Student Understanding of the C++ Notional Machine Through Traditional Teaching with Conceptual Contraposition and Program Memory Tracing

Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, Vladimir Lara-Villagrán
CITIC-ECCI
Universidad de Costa Rica
{jeisson.hidalgo, gabriela.marin, vladimir.lara}@ucr.ac.cr

***Abstract*—In order to learn a programming language, a correct understanding of its notional machine is mandatory. Students acquire that comprehension mainly through visual and verbal explanations provided by professors, books, videos, and other instructional materials. This research applied the conceptual contraposition technique and program memory tracing technique to the prevalent teaching method in our country: the lecture. The understanding of the C++ notional machine was evaluated on students of a Programming II (CS2) course that implemented the mentioned methods. Results revealed that these techniques applied to the lecture are insufficient to help students develop satisfactory mental models of the C++ notional machine.**

Keywords—programming learning; notional machine; lecture; constructivism; program memory tracing; program visualization

I. INTRODUCCIÓN

A diferencia de otros tipos de textos, como novelas o artículos científicos, el código fuente de los programas para computadora tiene un dualismo que es obvio para programadores consolidados, pero no tanto para aprendices de la programación [1]. Aunque el código fuente se escribe de forma estática en editores de texto, tiene un comportamiento dinámico a la hora de ejecutarse en la computadora. Una comprensión correcta del comportamiento dinámico es imprescindible a la hora de escribir código estático válido, y esta habilidad es especialmente difícil para algunos estudiantes [1].

Para comprender el comportamiento del código en tiempo de ejecución, es necesario comprender cómo trabaja la máquina que se está programando. Es común que varios cursos se incluyan en las carreras de computación con este objetivo, como circuitos digitales, arquitectura de computadoras y ensambladores [2], los cuales no son necesariamente previos a los cursos de programación. Sin embargo, no es requisito un dominio de la máquina real para comprender la dinámica de los programas en tiempo de ejecución. Los lenguajes de programación, a través de sus constructos, proveen una abstracción de la máquina real que Boulay et al. en 1981 llamaron **máquina nocional** [3]. Cada lenguaje de programación provee una máquina nocional. Por ejemplo, un programador de C podría concebir que la máquina posee tipos de datos y ejecuta funciones, cuando la máquina real no dispone de estos constructos. Un programador de Java podría pensar que la máquina es capaz de ejecutar métodos polimórficamente y dispone de un recolector de basura [4]. La parte derecha de la Fig 1 representa la relación entre la máquina nocional y la máquina real mediada por el lenguaje de programación.

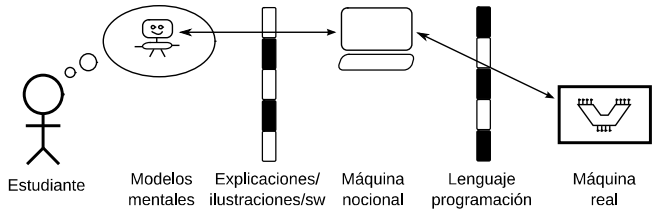

Fig 1 Modelos mentales de la máquina nocional

En su tesis doctoral, Sorva indica que los estudiantes no se apropian directamente de la máquina nocional del lenguaje en que programan, sino que crean modelos mentales de ella [1], como se diagrama en extremo izquierdo de la Fig 1. Los modelos mentales son subjetivos y basados en creencias, por lo que son propensos a imprecisiones. Amparado por amplia literatura científica, Sorva afirma que modelos mentales correctos de la máquina nocional son imprescindibles para el aprendizaje de la programación [1], [4].

Los estudiantes construyen los modelos mentales de la máquina nocional a través de las explicaciones verbales y visuales que realizan los profesores y los materiales didácticos empleados. Hertz y Jump encontraron que utilizar la técnica del *rastreo de memoria de programa* para explicar el funcionamiento de la máquina nocional ayudó a los estudiantes a crear modelos mentales correctos [5]. La técnica fue empleada desde el inicio de curso tanto por el profesor en sus exposiciones, como los estudiantes durante trabajos grupales durante las lecciones [5].

En el caso de nuestra universidad, la fuente primaria para la construcción de modelos mentales son las explicaciones realizadas por el profesor en el salón de clases, pues los cursos de programación se han impartido tradicionalmente de forma magistral, con apoyo opcional de libros de texto. Existe cuestionamiento en la comunidad científica sobre la efectividad de los métodos educativos en el aprendizaje de la programación, incluyendo la clase magistral [6]. Las

978-1-4673-9143-6/15/$31.00 ©2015 IEEE

principales objeciones a este método son la pasividad del estudiante y la falta de motivación intrínseca [7].

Los autores del constructivismo social sugieren utilizar la técnica de la *contraposición conceptual* para incrementar la motivación intrínseca y propiciar un estado mental adecuado para la asimilación de nuevos conceptos [7]. El uso de esta técnica ha sido poco reportado en la literatura científica en el contexto de la educación de la computación. Ma et al. encontraron resultados positivos en los modelos mentales de los estudiantes al aplicarse de previo a visualizaciones de programa [8]–[10]. Sin embargo, no se ha encontrado evidencia de la aplicación de la contraposición conceptual a la clase magistral, la cual sigue siendo el método de enseñanza prevalente en nuestro país y probablemente en la mayoría de universidades latinoamericanas.

El objetivo de este trabajo es evaluar cualitativamente la comprensión (los modelos mentales) que tienen los estudiantes de la máquina nocional de C++, tras haber tomado un curso de programación siguiendo una didáctica tradicional (clases magistrales) enriquecida con las técnicas de contraposición conceptual y rastreo de memoria de programa. La sección que continúa explica los métodos de aplicación (contraposición conceptual, rastreo de memoria de programa) y de evaluación de los modelos mentales (coloquio). La sección 3 expone los resultados obtenidos. La sección 4 presenta una interpretación de los resultados. Como es habitual, la sección 5 concluye el trabajo con un resumen del aporte de los resultados y propone formas de enriquecer la investigación afín.

