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

Etapa III: Verificaciones Estáticas, Ambiente de Ejecución, y Evaluación Restringida de Expresiones

En esta etapa vamos a crear el ambiente de ejecución, agregando una tabla de símbolos globales que incluye tanto funciones predefinidas como variables definidas por el usuario. También vamos a validar acciones y expresiones de Stókhos estáticamente, es decir antes de ejecutarlas y evaluarlas respectivamente. Esto se logra utilizando la información contenida en la tabla de símbolos a la vez que recorremos el árbol abstracto de cada comando. Por último, vamos a evaluar expresiones que sólo utilizan constantes.

1. Validaciones Estáticas

Recuerden que la razón más importante para tener un sistema de tipos es realizar chequeos para tener garantías sobre el comportamiento de un programa *antes* de ejecutarlo.

1.1 Expresiones

Todo símbolo usado en una expresión debe haber sido definido previamente. Los tipos de las expresiones usadas como argumentos en las llamadas a funciones (éstas incluyen "azucares sintácticos": operadores) deben ser consistentes con el tipo de los parámetros de la función. Además, el tipo del valor retornado por la función debe ser consistente con su uso en el contexto de su aplicación: toda llamada a una función es una expresión.

1.2 Acciones

1.2.1 Definición de Variable

El símbolo que denota la variable no debe estar definido y el tipo declarado debe ser consistente con el tipo de la expresión usada para inicializar la variable. Esto lógicamente implica que el inicializador ha sido validado en primer lugar. Cualquier falla en la validación de una definición anula la acción completamente; es decir no tiene efecto alguno en el estado de la VM. Si la definición es válida, el símbolo debe ser agregado a la tabla de símbolos, asociándole un descriptor que, por ahora, sólo necesita contener el AST del tipo de la variable.

1.2.2 Asignación

El lado izquierdo de la asignación debe denotar una variable (es decir, ***es asignable***) de tipo (***ltype***) consistente con el tipo de la expresión usada para asignar (cambiar el valor de) la misma. Cualquier falla en la validación de una asignación anula la acción completamente; es decir no tiene efecto alguno en el estado de la VM.

Desde la perspectiva de la etapa III, esto es todo lo que hay que hacer con las acciones. Todavía no estamos ejecutando acciones, sino validándolas estáticamente y agregando símbolos al entorno. Pero *vayan pensando en los cambios* que habrá que hacer al manejo de las definiciones y asignaciones para ejecutarlas. Si quieren lanzarse a evaluar expresiones más allá de las que sólo tienen constantes, no tengan miedo: esta entrega está definida de tal manera que pueden adelantarse sin violar las especificaciones.

2. Ambiente de Ejecución

La mayoría de las funciones predefinidas de la VM son “mágicas”, o más propiamente “formas especiales”, en el sentido que no son expresables en el lenguaje mismo, aun suponiendo que permitiéramos la definición de funciones. Esto se debe a la simplificación del proyecto, pero no causa problemas. Además, ***no es inusual***.

2.1 Funciones Predefinidas

if

if(<condición>, < expT>, < expF>)
La función ***if*** retorna el resultado de evaluar <expT> si la condición usada como primer argumento se cumple, o el resultado de evaluar <expF> en caso contrario. La condición debe ser una expresión de tipo booleano y <expT> y <expF> deben ser expresiones del mismo tipo. Durante la ejecución, sólo la condición y una de las otras dos expresiones son evaluadas. La función ***if*** es mágica en Stókhos *por la misma razón que lo es en otros lenguajes* que emplean “evaluación estricta”: en vez de evaluar todos los argumentos antes de llamar a la función, sólo evalúa la condición, y la función evalúa uno de los dos argumentos adicionales. Esto es importante, ***y no solo por eficiencia***.

https://en.wikipedia.org/wiki/Evaluation_strategy#Strict_evaluation

type
type(<exp>)
La función ***type*** retorna el tipo de una expresión, *sin evaluar dicha expresión*. Efectivamente, esta función provee una ventana al cálculo de tipos realizado por la VM durante la validación estática, que es implementado por *Uds*. en esta etapa,. Nótese que esta función es mágica por varias razones: a) su resultado es de tipo ***tipo***, el cual no es parte del lenguaje, b) no tenemos forma de especificar el tipo del argumento *<exp>*, ya que puede ser una expresión de cualquier tipo, y c) esta función se evalúa a tiempo de análisis estático, no de ejecución. El siguiente fragmento ilustra el comportamiento de la función:

```
int n := 7;
int a := [ 6 * n, ‘n + 1’ ];

type(n) ==> int
type(a) ==> [int]
type(a[n - 3]) ==> int
type(n + 1) ==> int
type(42) ==> int
```

ltype
ltype(<exp>)
La función ***ltype*** retorna el tipo de una expresión *asignable*, o un error si la expresión no es asignable, es decir que no denota una variable. La función ***ltype*** es mágica por las mismas razones mencionadas para la función ***type***. El siguiente fragmento ilustra el comportamiento de ***ltype***:

```
int n := 7;
int a := [ 6 * n, ‘n + 1’ ];

ltype(n) ==> int
ltype(a) ==> [int]
ltype(a[n - 3]) ==> int
ltype(n + 1) ==> ERROR: la expresion ‘n + 1’ no tiene LVALUE
ltype(42) ==> ERROR: la expresion ‘42’ no tiene LVALUE
```

