayor seguridad. Mediante el uso de meta-funciones creadas para este fin, PumaScript analiza cada nodo del DOM externo que intenta agregar en el código fuente de la página actual. Luego realiza una evaluación basándose a una lista negra de acciones que no le son permitidas, lo que le permite decidir si agregar o no dicho nodo al DOM.

Estas acciones pueden ser configurables por el usuario en el caso que veremos a continuación podría ser la acción *"onmouseover"* en donde se sanitizaría el código y se evitaría la inyección de código malicioso.

Ejemplo: Casos no seguros de una implementación con innerHTML

Este es un caso típico de uso de la propiedad innerHtml, en la que se realiza una asignación directa de código externo por medio de un string. Podemos verlo en la Figura 6.

```
1  var contenedorInstancia = document.getElementById("contenedor");
2  var códigoExterno = document.getElementById("stringExternoConHTML").value;
3  contenedorInstancia .innerHTML = códigoExterno;
```

Figura 6. Asignación de string externo con HTML.

En el ejemplo *stringExternoConHTML* es una caja de texto en la cual se ha introducido código malicioso por un usuario, y *contenedorInstancia* una etiqueta HTML del tipo *div*. Como se puede apreciar, cualquiera sea el valor de *codigoExterno*, éste será ingresado como código HTML al elemento *contenedorInstancia*.

Continuando con el ejemplo, si un usuario externo introdujera el texto de la Figura 7 en la caja de texto.

```
1  <h3 onmouseover="xssFunction()">Soy un titulo<h3>
```

Figura 7. Ejemplo de inyección.

Nos encontraríamos en presencia de un hueco de seguridad importante y que le permite a un usuario aprovecharse de la vulnerabilidad e inyectar código malicioso desde una fuente externa en la aplicación web.

REEMPLAZANDO LA FUNCIÓN *eval*

La función *eval* recibe como parámetro un Sting. Si dicho parámetro representa una expresión, *eval* tratará de realizar la evaluación y en caso de que represente una o más instrucciones válidas, las ejecutará. Si bien evaluar código en tiempo de ejecución es un proceso lento, ya que implica compilar y ejecutar código en el *runtime*, la función *eval* es muy utilizada dado su flexibilidad y potencia. Sin embargo el uso de esta función puede presentar una vulnerabilidad importante para la seguridad del programa [16].

Para comprender en detalle, los potenciales problemas que puede presentar un mal uso de la función eval, hemos realizado una caracterización de tres de los casos más comunes de uso de esta función en los cuales pueden presentarse vulnerabilidades, y realizamos una propuesta de solución utilizando nuestro enfoque de reescritura de código a través de nuestra plataforma PumaScript. A continuación describimos los tres casos mencionados.

1. **Evaluar una cadena del tipo JSON:** En este caso se evalúa una cadena JSON para obtener un objeto puro. Por ejemplo, en la invocación *var objectJson = eval('({' + stringJSON + '})')* el problema de seguridad radica en que el argumento *stringJSON* podría ser un *request* a algún servidor, dando la oportunidad de inyección de código malicioso en la variable. Para solucionar este problema se propone transformar esa llamada a la función *eval* en una llamada a la función *parse* del objeto JSON, la cual descarta el argumento si éste no es un string con formato JSON. La expresión del ejemplo, luego de la reescritura

quedaría de la siguiente forma: *var objectJson = JSON.parse('{' + stringJSON + '}')*.

2. **Acceder dinámicamente a propiedades:** Este caso es uno de los más reportados, probablemente por desconocimiento de las características básicas del lenguaje, ya que el uso de la sentencia *eval* suele ser innecesaria. Ejemplo: *var objectProperty = eval('info.' + getProperty())*. La solución para este caso sería la reescritura hacia la forma: *var objectProperty = info[getProperty()]*.

3. **Variable global:** En el caso de utilizar *eval('var x=' + userString + ';')* se está declarando a la variable *x* en forma global para todo el documento JavaScript. Para evitarlo, la transformación pasa por establecer que el código debe ser ejecutado en "modo estricto", de modo que no pueda usar variables no declaradas. Para lo cual PumaScript transformaría el código original en *eval('"use strict";' + ('var x=' + userString + ';'))*. Esta transformación limita el ámbito de la variable *x* al contexto de la función *eval* únicamente, con lo que dicha declaración ya no sería global. De esta manera evitamos que se redefinan valores globales que pueden afectar la ejecución interna del programa.

TRABAJOS RELACIONADOS

Existen trabajos previos que utilizan enfoques similares al nuestro. En esta sección comparamos PumaScript con las herramientas de reescritura de código y lenguajes de pre-procesamiento producidas como resultado de los mismos.