II. METODOLOGÍA

Se tomó como escenario el curso de Programación II de la carrera de Computación, impartido por uno de los autores de este artículo durante agosto a noviembre de 2014 en la Universidad de Costa Rica. Por convención el lenguaje de programación utilizado en este curso fue C++. La modalidad didáctica fue magistral. Cada concepto de programación que tiene un impacto en la máquina nocional, se introdujo utilizando las técnicas de contraposición conceptual y rastreo de memoria de programa. La evaluación se realizó al final del curso utilizando coloquios entre el profesor y cada estudiante que participó voluntariamente.

La técnica de **contraposición conceptual** consiste en plantear una situación o problema donde los conceptos presentes en la mente del estudiante son contradictorios o insuficientes para resolverlo [7]. El estudiante entra en un estado de incertidumbre cognoscitiva al vivenciar que sus creencias o conocimientos son equivocados o incompletos [7]. Para superar la incertidumbre y alcanzar un estado de equilibrio mental, el estudiante debe ineludiblemente reorganizar viejos conceptos o construir nuevos, lo cual ocurre de forma intrínseca sin dependencia de refuerzos externos [7]. La técnica de contraposición conceptual recibió otros nombres en Occidente, como teoría de la disonancia cognitiva (en inglés, *cognitive dissonance*) o conflicto cognitivo (en inglés *cognitive conflict*). Este último es el término empleado por Ma et al. [8]–[10].

La técnica de la contraposición conceptual se aplicó durante las clases magistrales para introducir conceptos de programación que tienen un efecto en la máquina nocional de C++. Por ejemplo, como preámbulo para introducir el concepto de apuntador, el profesor escribió un programa en la computadora de proyección, como el incluido en la Fig 2. El programa imprime todos los valores enteros entre dos números dados por el usuario. Si el usuario ingresa un rango invertido el programa los intercambia (`swap`) y procede como de costumbre.

```
01 #include <iostream>
02
03 void swap(long a, long b)
04 {
05    long temp = a;
06    a = b;
07    b = temp;
08 }
09
10 int main()
11 {
12    long a, b;
13    std::cin >> a >> b;
14
15    if ( a > b )
16       swap(a, b);
17
18    for ( long i = a; i <= b; ++i )
19       std::cout << i << ' ';
20
21    std::cout << std::endl;
22 }
```

Fig 2 Programa en C++ que imprime los valores en un rango entero

El profesor interrogó a los estudiantes acerca de sus predicciones sobre el programa. A vista de los estudiantes el programa parecía perfecto. El profesor ejecutó el programa en la computadora de proyección (Fig 3). Al ingresar un rango correcto (`10 20`, resaltado en negritas en la Fig 3), el programa imprimió los valores en el rango correctamente. Al ingresar un rango invertido (`20 10` en la Fig 3) el programa debió imprimir el mismo resultado, pero su salida fue vacía. Ver fallar el programa que parecía perfecto debió propiciar un estado de incertidumbre mental, necesario para el aprendizaje del nuevo concepto de acuerdo a la teoría de constructivismo social [7].

```
$ ./range
10 20
10 11 12 13 14 15 16 17 18 19 20
$ ./range
20 10

$
```

Fig 3 Dos ejemplos de ejecución del programa de rango en Unix. El símbolo $ indica que el sistema operativo está a la espera de comandos. Los textos en negritas son ingresados por el usuario. Lo demás es salida del programa.

El profesor explicó el comportamiento del programa en tiempo de ejecución utilizando la técnica de **rastreo de memoria de programa** (en inglés *program memory tracing*). Consiste en recorrer un programa línea a línea reflejando el efecto de cada una de ellas en un dibujo abstracto [5]. El dibujo (Fig 4) representa el estado del programa distribuido entre los diferentes segmentos de memoria de la máquina nocional [5]. En el caso de C/C++, la máquina nocional es muy cercana a la física, con al menos cuatro segmentos cuyos nombres y siglas en inglés son *code segment (CS)*, *data segment (DS)*, *stack segment (SS)* y *heap segment (HS)*. Son representados en la Fig 4 como zonas rectangulares. Para el programa de rango (Fig 2) sólo el segmento de pila (*SS*) es utilizado.

El profesor actuó como el procesador ejecutando cada línea del programa, actualizando el dibujo de memoria en la pizarra,

y explicando verbalmente lo ocurrido con el fin de hacer evidente la causa del fallo. En el caso del programa de rango (Fig 2) la función `swap` recibe dos números como parámetros por valor. Los recibe de esta forma pues son los conceptos presentes en la mente de los estudiantes antes de introducir el concepto de apuntador, y los que probablemente utilicen para implementar el intercambio.

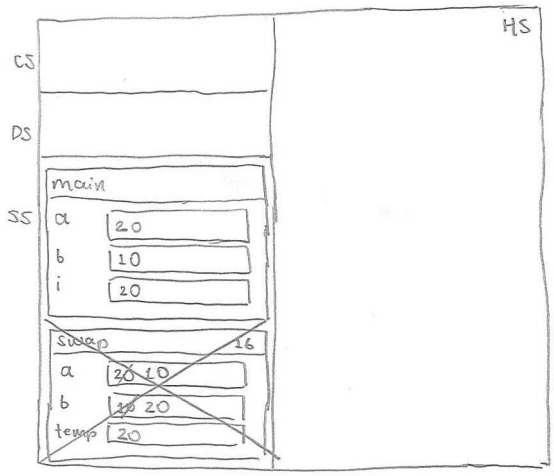

Fig 4 Rastreo de memoria del programa de rango antes de finalizar su ejecución (línea 22). El dibujo fue reconstruido en papel a partir del hecho por el profesor en la pizarra, luego escaneado con fines de legibilidad.

Tras haber observado que el programa falla y su causa, el estudiante aún continúa en estado de incertidumbre mental y requiere saber cómo se corrige. El profesor entonces introdujo el nuevo concepto, en este caso, el apuntador y su sintaxis. Corrigió el programa y lo ejecutó manualmente actualizando en el rastro de memoria de la máquina nocional el efecto de los apuntadores. Finalmente ejecutó el programa en el equipo de proyección, el cual produjo el resultado esperado.

Este método de introducir el concepto tras una contraposición conceptual y su rastro en la máquina nocional, se utilizó para cada concepto que tiene efecto en el estado del programa. ¿Habrán construido los estudiantes modelos mentales correctos de la máquina nocional con este método?

