Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Febrero 2022 – abril 2022

Etapa IV: Completando la Máquina Virtual de Stókhos

En esta etapa del proyecto vamos a completar la máquina virtual de Stókhos. Toda la funcionalidad de las etapas anteriores debe estar disponible. Además,

- Vamos a evaluar cualquier expresión válida del lenguaje
- Toda acción válida deberá ser ejecutada, alterando el estado de la máquina virtual de manera adecuada
- El ambiente de ejecución va a ser enriquecido con más funciones predefinidas

1. Recordatorio: Formato de Salida y Otros Requerimientos

1.1 Formato de Salida

A pesar de haberlo dicho en etapas anteriores, la mayoría no está respetando los requerimientos, dificultando la corrección del laboratorio. En la etapa IV voy a ser estricto al respecto. Recuerden:

Si una acción se ejecuta sin problemas: **ACK: <eco de la acción >**

Si una expresión se evalúa exitosamente: **OK: <eco de la expresión> ==> <resultado>**

En los demás casos: **ERROR: <mensaje>**, por ejemplo: "ERROR: el símbolo yerba no está definido."

1.2 Requerimientos de Sentido Común

Actualicen el README.md y hagan que sea algo más organizado y legible. Ustedes deberían proveer un script llamado *build* o *build.bat* para generar el programa, de ser necesario, y un script llamado *run* o *run.bat*, para correrlo. Yo terminé preparando esos scripts por mi cuenta: la mayoría fueron triviales, pero algunos me dieron problemas en la etapa 1. Ustedes ni se enteraron de esto porque no quería darles angustias. Algunas instrucciones no eran claras y nadie me dijo si tenían una preferencia entre Linux y Windows: varios proyectos funcionan en ambas plataformas, pero otros no, y yo tuve que probarlos en ambas. Por favor, no usen colores en la salida que hacen imposible corregir: por ejemplo, azul oscuro sobre fondo negro.

Asegúrense que la VM y el REPL cumplen las especificaciones de las etapas anteriores: ¡corrijan los errores! Tanto la VM como el REPL nunca deberían caerse: usen excepciones de manera adecuada, sin excesos.

La etapa IV debe entregarse en la rama principal: de esa manera no tengo que probar varias ramas y adivinar.

Extensiones al lenguaje o la funcionalidad del REPL, *incluyendo las que se les ocurran a Ustedes*, pueden ayudarles a mejorar la nota de entregas anteriores. Comuníquense conmigo para negociar y darles sugerencias.

2. Expresiones

A estas alturas del proyecto, por no decir de su carrera, deberían tener buena intuición de lo que hay que hacer para evaluar expresiones. Stókhos es semejante a los lenguajes que ustedes conocen, con la adición de expresiones diferidas como parte del lenguaje.

Recomendamos ***enfáticamente*** volver a **leer la definición informal de Stókhos.** Además, **deben estudiar las láminas de la clase 8 de abril: son parte integral del material de soporte para la etapa 4.**

Aquí recapitulamos, en Castellano , la sección “Conceptos de Lenguajes de Programación”, que comienza en la lámina 27, pero las láminas contienen ilustraciones y ejemplos esenciales para entender la semántica.

Stókhos extiende el modelo tradicional de LVALUE y RVALUE [Christopher Strachey, 1967] de programación imperativa, agregando el concepto de CVALUE para separar el contenido (estado) de una variable del valor de ésta al ser usada en una expresión, es decir el RVALUE de la expresión que consiste en la variable en sí. Dicha extensión de la semántica se debe a que las variables en Stókhos pueden contener expresiones no evaluadas.

Recuerden, una vez más, que no toda variable en Stókhos tiene un identificador (símbolo): los elementos de un arreglo son variables *independientes*. Esto significa que, al igual que en C, Python, JavaScript, Java, etc., un elemento de un arreglo tiene su propio l-value: se le puede asignar un nuevo estado sin tener que tocar las otras variables en el arreglo y puede ser evaluado eficientemente, sin tener que evaluar todo el arreglo.

