Práctica

PROG

Conceptos Avanzados de Programación

Centro: UPC - FIB
Titulación: Grado en Ingeniería Informática
Cuatrimestre: 2018-2019 Q1
Profesor: Jordi Delgado Pin

Alumnos:
Isabel Codina García
Borja Fernández Ruizdelgado

Índice

Introducción **2**

Escribiendo el código **3**

Binding de las variables 4

Binding de las instrucciones 5

Binding de return y ejecución y Retorno con continuación 6

Testing **7**

Conclusiones **9**

Introducción

El objetivo de la práctica de CAP de este año es entender el funcionamiento de la pila de ejecución y poder modificarla. Estos conceptos han sido explicados a lo largo del curso y es posible aplicarlos gracias a que *Pharo* permite la introspección y intercesión de la pila.

Después de leer el enunciado, nos empezamos a plantear cómo realizar la práctica. La primera idea que surgió fue la de hacer todo en un string y utilizar el método ***compile:***. Nos dimos cuenta de que era demasiado complejo por lo que deberíamos encontrar otra manera.
Leyendo el enunciado, vimos la traducción dada por el profesor y nos dimos cuenta de que era un bloque dentro de otro así que nos surgió la idea de hacerlo de forma recursiva.

Escribiendo el código

Después de las lecturas del enunciado y del capítulo 14, creamos el paquete ***Practica*** con la clase ***PROG*** y con la función ***withInit:do:*** en el class side.

Posteriormente, tal y como dice el enunciado de la práctica, creamos el método ***changeBinding:*** dentro de la clase *Symbol* el cual nos permite cambiar en tiempo de ejecución el *binding* de un símbolo en un bloque. Este método es casi igual al método de ***binding*** del paquete *DeutschByte* con la diferencia que en este método en vez de consultar el valor lo sustituimos por uno diferente y a demás retornamos de dentro de la función; a la hora de retornar lo hacemos con *nil* puesto que no necesitamos retornar nada relevante para el propósito de esta práctica.
Para comprobar que se ha realizado correctamente, creamos la clase ***PracticaTest*** donde escribiremos el test para ***changeBinding:***. Este test está explicado más adelante en el capítulo de Testing de este informe.

En el enunciado de la práctica se nos da una "traducción" a cómo debería ser el código resultante de la función que hemos de implementar en *Smalltalk* por lo que decidimos dividir la función ***withInit:do:*** en 4 fases diferentes:

1. Binding de las variables.
2. Binding de las instrucciones.
3. Binding de return y ejecución.
4. Retorno con continuación.

Binding de las variables

Una vez definido el *Roadmap* empezamos haciendo una función que nos permitiese, dado un bloque, hacer *binding* de varios símbolos de manera recursiva dentro del mismo. La función la llamamos ***addVariables:in:*** que acepta el *Array* de variables como primer parámetro y un bloque como segundo parámetro.

En este punto encontramos un problema ya que en esta función cogemos de manera recursiva la última variable de este *Array* de variables, pero no sabemos de qué tipo es, puesto que según se especifica en el enunciado se pueden pasar variables que son tanto un símbolo suelto (**{#var}**), como un *Array* con un símbolo y el valor que queremos asignarle inicialmente (**{#var . 10}**). Para saber qué tipo de variable se nos pasa y poder hacer el *binding* correctamente utilizamos la introspección que nos permite hacer *Pharo* sobre los objetos con el fin de discernir si la variable es un *Array* o no.

Finalmente, englobando el bloque pasado por parámetro, creamos un nuevo bloque con el *binding* del símbolo con su valor correspondiente. Eliminamos la última posición del *Array* de variables y comprobamos si está vacío. Si aún hay variables, llamamos otra vez a la misma función recursivamente con el *Array* de variables sin su última posición y el nuevo bloque que acabamos de crear.

Para comprobar el buen funcionamiento de la nueva función creamos un test simple pero efectivo dentro de ***PracticaTest***. Véase el apartado de *Testing* para más información.

Binding de las instrucciones

Una vez comprobado el buen funcionamiento de la función descrita anteriormente pasamos al siguiente punto de nuestro *Roadmap* en el cual necesitamos una función para poder hacer *binding* de símbolos con bloques de código, tal y como se especifica en el enunciado.
Para ello definimos una nueva función llamada ***addCode:in:*** la cual acepta un *Array* con el código como primer parámetro y finalmente un bloque de código en el cual queremos hacer el *binding*.
El problema de esta función reside en que la última de las instrucciones requiere de poder hacer retorno mediante el símbolo **#RETURN** por ello, en esta función, se coge la última instrucción del *Array* de código y se crea un bloque nuevo con el *binding* pertinente para finalmente llamar a otra función recursiva auxiliar en el caso en que el *Array* de código tenga más instrucciones a las que hacer *binding* dentro del bloque. Cabe destacar que esta función opera de manera casi idéntica a la de añadir variables pero decidimos crear una función a parte para crear código lo más claro posible.