De acuerdo a Vygotsky, el aprendizaje alcanza el nivel de conocimiento si los estudiantes forman sistemas de conceptos que reflejan las relaciones entre los objetos y fenómenos del mundo real, y estos sistemas son estables en el tiempo, es decir, en memoria de largo plazo [7]. A diferencia de estudios previos, en lugar de evaluar aisladamente un concepto de programación recién introducido [8]–[10], se procuró evaluar los sistemas de conceptos sobre la máquina nocional de C++ construidos por los estudiantes durante el semestre. La evaluación se hizo a través de coloquios al finalizar el curso a cambio de puntos adicionales en la nota del mismo.

El **coloquio** es un método de evaluación formativa ampliamente utilizado por los autores del constructivismo social [7]. Consiste en un diálogo entre el profesor y el estudiante mientras el segundo realiza una tarea educativa. En una primera fase el profesor sólo indaga el nivel de dominio que tiene el estudiante sobre el contenido y las habilidades evaluadas, llamado **nivel de desarrollo efectivo** [7]. En una segunda fase, el profesor solicita al estudiante regresar a la tarea, en especial donde ha detectado modelos mentales (sistemas de conceptos) deficientes. El profesor provee preguntas clave, información, herramientas o consejos que ayudan al estudiante a sobrepasar las deficiencias (andamiaje). Es decir, el profesor influye en la zona de desarrollo próximo del estudiante y evalúa el **nivel de desarrollo potencial** que el estudiante logró alcanzar con esta ayuda [7]. La primera fase está más orientada a la evaluación, la segunda más al aprendizaje del estudiante.

La tarea educativa escogida para la evaluación al finalizar el curso fue la siguiente. Se le pidió a cada estudiante que ejecutara línea a línea el programa de C++ listado en la Fig 5. En cada línea debía explicar qué hace la máquina nocional para ejecutarla. Se le brindó al estudiante una hoja en blanco y se le solicitó que dibujara el estado del programa. Un dibujo de rastro de memoria hecho por el profesor se encuentra en la Fig 6. El coloquio fue grabado sólo en audio. No hubo restricciones de tiempo. Los estudiantes tuvieron acceso a la documentación del algoritmo `std::sort()`.

```
01 #include <algorithm>
02 #include <iostream>
03
04 void calc(size_t size)
05 {
06    double* arr = new double[size];
07
08    for ( size_t i = 0; i < size; ++i )
09       std::cin >> arr[i];
10
11    std::sort(arr, arr + size);
12    if ( size % 2 == 0 )
13       std::cout<<(arr[size/2] + arr[size/2 - 1]) / 2;
14    else
15       std::cout << arr[size / 2];
16 }
17
18 int main()
19 {
20    size_t size;
21    while ( std::cin >> size )
22       calc(size);
23
24    return 0;
25 }
```

Fig 5 Programa en C++ utilizado para evaluar

El programa de la Fig 5 calcula la mediana de conjuntos de datos ingresados en la entrada estándar. El tamaño es ingresado antes del conjunto de datos. Se le indicó a cada estudiante asumir que el usuario del programa introduce los valores (`3`, `100`, `70`, `90`, `EOF`), donde el `3` indica que el conjunto de datos tiene tres valores. Para el segundo conjunto se ingresa el carácter especial fin-de-archivo (`EOF`) como su tamaño. El programa reacciona terminando su ejecución. Cada vez que se invoca al método `calc()`, se crea un arreglo en memoria dinámica (línea `06`) que no es eliminado, y por tanto genera una fuga de memoria. Las declaraciones de este párrafo no fueron comunicadas a los estudiantes, excepto los datos de entrada.

No todos los conceptos de programación vistos en el curso se evaluaron durante los coloquios, sino aquellos conceptos requeridos para comprender la ejecución del programa de la Fig 5 en la máquina nocional de C++. Se listan a continuación, junto con las acciones esperadas que debe realizar el estudiante.

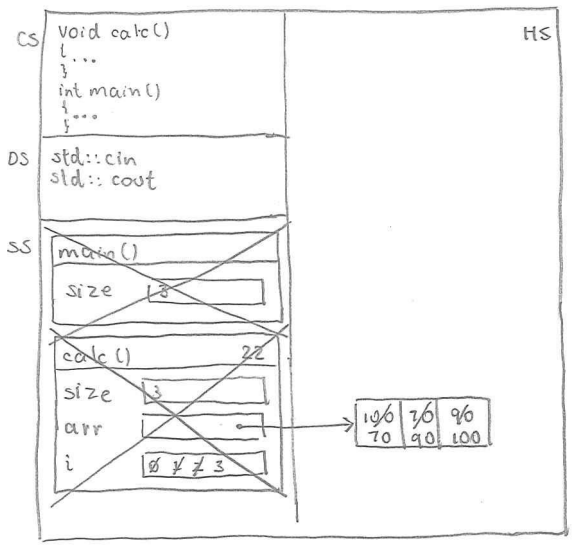

Fig 6 Rastreo de memoria del programa tras finalizar su ejecución (línea `24` de la Fig 5). Hecho por el profesor en papel y escaneado con fines de comparación con los rastreos de memoria producidos por estudiantes.