Las dos últimas son errores porque la expresión no tiene LVALUE, lo cual es equivalente a decir que la expresión no puede ser usada a la izquierda de una asignación, lo cual es equivalente a decir que no denota una variable.

reset
reset()
:: reset := fn() => bool -> … ;
La función ***reset*** elimina todas las variables definidas por el usuario en la VM. Regresa true si es exitosa. La firma de esta función puede ser expresada en Stókhos extendido con abstracción funcional, como ven aquí en rojo.

uniform
uniform()
:: uniform := fn() => num -> … ;
La función ***uniform*** retorna un número entero "aleatorio" entre 0 y 1. Para implementar ***uniform*** en JavaScript usen la función Math.random() y en Python usen la función random.uniform(0, 1). Contacten al preparador en caso de necesitar una función similar. En rojo se muestra la firma de la función ***uniform*** en Stókhos extendido.

floor
floor(<exp>)
:: floor := fn(num x) => num -> … ;
La función ***floor*** retorna el mayor número entero n tal que n <= x. La firma de esta función puede ser expresada en Stókhos extendido con abstracción funcional.

length
length(<exp>)
:: length := fn([<T>] a) => num -> …;
La función ***length*** retorna la longitud de un arreglo de cualquier tipo. Como ven, la firma de esta función podría ser expresada en Stókhos extendido con abstracción funcional y polimorfismo paramétrico.

sum
sum(<exp>)
:: sum := fn([num] a) => num -> …;
La función ***sum*** retorna la suma de un arreglo de números.

avg
avg(<exp>)
:: avg := fn([num] a) => num -> sum(a) / length(a);
La función ***avg*** (average) retorna el valor promedio de un arreglo de números. Una *definición completa* de esta función se muestra en rojo, expresada en un Stókhos extendido con abstracción funcional.

pi
pi()
:: pi := fn() => num -> …;
La función ***pi*** retorna una buena aproximación (ver fp64 en Wikipedia) a la constante π.

now
now()
:: now := fn() => num -> …;
La función ***now*** retorna un número entero correspondiente al número de milisegundos transcurridos desde un punto de referencia en el tiempo, siendo el "Unix Epoch" el más popular. Sin embargo, el punto de referencia no es importante: lo que nos interesa es poder medir el tiempo transcurrido entre eventos. Este enlace puede ayudarles a conseguir una función que cumpla con estas características: https://currentmillis.com/

En la etapa IV vamos a agregar más funciones predefinidas. Enriquecer el entorno hace el proyecto mucho mas interesante y divertido. Una función en particular nos va a permitir realizar simulaciones estocásticas.

3. Evaluación de Expresiones Triviales

En esta etapa también vamos a comenzar a evaluar expresiones, limitándonos a las que no usan variables definidas por el usuario, por ejemplo: ***true***, 42, 6 < 7, 6 * 7 + 2^3, ***pi***(), ***sum***([1, 2]), etc. La restricción de no usar variables sólo aplica a la *evaluación dinámica* en esta etapa. Al hablar de *evaluación*, se asume por defecto que nos referimos a evaluación dinámica, es decir a tiempo de ejecución.

El cálculo de tipos es un ejemplo de *evaluación estática*. Para reiterar lo obvio, las expresiones mencionadas en la sección 1.1 *incluyen* las variables definidas por el usuario. Es decir, las validaciones *estáticas* en esta etapa aplican a *cualquier* expresión de Stókhos. Por esta razón, guardamos el AST correspondiente al LTYPE de cada variable en su descriptor: esto nos permite realizar el chequeo de tipos. En la etapa IV vamos a cubrir la evaluación de todas las expresiones y la ejecución dinámica de las acciones.

Por lo tanto, en esta etapa deben implementar la semántica de *todas* las funciones predefinidas y *todos* los operadores incluidos en el lenguaje (que son funciones predefinidas con sintaxis azucarada) evaluándolos cuando sus argumentos cumplen las restricciones de esta etapa y arrojando un simple mensaje de error cuando no las cumplen, por ejemplo: "ERROR: evaluación de variables no implementada en esta etapa".