Definimos la función recursiva, que requiere de la anterior para operar correctamente, llamada ***addCode:in:previousLabel:*** la cual nos permite añadir de manera similar a como funciona la función de añadir variables. Está función es muy interesante puesto que es la que nos permitirá pasar de una línea a otra de ejecución y por ello hay que prestar mucha atención a su implementación.
Lo primero que observamos dentro de la implementación que hemos realizado es que la función tiene un nuevo parámetro llamado **previousLabel** el cual hace referencia a la etiqueta de código que queremos que se ejecute posteriormente a la actual (recordamos que estamos iterando recursivamente de la última parte del código a la primera y por ello **previousLabel** hace referencia a la siguiente línea de instrucción que queremos ejecutar).
Esta etiqueta anterior es de suma importancia, ya que cuando hacemos el *binding* del símbolo con el bloque de código, no solo ponemos el código que se nos pasa por parámetro si no que también tenemos que añadir el código que nos permite ejecutar la siguiente línea de código, que será ***thePreviousLabel binding value***. Esta parte del código es una de las claves del funcionamiento de esta práctica puesto que se tiene que ejecutar el programa secuencialmente línea por línea, y esto permite pasar de una línea a la siguiente. Esto nos da a entender que para realizar un salto de cualquier línea del código a otra etiqueta, por ejemplo a **#label5**, hemos de hacer el *binding* de la etiqueta (***#label5 binding***) que nos dará

el bloque que tiene como valor este símbolo y finalmente ejecutarlo con *value* (***#label5 binding value***).

El contenido de cada etiqueta (**#label1**, **#label2**, etc.) es el bloque creado dentro de ***addCode:in:previousLabel:***, y no es posible comprobar que contiene el bloque dado por los parámetros de **PROG**. Si comprobamos el contenido de los labels, veremos solo los parámetros utilizados en el bloque y no su valor. Por ello no es posible hacer un test convencional, por lo tanto hemos realizado tests manuales con *Halt* e inspeccionando el contexto.

Aunque parece que haya sido sencillo crear estas últimas funciones, no ha sido así. Se han tenido algunos problemas que se pueden ver en el apartado de Testing.

Binding de return y ejecución y Retorno con continuación

Para terminar se realizan los dos últimos puntos de nuestro *Roadmap* simultáneamente.
Con el fin de poder retornar el valor de **#RETURN** necesitaremos una continuación tal y como se explica en el enunciado. Utilizamos ***callcc:*** para capturar la continuación y poder volver de la ejecución del código. Hay que pensar que al hacer el retorno, la variable que utilizábamos para capturar la continuación ya no contendrá una continuación si no que contendrá el valor del retorno de la ejecución de nuestro código, y por lo tanto queremos retornar ese valor. Con el fin de ver si hemos de retornar o ejecutar el código utilizaremos una vez más la introspección de *Pharo* y por lo tanto si el objeto es de la clase *Continuation* habremos de ejecutar el código, en el caso contrario retornaremos. Cabe destacar que sería posible generar una lista de instrucciones que devolviese una continuación y en ese caso nuestro código no funcionará correctamente; como el enunciado no especifica si es posible que se devuelvan continuaciones o no hemos decidido dejarlo así; pero en el caso en el que sí que se necesitase devolver una continuación, el programa tendría que ser modificado levemente con una variable en el *Class side* que nos indicase si hemos de retornar el resultado o hemos de hacer la ejecución del código, en vez de utilizar la introspección que se hace ahora mismo.
Se ha decidido crear las funciones en el *Class side* con el fin de poder aprovechar las funciones para poder insertar dentro de bloques tanto código como variables sin necesidad de tener una instancia de la clase ***PROG*** ya que pueden resultar bastante útiles en futuras aplicaciones.

Testing

Para comprobar que todo funciona correctamente, se deben hacer pruebas. Pero estas pruebas no se realizan solo al final, ya que los errores podría estar en cualquier punto del código. Por ello hemos ido realizando tests a medida que íbamos avanzando en el progreso de escribir el código. Creamos la clase ***PracticaTest*** en la que escribimos todos los test.