1. Segmentación de memoria (*memory segmentation*). El estudiante debe distribuir la memoria del programa en cuatro segmentos representados como zonas rectangulares en el dibujo, nombrarlos en español o inglés, e indicar el propósito de cada uno.
2. Punto de entrada del programa (*entry point*). El estudiante debe iniciar la ejecución en la función libre `main` en la línea `18`, y no en `calc` u otra línea del programa.
3. Invocación de función (*function call*). El estudiante debe representar la invocación de ambas funciones, `main` y `calc` con rectángulos (*stack frames*) en el segmento de pila únicamente.
4. Variable local (*local variable*). Se debe representar dentro de la invocación de función a que corresponde y no en otro lugar. Debe tener un nombre y un valor.
5. Ciclo *while*. El estudiante debe indicar que el cuerpo del ciclo (la invocación a la función `calc` en la línea `22`) se invoca repetidamente hasta que la condición del ciclo se haga falsa (el usuario ingresa `EOF`, línea `21`).
6. Entrada estándar (*standard input*). El estudiante debe indicar que el programa espera un valor en la entrada estándar o teclado en las líneas `21` y `09`. Cuando el valor se ingresa, debe dibujarlo en el espacio de memoria destino.
7. Parámetro de función (*function parameter*). Se debe representar como una variable local, y se debe inicializar con el valor provisto durante la invocación de la función. Ocurre en las líneas `22` y `04`.
8. Asignación/alojamiento de memoria dinámica (*dynamic memory allocation*). El estudiante debe dibujar espacio para variables en el segmento de memoria dinámica (*heap segment*) al invocar el operador `new` en la línea `06`.
9. Arreglo o vector (*array* o *vector*). El estudiante debe dibujar el número exacto de elementos continuos (`3` al ejecutar la línea `06`).
10. Apuntador (*pointer*). El estudiante debe dibujar el apuntador como una variable más y representar lo apuntado con una flecha o una dirección de memoria hipotética establecida como su valor, siempre y cuando indique que esa dirección corresponde a la ubicación del dato referido. En el caso de la línea `06`, el apuntador debe ubicarse como una variable local en la invocación de la función `calc`.
11. Ciclo *for*. El estudiante debe ejecutar sus pasos en orden: inicialización, condición, cuerpo, e incremento; y repetir los últimos tres en orden hasta que la condición se haga falsa. También es válido interpretar el ciclo a un nivel mayor de abstracción. Por ejemplo un estudiante podría explicar las líneas `08` y `09` como "aquí se leen todos los elementos del arreglo a partir de la entrada estándar".
12. Función de biblioteca (*library function*). La línea `11` invoca una función que no está definida en el programa sino en la biblioteca estándar de C++. La función se estudió durante las lecciones. Durante el ejercicio se proveyó acceso a su documentación oficial. El estudiante debe indicar el propósito de la función y su efecto en la memoria del programa (ordena los valores del arreglo).
13. Iterador (transliteración de *iterator*). La función de biblioteca `std::sort` no espera un arreglo, sino un rango de valores demarcado por iteradores. Un iterador es un objeto capaz de recorrer los elementos de una estructura de datos, incluso aunque los elementos no estén continuos en la memoria. El iterador imita la interfaz de un apuntador. La función `std::sort(begin,end)` recibe dos iteradores `begin` y `end`, y ordena los elementos comprendidos en el intervalo `[begin, end[`.
14. Aritmética de apuntadores (*pointer arithmetic*). La línea `11` suma un entero a un apuntador (`arr + size`) para obtener un apuntador hacia un elemento inexistente (inmediatamente después del último elemento válido), con el fin de construir un iterador hacia el final del arreglo.
15. Condicional *if/else*. El estudiante debe evaluar la condición de la línea `12`. Si esta es verdadera debe ejecutar la línea `13` y no la `15`. Si la condición se evalúa falsa, debe ignorar la línea `13` y ejecutar la `15`.
16. Evaluación de expresiones (*expression evaluation*). El estudiante debe realizar las operaciones aritméticas o booleanas que componen la expresión de acuerdo a la prioridad de operadores. Además mencionar el resultado de la evaluación, el cual debe ser un único valor.
17. Indexación de arreglos (*array indexing*). Al ejecutar la línea `13` ó `15`, el estudiante debe evaluar una expresión aritmética y utilizar el resultado entero para acceder a un valor del arreglo.
18. Salida estándar (*standard output*). El estudiante debe escribir o verbalizar el valor que se imprime en la salida estándar (por defecto, la pantalla).
19. Retorno de función (*function return*). Cuando el estudiante ejecuta las líneas `16` ó `25`, debe indicar que la función termina su invocación, su memoria (*stack frame*) es liberada del segmento de pila, y el control regresa al punto donde se invocó la función.
20. Fuga de memoria (*memory leak*). Al retornar de la función `calc` en la línea 16, el estudiante debe eliminar el apuntador

`arr` y no la memoria apuntada. Debe descubrir que se provoca una fuga de memoria e indicar sus consecuencias.

21. Liberación de memoria (*memory deallocation*). Si el estudiante descubre la fuga de memoria, el profesor solicitará cómo corregirlo. El estudiante debe alterar el programa para agregar una invocación al operador `delete[]`.
22. División entera (*integer division*). El estudiante debe distinguir entre una división entera y una flotante. Además indicar el resultado de la división entera (`/`) o el residuo (`%`) de acuerdo al operador utilizado. Este concepto se consideró parte del concepto 16, pero fue separado posteriormente, durante el análisis de datos.

Algunos conceptos de programación no se incluyeron en la lista anterior por no tener una implicación en la máquina nocional o por estar incluidos en otros ya listados. Son ejemplos los conceptos: incremento de variable entera, espacios de nombres, el tipo de datos renombrado `size_t`, e inclusión de archivos de encabezado.

III. Resultados

De 18 estudiantes matriculados en el curso, 13 se encontraron activos al finalizar el semestre. Participaron 11 estudiantes en los coloquios de forma voluntaria a cambio de puntos adicionales en la nota del curso. En lo siguiente se resume la actuación de cada uno de ellos, en especial sobre los indicios relacionados con modelos mentales de la máquina nocional. En el Cuadro 1 se incluye una evaluación sumativa subjetiva, elaborada por el profesor del curso a partir de los coloquios. Se calificó la comprensión de cada estudiante sobre cada concepto con un valor real entre 0 y 1, donde 1 indica evidencia de dominio del concepto y 0 ausencia de evidencia. El color de cada celda refleja su valor en un rango de rojo (0) a verde (1). Las celdas en blanco corresponden a valores que no se pudieron obtener a partir de las grabaciones de audio.

Cuadro 1. Calificación subjetiva del profesor para cada concepto por participante en coloquios. El orden de los conceptos (filas) y los participantes (columnas) coincide con los presentados en el texto. `Avg` representa promedio.