PLATAFORMAS GENÉRICAS DE REESCRITURA DE CÓDIGO

Stratego XT [17] y DMC [18] son dos de las plataformas más desarrolladas para construir herramientas de transformación *source-to-source* utilizando técnicas de reescritura de código. Su principal ventaja es la flexibilidad, ya que ambas plataformas permiten construir herramientas para cada uno de los pasos de la cross-compilación *source-to-source*, tales como constructores de parsers, constructores de analizadores semánticos y *pretty printers* para diferentes lenguajes. Sin embargo, nuestro enfoque, tal como fue implementado en PumaScript, es más simple ya que no apuntamos a proveer herramientas genéricas adaptables a cualquier lenguaje de programación, sino que nos enfocamos sólo en la reescritura de JavaScript. Esto nos permite proveer un *stack end-to-end* para analizar y procesar JavaScript listo para ser utilizado por desarrolladores o investigadores. Por el contrario, si un investigador decidiera utilizar herramientas genéricas como las mencionadas a fin de procesar JavaScript debería tomarse el tiempo de construir todas las partes por sí mismo, incluyendo el *parser*, *runtime* y *pretty printer* antes de poder comenzar a trabajar en el problema que desea abordar en particular.

PREPROCESADORES PARA JAVASCRIPT

La mayor parte de herramientas de preprocesado de JavaScript pueden agruparse dentro de los denominados "minificadores de código", tales como JSMin [19], JSZap [20] o Google Closure Compiler [21]. Este tipo de herramientas tienen como finalidad reducir el tamaño de los scripts JavaScript. A diferencia de PumaScript, estos preprocesadores poseen una funcionalidad fija, la cual no puede ser extendida por el usuario sin tener que modificar la implementación de los mismos. PumaScript, por su parte, permite extender fácilmente sus funcionalidad de análisis y reescritura a través de las meta-funciones.

Una herramienta de pre procesamiento más similar a PumaScript es Sweet.js [10]. Este lenguaje de macros higiénicos permite construir macros programables por el usuario de forma similar al lenguaje LISP y sus derivados [22]. Sin embargo, a diferencia de PumaScript, Sweet.js es un sistema de macros que opera a nivel léxico en lugar de hacerlo a nivel sintáctico. Esto le permite a Sweet.js una mayor expresividad léxica a la hora de definir las macros, lo cual, si bien puede ser de utilidad, al mismo tiempo implica que un programa en Sweet.js puede tener una apariencia muy distinta a un programa JavaScript, dificultando su lectura y mantenimiento. Por el contrario, PumaScript obliga al desarrolla-

dor a mantener la misma estructura sintáctica que JavaScript.

Otra diferencia, es que las macros de Sweet.js son siempre expandidas en la línea donde la macro es llamada. En el caso de PumaScript, en cambio, una meta-función en PumaScript puede realizar introspección de cualquier parte del árbol sintáctico del programa e inyectar código en cualquier porción del programa y no sólo en el lugar en el cual la meta-función fue llamada. Adicionalmente, una meta-función puede decidir ignorar la reescritura en el lugar de la llamada retornando "null".

Finalmente, una diferencia importante entre PumaScript y Sweet.js es que el primero no posee una pasada diferenciada de expansión. Cuando Sweet.js evalúa un programa, expande todas las macros encontradas y genera un código de salida. Durante esta pasada no es posible ejecutar código JavaScript fuera del cuerpo de implementación de las macros, sólo las macros son instanciadas y ejecutadas. Por el contrario, en PumaScript las meta-funciones con capacidad de introspección y reescritura conviven con las funciones tradicionales en tiempo de ejecución. PumaScript no define una pasada de expansión de macros o meta-funciones, por lo tanto las meta-funciones conviven con el resto de la funcionalidad provista por JavaScript estándar. De esta manera, quien programa las meta-funciones tiene la libertad de ejecutar las expansiones e introspecciones de las meta-funciones en cualquier momento de la ejecución del programa original.

CONCLUSIONES Y FUTURAS DIRECCIONES

En el presente trabajo presentamos una plataforma de meta-programación para JavaScript denominada PumaScript. Esta plataforma extiende a JavaScript con capacidades de introspección y meta-funciones, lo que permite analizar, manipular y transformar el código fuente de programas JavaScript. Las meta-funciones de PumaScript poseen características similares a macros programables en otros lenguajes, pero con algunas diferencias distintivas, tales como la de permitir realizar búsquedas e introspección en todo el programa, controlar la reescritura en el sitio llamador, la posibilidad de reescribir otras porciones del programa no involucradas en la meta-función o sus invocaciones y la posibilidad de convivir en el mismo tiempo de ejecución con funciones tradicionales.

Como prueba de concepto, utilizamos PumaScript para explorar dos potenciales líneas de investigación. Por un lado, mostramos cómo es posible utilizar técnicas de reescritura de código para convertir patrones de código relativamente lentos en patrones más eficientes. Y en segundo lugar, utilizamos PumaScript para identificar y refactorizar el uso de las API inseguras *"eval"* e *"innerHTML"* en programas JavaScript. En ambos casos el análisis y la experimentación requeridos por la investigación necesitó de muy poco esfuerzo, pudiendo ser escritos y ejecutados en poco tiempo gracias a la infraestructura provista por PumaScript. Esta plataforma nos permitió enfocarnos en las hipótesis y la construcción de los experimentos en lugar de abocarnos a resolver los problemas de procesamiento y generación de JavaScript.