En la etapa IV veremos en más detalle cómo manejar expresiones acotadas en Stókhos. Por ahora, introducimos la meta-función de evaluación, con la notación ***=RV=>***, usando ejemplos de esta etapa:

7 * 7 – 7 *=RV=>* 42 **6 < 7 *=RV=>* *true*** **!*true* *=RV=>* *false***

En Stókhos, el resultado de evaluar una expresión encerrada entre comillas simples es la expresión misma:

'7 * 7 – 7' *=RV=>* 7 * 7 – 7 **'6 < 7' *=RV=>* 6 < 7** **'x + y' *=RV=>* x + y**

Como ven, evaluar expresiones acotadas en Stókhos, con o sin variables, *es trivial*. Lo difícil es *regenerar* las expresiones a partir del árbol abstracto para que se vean bonitas, pero no tienen que preocuparse por esto: vamos a permitir que las expresiones se vean de la misma forma que lo hacen al mostrar el AST en la etapa II.

La acotación de expresiones se vuelve interesante al usar variables. La acotación es lo que nos permite almacenar expresiones no reducidas en las variables: esta es la clave que le agrega la *sazón reactiva* al lenguaje. Aquí se muestra una sesión interactiva usando el ejemplo clásico de las ***tres cajitas***:

<Stokhos> num x := 7;

ACK: num x := 7;

<Stokhos> num y := 8;

ACK: num y := 8;

<Stokhos> num z := 'x + y'; **🡐 noten la acotación: esto hace que z almacene la expresión x + y**

ACK: num z := x + y;

<Stokhos> z

OK: z ==> 15

<Stokhos> x := 8; **🡐 al cambiar la x ...**

ACK: x := 8;

<Stokhos> z

OK: z ==> 16 **🡐 ... el resultado de evaluar z cambia!**

4. Sugerencias y "Reglas de Juego" para la Entrega

5.1 Buenas noticias:

Uds. ya han completado los comandos mágicos del REPL. No va a haber tal cosa como un .check para validar acciones y expresiones. Tenemos todo lo que necesitamos para probar la implementación recortada del lenguaje. Sin embargo, no quiten los comandos .lex y .ast: la magia anterior todavía es útil para investigar errores a nivel sintáctico y deben mantenerla como parte de la entrega.

Si un comando en el REPL no es mágico (no comienza con un punto) simplemente lo envían a una función llamada ***process*** (nombre obligatorio) para ser procesado. Este va a ser el uso más frecuente del REPL.

5.2 Arquitectura sugerida

La función ***process*** pasa el input al parser (reconocedor sintáctico) con la esperanza de obtener un AST. El parser ya sabe cómo usar el lexer/tokenizer, asumiendo que Uds. han seguido las sugerencias hasta ahora. Si el parser regresa un error, indicado en el mismo AST (nada les prohíbe tener un nodo de tipo ***error***) o por medio de una excepción, Uds. ya deberían saber que hacer. Si el AST es "*bueno*" (sintácticamente hablando), ***process***, siendo parte de una VM paranoica ☺, lo pasa a ***validate***, para ver si es "*realmente bueno*". La función ***validate*** realiza las validaciones estáticas requeridas por esta etapa, más las que se les puedan ocurrir, y retorna el AST, posiblemente transformado. Ahora ***process*** tiene que tomar una decisión:

- Si el AST es una acción (definición o asignación), lo despacha a ***execute***.
- Si el AST es una expresión, lo despacha a … (wait for it) … ***eval***.

Alternativamente, ***process*** puede decidir despachar el AST a ***execute*** o ***eval*** *antes* de hacer las validaciones, y cada una de esas dos funciones manda a hacer las validaciones necesarias. Noten que validar acciones implica validar expresiones, y ejecutar acciones implica evaluar expresiones, así que, de una forma u otra, es bueno que implementen su librería de validadores y evaluadores de nodos de AST, compartida por el resto de la VM.

5.3 Output y Otros Requerimientos

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

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

Los mensajes de error deben comenzar con "**ERROR:** ", por ejemplo: "ERROR: símbolo no definido: tirge"

README.md actualizado de ser necesario (no me conviertan en empleado de soporte de sistemas)

REPL que cumple las especificaciones de todas las etapas anteriores (¡corrijan los errores!)

5.4 Consideraciones Finales

Parece ser que algunos esperaron hasta la ultima hora para comenzar a implementar el parser. Por favor no se vuelvan complacientes. También hemos notado que no leen con cuidado las etapas y la definición del lenguaje. Apliquen un mayor esfuerzo, razonando sobre lo que leen, para poder entender las implicaciones.

Nos reservamos el derecho de agregar o modificar requerimientos de ser necesario. Hasta ahora nos ha ido bien, pero la experiencia enseña que es fácil olvidarse de algo o conseguir un detalle inesperado. Créanme que todos nosotros hacemos un esfuerzo para que los estudiantes aprendan sin más estrés del necesario.

Están cercanos a implementar un lenguaje: ¡Les deseamos un buen trabajo!