		Participante											
Concepto		1	2	3	4	5	6	7	8	9	10	11	Avg
1	seg	0,3	0,7	0,4	0,3	0,6	0,3	0,2	0,4	0,6	0,8	0,8	**0,49**
2	main	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,9	1,0	0,9	**0,98**
3	call	0,0	1,0	0,1	0,5	0,8	1,0	0,8	0,4	0,2	0,6	1,0	**0,58**
4	auto	0,2	0,7	0,5	0,3	0,7	0,7	0,2	0,1	0,7	0,3	0,9	**0,48**
5	while	0,0	1,0	1,0	1,0	0,9	1,0	1,0	1,0	0,9	1,0	1,0	**0,89**
6	cin	1,0	1,0	1,0	1,0	0,7	0,9	0,0	0,7	0,7	0,9	1,0	**0,81**
7	arg	0,0	1,0	0,0	0,1	0,2	0,0	0,1	0,1	0,0	0,8	0,1	**0,22**
8	new	1,0	1,0	0,5	0,8	1,0	0,0	1,0	0,9	1,0	0,7	1,0	**0,81**
9	arr	0,6	0,7	0,3	0,7	0,7	0,4	0,5	1,0	1,0	1,0	0,8	**0,70**
10	ptr	1,0	1,0	0,3	0,7	1,0	0,2	0,9	1,0	1,0	0,6	1,0	**0,79**
11	for	1,0	0,8	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**0,98**
12	sort	1,0	0,9	0,9	1,0	0,9	0,9	1,0	0,8	1,0	0,9	1,0	**0,94**
13	itr		0,0	0,0	0,0	0,0	0,0				0,0	0,6	**0,09**
14	ptr+i		0,1	1,0	1,0	1,0	0,3		0,5		1,0	0,9	**0,73**
15	if	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**1,00**
16	expr	1,0	0,7	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,9	**0,96**
17	arr[i]	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**1,00**
18	cout	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**1,00**
19	return	0,4	1,0	1,0	1,0	0,4	0,2	0,2	0,2	0,5	0,9	0,8	**0,60**
20	leak	0,4	1,0	1,0	1,0	0,0	0,0	0,1	0,0	0,0	0,0	1,0	**0,41**
21	delete	0,7	1,0	0,7	0,8	0,0	0,0	0,1	0,0	0,0	0,0	1,0	**0,39**
22	/%	1,0	0,0	0,2	1,0	0,1	1,0	1,0	0,1	0,0	0,9	0,9	**0,56**
	Avg	**0,68**	**0,80**	**0,68**	**0,78**	**0,68**	**0,59**	**0,66**	**0,63**	**0,68**	**0,75**	**0,89**	**0,70**

Las descripciones cualitativas siguientes se incluyen –en forma resumida– por su valor para explicar los datos sumativos del Cuadro 1, y por su riqueza para explicar los modelos mentales de los aprendices. Se incluye además los rastros de memoria realizados por los estudiantes con menor y mayor rendimiento en Fig 7 y Fig 8 respectivamente, de acuerdo a la evaluación sumativa. En la sección IV se discuten tanto los resultados sumativos como cualitativos.

Participante 1. Distinguió el *heap segment* pero no los demás segmentos. Ubicó dos variables locales en un segmento sin nombre, presumiblemente el *stack frame* del `main`, pero no distinguió las invocaciones de funciones. En lugar del parámetro `size` en `calc` utilizó el `size` del `main`. Agregó el tamaño del arreglo (variable local) como primer elemento del arreglo en memoria dinámica, que lo llevó a decisiones e impresiones incorrectas incluso con andamiaje. Interpretó el módulo 2 como preguntar si el dividendo es par. Terminó abruptamente la ejecución del programa desde la línea `16`. Indicó que la máquina limpia la memoria automáticamente. Detectó y corrigió la fuga de memoria sólo con andamiaje.

Participante 2. No recordó los nombres de los segmentos de memoria, pero tuvo una idea clara del uso de tres de ellos. Especificó claramente la invocación de función y el paso de parámetros. Confundió el tamaño y capacidad del arreglo de reales con una cadena de caracteres, por lo que agregó un espacio más aunque no lo usó. Consideró la aritmética de punteros como una suma de tipos incompatibles, luego realizó suposiciones erradas. Se empecinó en conjeturar la implementación de `std::sort` (violación al principio de encapsulación), necesitó mucho andamiaje para abstraer su funcionamiento. Realizó divisiones reales de índices en lugar de divisiones enteras y se confundió al acceder a los arreglos (línea `15`). A través de supuestos errados llegó a la impresión correcta de los resultados. Evaluó las expresiones aritméticas dibujando valores temporales en la memoria de pila que la máquina nocional no almacena ahí. Detectó y corrigió correctamente la fuga de memoria.

Participante 3. Nombró correctamente sólo el segmento de memoria dinámica. Para los demás explicó correctamente su uso. No representó la invocación a la función `calc` como un *stack frame* con variables locales, sino como un apuntador hacia el valor del parámetro `size`. Idealizó tanto al apuntador como el arreglo como una clase contenedora, ambos ubicados en memoria dinámica. Consideró que el programa ocasiona un error al intentar acceder al elemento en la posición `arr + size` (línea `11`). Dudó si la división entera descarta o redondea decimales. Se inclinó por el redondeo hacia abajo por lo que llegó a imprimir el resultado correcto. Detectó la fuga de memoria. Consideró que el `delete[]` debe invocarse en un ciclo para que elimine todos los elementos del arreglo. Terminó la ejecución del programa correctamente.

Participante 4. Indicó que la máquina divide la memoria en subsecciones o espacios y sólo recordó el nombre de uno de ellos: "la pila". Separó la pila de la memoria dinámica. Consideró la variable `size` (línea `20`) tanto local a `main` como global al programa. Duplicó la invocación de `main` en el segmento de pila. Justificó que `size` estaba dos veces en el segmento de pila porque en una estaba declarada y en otra

inicializada, lo que refleja confusión con el segmento de código. Asumió que `calc` puede acceder a una variable local de `main`. Consideró que un apuntador es lo mismo que un arreglo. Realizó la aritmética de punteros correctamente y consideró que el programa debía caerse al invocar a `std::sort`. Con poco andamiaje pudo avanzar fácilmente. Realizó las divisiones y residuos enteros e imprimió el valor correcto en la pantalla. Detectó la fuga de memoria, pero trató de corregirla con la función de biblioteca `free()`. Requirió abundante andamiaje para recordar el operador `delete[]`.