Como futuro trabajo existen varias direcciones que consideramos de interés. En principio, será de utilidad extender las meta-funciones para que actúen como macros higiénicos al inyectar código en diferentes contextos y mejorar el runtime de PumaScript para recopilar información de tipos en tiempo de ejecución sobre variables, parámetros, objetos y funciones. También esperamos integrar PumaScript con tecnologías de desarrollo del lado del servidor, tales como Node.js y Grunt, lo cual permitiría hacer el uso de PumaScript más amigable para desarrolladores, al ser integrado dentro de su proceso de trabajo habitual.

En cuanto a las dos pruebas de concepto mostradas, la reescritura de patrones lentos muestra una utilidad con un potencial muy interesante de confirmarse la utilización de dichos patrones en código real de aplicaciones JavaScript. Para poder verificar dicha ocurrencia, nos proponemos ejecutar los scripts de reescritura sobre proyectos de software libre de gran envergadura y luego medir el

impacto en su tiempo de ejecución. Adicionalmente, explorar el código de dichos proyectos en búsqueda de nuevos patrones de código poco eficiente a fin de refactorizarlo con nuestra plataforma.

Finalmente, PumaScript queda disponible como plataforma de introspección y meta-programación para JavaScript para ser utilizado por otros investigadores o desarrolladores que vean utilidad en alguna de las tantas posibles aplicaciones del análisis estático y dinámico de código, así también como la re-escritura o cross-compilación, permitiendo enfocarse en los problemas particulares que se deseen encarar y no en el procesamiento o manipulación de código JavaScript.

REFERENCIAS BIBLIOGRÁFICAS

[1] CHANG, M; SMITH, E; REITMAIER, R; BEBENITA, M; GAL, A; WIMMER, C; EICH, B; FRANZ, M. (2009). Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. *Proceedings of the 2009 ACM SIGPLAN/ SIGOPS International Conference on Virtual Execution Environments*, Washington.

[2] GOOGLE Inc. (2013). *V8 JavaScript virtual machine*. https://code.google.com/p/v8

[3] GAL, A; EICH, B; SHAVER, M; ANDERSON, D; KAPLAN, B; HOARE, G; MANDELIN, D; ZBARSKY, B; ORENDORFF, J; RUDERMAN, J; SMITH, E; REITMAIER, R; HAGHIGHAT, M.R; BEBENITA, M; CHANG, M; FRANZ, M. (2009). Trace-based Just-in-Time Type Specialization for Dynamic Languages. *Programming Language Design and Implementation (PLDI 2009)*, Dublin.

[4] Surfin' Safari. Announcing SquirrelFish. https://www.webkit.org/blog/189/announcing-squirrelfish/

[5] Advances in JavaScript Performance in IE10 and Windows 8. http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx

[6] HERMAN, D; WAGNER, L; ZAKAI, A. (2013). *asm.js specification*. http://asmjs.org/spec/latest/

[7] HACKETT, B, GUO, S-Y. (2012). Fast and precise hybrid type inference for JavaScript. *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, Beijing.

[8] INTEL Corp, *SIMD in JavaScript*. https://01.org/node/1495

[9] HERHUT, S; HUDSON, R.L; SHPEISMAN, T; SREERAM, J. (2013). River trail: a path to parallelism in JavaScript. *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, Indianapolis.

[10] Sweet.js. http://sweetjs.org/

[11] ECMAScript code generator EscodeGen. https://github.com/estools/escodegen

[12] ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/

[13] ECMAScript 6. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[14] Documentación de la API de Jquery. (2015). http://api.jquery.com

[15] HEIDERICH, M; SCHWENK, J; FROSCH, T; MAGAZINIUS, J; YANG, E.Z. (2013). mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, Berlin.

[16] YUE, C; WANG, H. (2013). A measurement study of insecure javascript practices on the web. *Journal ACM Transactions on the Web (TWEB)*, v.7, n.2.

[17] BRAVENBOER, M; KALLEBER, K.T; VERMAAS, R; VISSER, E. (2008). Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, v.72, n.1-2, p.52-70.

[18] BAXTER, I.D; PIDGEON, C; MEHLICH, M. (2004). DMS: Program transformations for practical scalable software evolution. *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, Washington: IEEE Computer Society, p. 625-634.

[19] CROCKFORD, D. (2003). *JSMin: The JavaScript minifier*. http://www.crockford.com/javascript/jsmin.html

[20] BURTSCHER, M; LIVSHITS, B; ZORN, B.G; SINHA, G. (2010). JSZap: Compressing JavaScript Code. *Proceedings of the 2010 USENIX Conference on Web Application Development*, Boston, p.4.

[21] Google closure compiler. https://developers.google.com/closure/compiler

[22] DISNEY, T; FAUBION, N; HERMAN, D; FLANAGAN, C. (2014). Sweeten your JavaScript: hygienic macros for ES5. *DLS '14 Proceedings of the 10th ACM Symposium on Dynamic Languages*, New York, p.35-44.