En particular, si una expresión φ tiene LVALUE, entonces:

$$\text{RVALUE}(\varphi) = \text{RVALUE}(\text{CVALUE}(\varphi)) \qquad (1)$$

Usando la notación equivalente de transformación de evaluación, si una expresión φ tiene LVALUE:

$$\varphi \ =RV=> \ \text{RVALUE}(\text{CVALUE}(\varphi)) \qquad (1)$$

Donde CVALUE(φ) es el estado de la variable denotada por la expresión φ. Con mayor propiedad, podríamos decir RVALUE(φ) = CVALUE(LVALUE(φ)), pero se sobreentiende que el estado de la variable denotada por la expresión φ, es el *contenido de la localidad* denotada por φ. La regla (1) especifica cual es el valor de una variable al ser usada como expresión, sea el estado de la variable una expresión reducida o no.

Con respecto a las expresiones acotadas, tenemos:

$$\text{‘}\varphi\text{’} \ =RV=> \ \varphi \qquad (2)$$

La regla anterior, junto con la regla (1) y la semántica de la asignación - que es *idéntica* a la de los lenguajes imperativos tradicionales - permite implementar el sabor peculiar de *laziness* en Stókhos, es decir su semántica de evaluación diferida de expresiones almacenadas en variables.

De manera trivial, el RVALUE de una constante literal es la constante misma:

$$42 \ =RV=> \ 42$$

true $=RV=>$ ***true***

Sólo nos queda especificar cómo calcular el RVALUE de una aplicación funcional. En general, con la excepción de funciones especiales (por ejemplo, ***formula***) Stókhos emplea evaluación estricta, también conocida como orden aplicativo: los argumentos son evaluados antes de aplicar (llamar a) la función.

$$\text{RVALUE}(\psi(\varphi)) = \text{RVALUE}(\psi)(\text{RVALUE}(\varphi))$$

En este laboratorio, al no tener expresiones lambda o funciones definidas por el usuario, el valor del funtor ψ, es decir el RVALUE(ψ), se obtiene directamente con una búsqueda en la tabla de símbolos predefinidos. Sólo es necesario evaluar los argumentos (tomando en cuenta las excepciones) y despachar la llamada:

$$\text{RVALUE}(\textbf{\textit{floor}}(\varphi)) = \text{RVALUE}(\textbf{\textit{floor}})(\text{RVALUE}(\varphi)) = \textbf{\textit{floor}}(\text{RVALUE}(\varphi))$$

La función ***floor***, en rojo, denota la función interna, que Ustedes implementaron, aprovechándose de otra implementación en el lenguaje usado para implementar la librería funciones predefinidas de la VM. Esta explicación es para que vean que ***existe*** un proceso de evaluación del funtor ☺ pero es trivial.

Evaluar una expresión nunca implica cambiar el ciclo de cómputo, con dos excepciones que veremos más adelante: incluyendo una función cuyo efecto de borde es precisamente incrementar el contador del ciclo.

3. Acciones

2.1 Definición de Variable

Toda definición debe primero ser validada siguiendo los requerimientos de la etapa III. Si la definición es válida, la variable pasa a ser parte del ambiente de ejecución y su estado inicial es el resultado de evaluar la expresión al lado derecho del símbolo de asignación. La semántica de la inicialización cumple con la misma postcondición de la asignación, explicada a continuación.

2.2 Asignación

Toda asignación debe primero ser validada siguiendo los requerimientos de la etapa III. Si la asignación es válida, el *resultado* de evaluar la expresión al lado derecho del símbolo de asignación pasa a ser el *estado* de la variable indicada en el lado izquierdo del símbolo de asignación. De manera más formal, luego de ejecutar la asignación $\boldsymbol{\sigma := \varphi}$; se cumple: $\mathbf{CVALUE}(\sigma) = \mathbf{RVALUE}(\varphi)$.

Al ejecutar una acción siempre debe incrementarse el ciclo de cómputo. Existen variantes, con sus sutilezas, que permiten preservar los invariantes deseados en un ambiente de computación reactivo, como lo es el que están construyendo: en particular, la consistencia de los valores observados, al evaluar expresiones, con las fórmulas de las variables del modelo. Les voy a mandar un quiz respecto a este tema: es opcional, pero voy a evaluar a los que contesten y otorgarles puntos adicionales de acuerdo con la calidad de las respuestas.