Participante 5. No recordó los nombres de los segmentos, pero logró indicar que hay tres "partes": "donde van las cosas dinámicas", "donde se corren los métodos", y "donde se guardan las variables globales". Aunque verbalmente explicó la mecánica de invocación de funciones, no lo reflejó en el dibujo (*stack frames*). Separó correctamente entre apuntador y el arreglo apuntado. Utilizó verbalmente tamaños de tipos de datos, y gráficamente flechas y direcciones de memoria hipotéticas para ilustrar el apuntador, lo cual refleja dominio del concepto. Agregó incorrectamente el carácter fin de cadena (`'\0'`) al arreglo de números reales. Explicó la lectura de los elementos del arreglo con alto nivel de abstracción. Tuvo serias dificultades para realizar la división `3/2` y utilizar el resultado como índice en un arreglo. Opinó que el programa no debe compilar. Con andamiaje indicó que el programa se debe caer. No eliminó el *stack frame* al ejecutar la línea `16`. No detectó la fuga de memoria. Terminó la ejecución del programa sin poder explicar cómo se rompe el ciclo `while` en la línea `21`. Requirió andamiaje para recordarlo.

Participante 6. Recordó que existen segmentos de memoria. Separó uno para los recursos del sistema operativo refiriéndose al *heap segment*, y otro "donde van las funciones" en el que ubicó todas las variables del programa porque "no veo que [este programa] use memoria dinámica". Declaró no conocer cómo llega el dato del teclado a la variable cuando se lee de la entrada estándar. Separó las invocaciones de funciones (*stack frames*) con sus variables locales, excepto los parámetros. No distinguió entre apuntador y arreglo; los dibujó como una misma entidad. Explicó el ordenamiento en alto nivel pensando que `std::sort` recibe índices en lugar de iteradores. Realizó correctamente los módulos y divisiones enteras, e imprimió el elemento central correctamente. No pudo explicar cómo `calc` terminó su invocación ni lo reflejó en el rastro del programa (Fig 7). Indicó creer que por razones del estándar [C++] la memoria del programa se libera automáticamente.

Participante 7. Separó tres segmentos de memoria: “el segmento de datos que contiene las variables locales”, otro para funciones al cual no pudo recordar el nombre, y separó la memoria “del sistema” de los dos anteriores sin poder explicar por qué. Creó tanto un apuntador como una función, ambos llamados `calc` sin poder explicar la diferencia. No hizo la lectura del tamaño en la línea `21`, ni se pudo explicar cómo la función `calc` leía el tamaño de la entrada estándar. Construyó el arreglo correctamente en la “memoria del sistema” al invocar el operador `new` en el dibujo, pero verbalmente lo explicó como un arreglo de apuntadores a `double`, luego como una matriz y con andamiaje los interpretó correctamente como un arreglo de valores reales. Explicó en alto nivel el funcionamiento de `std::sort`. Calculó las divisiones y módulos enteros sin errores. Imprimió el valor esperado en la salida estándar. Tuvo serias dudas de cómo el control sale del ciclo en la línea `21` en la primera iteración; sin embargo, cuando la entrada fue el `EOF` lo recordó espontáneamente. Requirió andamiaje para detectar la fuga de memoria.

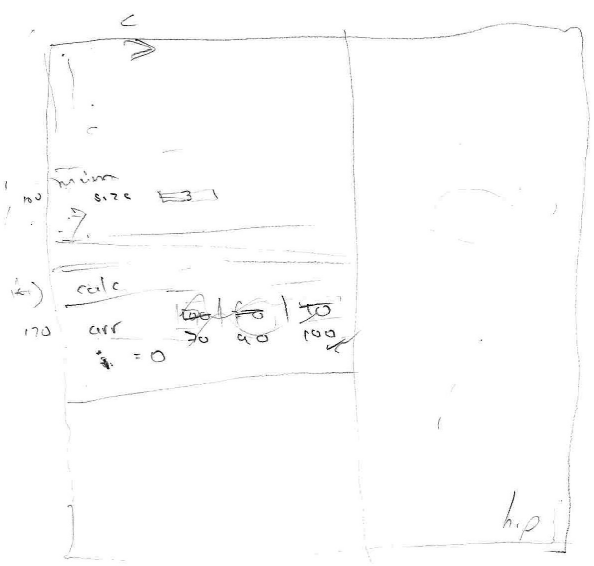

Fig 7 Rastro de memoria del programa realizado por el participante 6

Participante 8. Separó en el dibujo entre variables locales, invocaciones de funciones, y memoria dinámica. Sin embargo, separó las variables locales de sus valores, los cuales colocó en la memoria dinámica indicando que las variables apuntan hacia sus valores. Llenó el arreglo con los valores del contador. Requirió andamiaje para notar que los valores realmente se leían de la entrada estándar. Intentó explicar la implementación del algoritmo `std::sort`, aunque abstrajo luego su efecto en el arreglo. Dudó cómo trabaja el módulo y la división entera. No pudo explicar cómo la computadora indexa en el arreglo a partir de un número real (el resultado de la división `3/2`). Asumió que la computadora toma la parte entera de la división, pero luego dudó y no pudo explicar por qué el módulo `3%2` sí genera un valor entero pero no la división `3/2`. Indicó que la computadora elimina la memoria dinámica automáticamente.

Participante 9. Distinguió correctamente entre el segmento de código y el de memoria dinámica. Sin embargo, separó el segmento de pila (no recordó el nombre) en dos, uno para variables y otro para “invocaciones de métodos y punteros”. Intentó iniciar la ejecución desde la línea `1`, requirió mínimo andamiaje para descubrir la función `main`. Mezcló instrucciones con invocaciones a funciones en uno de los segmentos. Conjeturó y explicó el funcionamiento `std::sort` correctamente en alto nivel. Interpretó inicialmente el "módulo 2" como equivalente a preguntar si el número es par. Luego dudó, e indicó que el programa debe caerse en tiempo de ejecución al intentar acceder a una posición `3/2` que no existe. Con andamiaje indicó que el programa imprime valores espurios. Indicó que los vectores en memoria dinámica se destruyen automáticamente. Terminó abruptamente la ejecución del programa desde la línea `13`. Con andamiaje llegó a resultados incorrectos del programa. Requirió explicaciones de

funcionamiento de la división entera para llegar al resultado correcto.