La primera parte del código fue la de cambiar el *binding* de los símbolos (***changeBinding:***). Así que tenemos que probar que dentro de un bloque podemos cambiar el *binding* de un símbolo a otro valor con ***changeBinding:*** y una vez se recupere con ***binding*** sea el nuevo valor. Escribimos un test sencillo en el que con el ***bindTo:in:*** asignamos al símbolo **#a** el valor 1 dentro de un bloque en el que inicialmente se comprueba con una aserción que el *binding* es 1, luego se cambia el *binding* a 2, y ese nuevo valor se debe ver cuando se compruebe en la segunda aserción con el ***binding*** del símbolo.

```
testChangeBinding
#a bindTo: 1 in: [
    self assert: #a binding = 1.
    #a changeBinding: 2.
    self assert: #a binding = 2 ].
```

La prueba finalizó correctamente a la primera, así que pasamos al siguiente apartado.

La segunda parte fue la de hacer el *binding* de las variables del programa (***addVariables:in:***). La prueba esencialmente era ver si la diferenciación del tipo de variables era correcta: si se diferenciaban aquellas que tenían inicialmente un valor de las que no. No tenemos que probar nada más del ***addVariables:in:*** ya que básicamente es el ***bindTo:in:*** que ya hemos probado anteriormente. Entonces creamos un bloque con dos aserciones que comprobaran que el *binding* del símbolo **#b** es *nulo*, y el de **#a** es 2, ya que el array que pasamos como parámetro a ***addVariables:in:*** es **{ #b . { #a . 2 } }**.

```
testVariableBinding
| block |
block:= [self assert: #b binding = nil.
         self assert: #a binding = 2 ].
block := PROG addVariables: { #b .{#a . 2} } in: block.
block value.
```

El resultado de la prueba es correcto, así que damos por terminado el apartado de las variables.

Así pues, llegamos al momento de probar si añadir las instrucciones funciona, y aprovechamos las pruebas dadas en el enunciado para probarlo. Simplemente con la primera, nos damos cuenta de que algo no funciona. *Pharo* se queda colgado, significando que hay un bucle infinito en algún lugar del código generado.

El problema fue a la hora de capturar el contexto. Observamos que si para la etiqueta previa utilizamos una variable global, el bloque no captura dentro de su contexto el valor si no la referencia, por lo que no podemos seguir ejecutando más que la primera instrucción debido a que el valor final de esta es siempre la primera etiqueta (la última en haberse asignado debido a tratar el código de atrás a adelante) y por ello el programa salta una y otra vez a la primera línea de instrucciones, haciendo un bucle infinito.

Nos dimos cuenta de la solución de este problema cuando la lista de instrucciones era de solo 1 instrucción, ya que entonces funcionaba correctamente, así que el problema tenía que estar en avanzar a la siguiente instrucción. La solución fue pasar por parámetro la anterior etiqueta.

Cambiado el código, los ejemplos funcionaban perfectamente, así que decidimos ir más allá.

Primero probamos un código en el que se comprobara que se podían poner cualquier símbolo como etiqueta en las instrucciones, no solo **#label1**, **#label2**, etc… Es decir, que no estuviera escrito en el código directamente que si estabas en la 1 pasaras a la 2, etc.. En teoría debería funcionar, ya que hace un ***bindTo:*** de un símbolo, y este podría ser cualquiera, y efectivamente funcionó.

Posteriormente, decidimos probar la versión iterativa del factorial. Primero escribimos un código en el *Workspace* que funcionara y posteriormente lo “traducimos” a la lista de instrucciones utilizando el *binding*. Debería funcionar ya que era como uno de los códigos de ejemplo, y así fue.

Finalmente, decidimos probar si las continuaciones funcionaban. Así que cogimos de las transparencias 3 ejemplos de continuaciones y las “traducimos” a la lista de instrucciones. Y el resultado fue exitoso.

Conclusiones

Esta práctica nos ha permitido entender mejor cómo funciona la pila de ejecución ya que el funcionamiento de la misma depende de la capacidad de *Pharo* de modificar la pila en tiempo de ejecución, permitiéndonos hacer métodos como el ***bindTo:in:*** o el ***changeBinding:***, e incluso permitirnos utilizar Continuaciones.
También nos ha ayudado a acabar de entender el concepto de *Closure*, ya que debido a nuestros errores hemos podido observar cómo se capturan los diferentes tipos de variables.

Nos gustaría acabar diciendo que, gracias a esta práctica, hemos aprendido bastante de Smalltalk el cual nos parece un lenguaje de programación diferente a lo que estamos acostumbrados y encontramos que la implementación de la orientación a objetos es muy correcta.