4. Nuevas Funciones Predefinidas

Además de las 11 funciones predefinidas descritas en la etapa III, el ambiente de ejecución debe proveer las siguientes:

ln
ln(<exp>)
:: ln := fn(num x) => num -> … ;
La función ***ln*** retorna el logaritmo natural del argumento. Si el logaritmo no existe arroja un error.

exp
exp(<exp>)
:: exp := fn(num x) => num -> … ;
La función ***exp*** retorna el exponencial del argumento x, es decir e^x.

sin
sin(<exp>)
:: sin := fn(num x) => num -> … ;
La función ***sin*** retorna el seno del argumento, es decir $\sin x$.

cos
cos(<exp>)
:: cos := fn(num x) => num -> … ;
La función ***cos*** retorna el coseno del argumento, es decir $\cos x$.

formula
formula(<exp>)
Esta es una función especial que podríamos haber llamado ***cvalue***, ya que simplemente regresa el CVALUE (contenido) de la variable denotada por <exp> *sin evaluar la variable*. En vez de pasar el RVALUE de <exp> a la función, deben pasar el LVALUE, calculado como si <exp> apareciera en el lado izquierdo de una asignación.

tick
tick()
:: tick := fn() => num -> … ;
Al ejecutar esta función la VM incrementa el ciclo de cómputo. La función regresa el número del nuevo ciclo.

array
array(<size>, <init>)
:: array := fn(num size, num init) => [num] -> … ;
:: array := fn(num size, bool init) => [bool] -> … ;
La función array sirve como constructor de arreglo. El argumento *<size>* indica el tamaño del arreglo y el argumento *<init>* especifica como inicializar sus elementos. Lo *único especial* de esta función es que, en principio, hay dos funciones: si *init* es de tipo ***num***, retorna un arreglo de números, si *init* es de tipo ***bool***, retorna un arreglo de booleanos. No hay otras posibilidades, debido a las restricciones draconianas de este laboratorio. Aunque no soportamos polimorfismo paramétrico en Stókhos, este es un simple caso de "sobrecarga": por ejemplo, algunos lenguajes permiten usar el operador + binario (una función con sintaxis "azucarada") para sumar números y para concatenar strings. Este tipo de polimorfismo es fácil de implementar. Si les causa mareos o angustias, sólo implementen la primera versión: ¡necesitamos crear arreglos de números!

Como pueden imaginarse, la función ***array*** evalúa *init* tantas veces como indicado por *size*, usando el resultado de cada evaluación para inicializar un elemento del arreglo. Ahora bien, Stókhos emplea evaluación estricta en general y ***array*** no es especial en este sentido (¿recuerdan? acabamos de decir "lo único especial …") lo cual, junto con acotación, nos da mucha flexibilidad. Vean las siguientes "perlas" de ejemplo:

```
[num] A := array(N, floor(6 * uniform()) + 1);
formula(A)
OK: formula(A) ==> [ 3, 3, 3, 3 ]

[num] B := array(N, 'floor(6 * uniform()) + 1');
formula(B)
OK: formula(B) ==> [ 2, 1, 6, 4 ]

[num] C := array(3, ' 'floor(6 * uniform()) + 1' ');
formula(C)
OK: formula(C) ==> [ floor(6 * uniform()) + 1, floor(6 * uniform()) + 1, floor(6 * uniform()) + 1 ]
```

¿Qué está pasando aquí? Primero vean que la expresión *floor(6 * uniform()) + 1* siempre retorna un número entero entre uno y seis: efectivamente, cada vez que evaluamos esa expresión (nuestro "mini chorizo") estamos lanzando un dado. En el ejemplo que define el arreglo A, esa expresión se evalúa *antes* (evaluación estricta, ¿recuerdan?) de llamar a ***array***, y ese lanzamiento del dado nos dio el número 3. El bello AST del mini chorizo se colapsó en un número: la función ***array*** nunca llego a ver el bello AST de donde vino el 3. Además, como pueden inferir por el resultado final, la evaluación de N *antes* de llamar a ***array*** dio 4, así que array se comportó exactamente como si hubiesen llamado ***array***(4, 3). Como repaso de las reglas de evaluación de expresiones, noten que N tiene l-value: no hay forma de saber si el CVALUE de N es el número 4 o una expresión que al ser evaluada dio 4: ¡de hecho el CVALUE de N puede ser nuestro mini chorizo! Un dado puede dictar que el tamaño del arreglo debe ser 4. No hay forma de saber de donde vino el 4, y tampoco importa, pero nos sirvió para repasar.