Participante 10. Recordó correctamente los cuatro segmentos de memoria y su funcionalidad, excepto que ubicó las variables locales en el segmento de datos, separadas de las invocaciones de funciones. Confundió los valores de variables integrales con valores de apuntadores y les otorgó inicialmente la dirección de memoria `0`. Concibió que el valor de un apuntador es otro apuntador en memoria dinámica, el cual apunta al arreglo de datos. Estimó los tamaños de las variables y estructuras a lo largo de su discurso verbal. Interpretó `std::sort` como "revolver los datos" hasta que leyó su documentación oficial. Tuvo dudas sobre el módulo y la división entera, aunque realizó correctamente las operaciones. Indexó el vector iniciando en 1, requirió andamiaje para descubrirlo. No descubrió la fuga de memoria; con andamiaje indicó que la memoria dinámica se libera automáticamente.

Participante 11. Recordó correctamente el propósito de tres segmentos (omitió el segmento de código). Entrelazó correctamente las variables locales con las invocaciones de funciones (**Error! Reference source not found.**). Sin embargo, incluyó también algunas instrucciones de código en este segmento. Necesitó andamiaje para descubrir e iniciar la ejecución desde el `main`. Dudó si el operador `>>` está sobrecargado en `std::cin` para leer variables de tipo `size_t`, lo cual demuestra dominio del tema de sobrecarga de operadores. No creó una variable local para el parámetro `size` al invocar la función `calc`. Copió los valores del arreglo en memoria dinámica al segmento de pila. Interpretó la expresión `arr+size` como suma de enteros y no aritmética de apuntadores. Tras leer la documentación infirió el rango de trabajo de la función. Se equivocó al evaluar las expresiones aritméticas aunque realizó los módulos y divisiones enteras correctamente. Tras solicitarle que volviera a ejecutar el cálculo, lo hizo correctamente. Terminó la ejecución de `calc` sin eliminar su *stack frame*; con mínimo andamiaje lo hizo correctamente. Detectó la fuga de memoria y la corrigió adecuadamente.

Fig 8 Rastro de memoria del programa realizado por el participante 11

IV. Discusión

Por conveniencia, el Cuadro 2 muestra de nuevo la evaluación sumativa al ordenar los conceptos (las filas) descendentemente por su nivel de comprensión. Se agruparon los conceptos en tres categorías indicadas en la columna `Acc`, las cuales indican que los estudiantes evaluados poseen:

- Excelentes modelos mentales de 7 conceptos evaluados (31.8%): condicionales, indexación de arreglos, salida estándar, punto de entrada del programa, ciclo `for`, evaluación de expresiones, y función de biblioteca. Ninguno de estos conceptos es introducido por primera vez en el curso de Programación II.
- Aceptables modelos mentales de 6 conceptos evaluados (27.3%): ciclo *while*, entrada estándar, asignación de memoria dinámica, apuntadores*, aritmética de apuntadores* y arreglos. Los 2 conceptos marcados con asteriscos son introducidos en el curso de Programación II.
- Deficientes modelos mentales de 9 conceptos evaluados (40.9%): retorno de función, invocación de funciones, división entera, segmentación de memoria*, variables locales, fugas de memoria*, liberación de memoria*, parámetros de función, e iteradores*. Los 4 conceptos marcados con asteriscos son introducidos en el curso de Programación II.

Cuadro 2. Calificación subjetiva del profesor para cada concepto por participante ordenado por promedio de comprensión del concepto.

		Participante												
Concepto		1	2	3	4	5	6	7	8	9	10	11	Avg	Acc
15	if	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**1,00**	31,8%
17	arr[i]	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**1,00**	
18	cout	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**1,00**	
2	main	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,9	1,0	0,9	**0,98**	
11	for	1,0	0,8	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	**0,98**	
16	expr	1,0	0,7	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,9	**0,96**	
12	sort	1,0	0,9	0,9	1,0	0,9	0,9	1,0	0,8	1,0	0,9	1,0	**0,94**	
5	while	0,0	1,0	1,0	1,0	0,9	1,0	1,0	1,0	0,9	1,0	1,0	**0,89**	27,3%
6	cin	1,0	1,0	1,0	1,0	0,7	0,9	0,0	0,7	0,7	0,9	1,0	**0,81**	
8	new	1,0	1,0	0,5	0,8	1,0	0,0	1,0	0,9	1,0	0,7	1,0	**0,81**	
10	ptr	1,0	1,0	0,3	0,7	1,0	0,2	0,9	1,0	1,0	0,6	1,0	**0,79**	
14	ptr+i		0,1	1,0	1,0	1,0	0,3		0,5		1,0	0,9	**0,73**	
9	arr	0,6	0,7	0,3	0,7	0,7	0,4	0,5	1,0	1,0	1,0	0,8	**0,70**	
19	return	0,4	1,0	1,0	1,0	0,4	0,2	0,2	0,2	0,5	0,9	0,8	**0,60**	40,9%
3	call	0,0	1,0	0,1	0,5	0,8	1,0	0,8	0,4	0,2	0,6	1,0	**0,58**	
22	/%	1,0	0,0	0,2	1,0	0,1	1,0	1,0	0,1	0,0	0,9	0,9	**0,56**	
1	seg	0,3	0,7	0,4	0,3	0,6	0,3	0,2	0,4	0,6	0,8	0,8	**0,49**	
4	auto	0,2	0,7	0,5	0,3	0,7	0,7	0,2	0,1	0,7	0,3	0,9	**0,48**	
20	leak	0,4	1,0	1,0	1,0	0,0	0,0	0,1	0,0	0,0	0,0	1,0	**0,41**	
21	delete	0,7	1,0	0,7	0,8	0,0	0,0	0,1	0,0	0,0	0,0	1,0	**0,39**	
7	arg	0,0	1,0	0,0	0,1	0,2	0,0	0,1	0,1	0,0	0,8	0,1	**0,22**	
13	itr		0,0	0,0	0,0	0,0	0,0				0,0	0,6	**0,09**	
	Avg	**0,68**	**0,80**	**0,68**	**0,78**	**0,68**	**0,59**	**0,66**	**0,63**	**0,68**	**0,75**	**0,89**	**0,70**	

Los resultados anteriores pueden interpretarse de la siguiente manera. Los estudiantes desarrollaron modelos mentales correctos de aproximadamente un tercio de los conceptos evaluados. Todos ellos fueron introducidos en el curso de Programación I (con Java como lenguaje de programación en el caso de nuestra universidad) y que fueron reforzados en el curso de Programación II (con C++). El método de enseñanza tradicional con contraposición conceptual y rastreo de memoria de programas pareciera

apropiado para este reforzamiento. Sin embargo, un 40.9% de los conceptos evaluados encontraron reflejaron modelos mentales incorrectos. Estos números son alarmantes, pues los conceptos en esta categoría son imprescindibles para lograr los objetivos del curso de Programación II y la adecuada formación del profesional en ciencias de la computación. Menos de un tercio de los conceptos evaluados (27.3%) se encontraron en un estado intermedio entre los dos anteriores.

El método de enseñanza aplicado tampoco fue consistente con el reforzamiento de conceptos. Cerca de la mitad de los modelos deficientes (en la tercera categoría) están asociados a conceptos introducidos en el curso previo de programación. Es decir, los estudiantes han "arrastrado" modelos mentales incorrectos a lo largo de dos cursos de programación en modalidad magistral. Un ejemplo notorio es el concepto de división entera aminorado por una generalización de la división real. Cerca de la mitad de los participantes experimentaron este problema, y mostraron severos estados de angustia al no poder resolver o explicar un fenómeno elemental de la máquina nocional.

Las observaciones obtenidas por el profesor durante los coloquios revelaron graves deficiencias de precisión terminológica y claridad conceptual al dividir el estado del programa en segmentos de memoria (concepto 1), ubicar las variables locales y distinguir los parámetros. Muy pocos estudiantes lograron detectar o corregir la fuga de memoria. Una hipótesis que queda abierta es si esta situación es agravada por el hecho de tomar el curso de programación previo con Java. El lenguaje de programación Java ofrece una sintaxis similar a C++ pero con un servicio de recolección de basura incorporado en el lenguaje, y por tanto carente del operador `delete`.

V. Conclusiones y trabajo futuro

Los resultados encontraron que la población de estudiantes evaluada presentó modelos mentales deficientes de más de un 40% de los conceptos de programación relacionados con la máquina nocional de C++, tras haber tomado un curso impartido en modalidad magistral de Programación II. Aunque un tercio de los conceptos tuvieron modelos mentales excelentes, ninguno corresponde a conceptos introducidos en el curso de Programación II. La conclusión de estos resultados es que la técnica de contraposición conceptual y la técnica de rastreo de memoria de programa aplicadas al método de enseñanza magistral, fueron insuficientes para asegurar la construcción de modelos mentales correctos de la máquina nocional por parte de los aprendices evaluados. Este resultado es de importancia al considerar que la enseñanza magistral mantiene su prevalencia en nuestro país y probablemente en la región latinoamericana.

La inclusión de prácticas más constructivistas en los ambientes de aprendizaje de la programación parece ofrecer resultados más positivos. Las técnica del rastreo de memoria cuando es realizada tanto por el profesor como los estudiantes durante la clase, ha sido considerada como exitosa por Hertz y Jump [5]. La contraposición conceptual ha sido reportada como exitosa al acompañar visualizaciones de programa [8]–[10].

Amplia literatura científica compilada por Sorva et al. señala revela una mezcla de resultados sobre la efectividad de las visualizaciones [11]. Estas herramientas son criticadas por los constructivistas por su pasividad y poco estímulo motivacional. Escaso trabajo se ha encontrado relacionado con la aplicación de principios constructivistas a las visualizaciones de programa [1]. Se conjetura que principios de ludificación [12] podrían ayudar en alcanzar este objetivo.

Agradecimientos

Esta investigación es fruto del proyecto 834-B3-255 del Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC) y la Escuela de Ciencias de la Computación e Informática (ECCI) de la Universidad de Costa Rica (UCR), con apoyo del Ministerio de Ciencia Tecnología y Telecomunicaciones (MICITT).

Referencias

[1] J. Sorva, "Visual Program Simulation in Introductory Programming Education," Aalto University, 2012.

[2] ACM and IEEE Computer Society, "Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science," 2013.

[3] B. du Boulay, T. O'Shea, and J. Monk, "The black box inside the glass box: presenting computing concepts to novices," *Int. J. Man. Mach. Stud.*, vol. 14, no. 3, pp. 237–249, Apr. 1981.

[4] J. Sorva, "Notional machines and introductory programming education," *ACM Trans. Comput. Educ.*, vol. 13, no. 2, pp. 1–31, Jun. 2013.

[5] M. Hertz and M. Jump, "Trace-based teaching in early programming courses," in *Proc. SIGCSE '13*, 2013, p. 561.

[6] A. Vihavainen, J. Airaksinen, and C. Watson, "A systematic review of approaches for teaching introductory programming and their influence on success," in *Proc. ICER '14*, 2014, pp. 19–26.

[7] Luria, Leontiev, and Vigotsky, *Psicología y pedagogía*. Sevilla, España: Ediciones Akal, 1986.

[8] L. Ma, J. Ferguson, M. Roper, and M. Wood, "Investigating the viability of mental models held by novice programmers," in *Proc. SIGCSE '07*, 2007, vol. 39, no. 1, pp. 499–503.

[9] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood, "Using cognitive conflict and visualisation to improve mental models held by novice programmers," in *Proc. SIGCSE'08*, 2008, vol. 40, no. 1, pp. 342–346.

[10] L. Ma, J. Ferguson, M. Roper, I. Ross, and M. Wood, "Improving the mental models held by novice programmers using cognitive conflict and jeliot visualisations," in *Proc. SIGCSE '09*, 2009, vol. 41, no. 3, pp. 166–170.

[11] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," *ACM Trans. Comput. Educ.*, vol. 13, no. 4, pp. 1–64, Nov. 2013.

[12] K. M. Kapp, *The Gamification of Learning and Instruction: Game-based Methods and Strategies for Training and Education*, 1st ed. Pfeiffer, 2012.