En el caso del arreglo B, la expresión acotada '*floor(6 * uniform()) + 1*' se evalúa *antes* de llamar a ***array***; de ahora en adelante vamos a dejar de recordarles que la evaluación de los argumentos de ***array*** es estricta, como lo es por defecto en Stókhos. En la función ***array***, el parámetro *init* es ahora el mini chorizo: las comillas desaparecieron debido a la evaluación antes de llamar a ***array***. El lazo de la función ***array*** que implementa la inicialización de cada elemento del arreglo hace que el mini chorizo se evalúe 4 veces, una para cada elemento, de esta manera:

para i de 0 a 3, A[i] := *floor(6 * uniform()) + 1;*

Así es como terminamos con un arreglo de 4 elementos, cada uno inicializado con un lanzamiento del dado.

El ejemplo del arreglo C nos permite, por primera vez, ver un ejemplo útil de acotación anidada. Pusimos un espacio entre las comillas simples para que no parecieran comillas dobles. Siguiendo las reglas que ya deben conocer de memoria se dan cuenta que la evaluación del argumento *init* en la llamada hace que el parámetro *init* recibido por ***array*** sea nuestro mini chorizo acotado: sólo se perdieron las comillas externas. Ahora el lazo de inicialización en ***array*** se comporta como si se ejecutaran estas asignaciones:

para i de 0 a 2, A[i] := *'floor(6 * uniform()) + 1';*

En cada inicialización, la evaluación del lado derecho elimina las comillas restantes, pero el bello AST de nuestro mini chorizo queda intacto y pasa a ser el CVALUE de cada elemento. Convenientemente, solo tenemos tres elementos en este ejemplo (vean la llamada) para que el resultado de ***formula*** pueda entrar en la línea. ☺

formula(C)
OK: formula(C) ==> [floor(6 * uniform()) + 1, floor(6 * uniform()) + 1, floor(6 * uniform()) + 1]

Si esa es la fórmula de C (el CVALUE del arreglo, que es el arreglo de los CVALUEs de los elementos) ¿cuál será el RVALUE de C? El RVALUE de un arreglo es el arreglo de los RVALUEs de los elementos, así que nos puede dar:

C
OK: C ==> [2, 1, 6]

Cada vez que evaluamos a C, *en un ciclo de cómputo distinto*, vamos a ver con alta probabilidad un resultado distinto. Como repaso, las fórmulas en C deben ser evaluadas a lo sumo una vez en un ciclo de cómputo dado: cada elemento de C tiene su propia memoizacion. Los elementos de C son variables *independientes*.

Como dijimos hace varias semanas, no se pide que implementen acotación anidada, pero permítanlas sintácticamente y asegúrense que lo único que hacen al evaluarlas es eliminar una capa de comillas, es decir retornar el AST que es hijo de la acotación. Si solo hacen esto, se van a sorprender de todo lo que funciona.

Miren lo que hemos logrado: podemos inicializar todos los elementos de un arreglo con un mismo número, indicado por el lanzamiento de un dado. También podemos inicializar cada elemento de un arreglo con un lanzamiento distinto, lo cual nos va a permitir crear arreglos de números aleatorios. Por último, podemos crear arreglos cuyos elementos contienen fórmulas que generan números aleatorios. Todo esto en un ambiente típico, con evaluación estricta y semántica de asignación idéntica a la de los lenguajes imperativos que Ustedes conocen. Sólo agregamos dos ingredientes: acotación y una nueva regla para evaluar expresiones que tienen l-value:

$$\text{RVALUE}(\text{`}\varphi\text{'}) = \varphi$$

$$\text{RVALUE}(\varphi) = \text{RVALUE}(\text{CVALUE}(\varphi))$$

y el resultado es lo que acaban de ver: más bonito y simple no se puede.

histogram

histogram(<exp>, <nsamples>, <nbuckets>, <lower bound>, <upper bound>)
:: histogram := fn(num x, num NS, num NB, num LB, num UB) => [num] -> … ;
Esta función permite evaluar una expresión numérica x en NS ciclos de cómputo, retornando la distribución de frecuencias en (NB + 2) barras. Los valores de x menores que el límite inferior (LB) son contados en la primera barra y los que son mayores o iguales al límite superior (UB) en la última. Cada valor en el intervalo [LB – UB) es contado en la barra correspondiente, entre las NB barras que dividen el intervalo [LB–UB) en partes iguales. Toda evaluación de x que resulte en un error debe ser ignorada, sin parar la ejecución. Internamente, ***histogram*** llama a ***tick*** NS veces, para avanzar el ciclo de cómputo. Ejemplos de uso serán ilustrados en un anexo separado.