Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI3661 - Taller de Lenguajes de Programación I
Enero-Marzo 2011

Programación Orientada a Objetos No Jerárquica

En este proyecto Ud. desarrollará en Scheme una simulación del modelo de almacenamiento de memoria de los lenguajes estilo LISP, incluyendo el método de recolección de basura *mark-and-sweep*. Su implementación debe estar constituida aplicando las técnicas de Programación Orientada a Objetos No Jerárquica provista por la librería Swindle.

Modelo de Almacenamiento de Memoria

Nuestra simulación del modelo de almacenamiento de memoria tendrá varias capas de abstracción, precisamente para que Ud. demuestre sus destrezas en el modelamiento Orientado a Objetos, combinando funcionalidad cruzada para que demuestre sus destrezas en el empleo de la Resolución Estándar de Métodos Genéricos provista por Swindle.

La memoria física

Simularemos la memoria física del computador como una colección ordenada de celdas de tamaño fijo, sobre la cual tenemos acceso directo de lectura y escritura a cualquier posición. Para esto, Ud. debe definir la clase `<memory>` que contenga un atributo `cells` basado en el tipo predefinido `<vector>` de Scheme. El constructor de la clase debe recibir la cantidad de celdas a reservar que necesariamente debe ser una cantidad positiva, reportando el error `"memory must have at least one cell"` en caso contrario.

Las capacidades de la clase `<memory>` incluyen:

- Un método `size` que reporte el tamaño de la memoria en cuestion.
- Un método `fetch` que retorne los contenidos de la $n-$ésima celda de la memoria.
- Un método `store!` que mute los contenidos de la $n-$ésima celda de la memoria con un nuevo contenido suministrado como parámetro.
- Los métodos `fetch` y `store!` deben cumplir con la correspondiente verificación de límites, emitiendo el error `"bus error -- core dumped"` cuando se intente acceder a una posición de memoria física inválida.

Las celdas de memoria

Cada una de las posiciones de memoria física corresponde a un objeto de la clase `<cell>`, que para efectos del proceso de recolección de basura deben contar con un atributo `crap` que indique si la celda es basura o no. En este caso, asegúrese que el método de acceso para lectura se comporte como un predicado, i.e. `crap?`, pero que el método de acceso para escritura se comporte como un mutador, i.e. `crap!`.

El Modelo de Memoria de LISP

Nuestra simulación del Modelo de Memoria de LISP es sumamente simple y se basa en utilizar las celdas de memoria bien sea para almacenar objetos concretos (números, símbolos, etc.) o bien la representación de listas como una pareja cabeza-cola conocida tradicionalmente como *cons-cell*[1].

Queremos que nuestro simulador sea capaz de manejar números, símbolos y listas arbitrariamente anidadas de ellos, así que vamos a requerir una serie de clases que mejoren la funcionalidad de `<cell>`.

Los Símbolos

La clase `<val>` servirá para modelar los valores concretos que pueden almacenarse en una celda, y solamente contará con un atributo `value` que lo contenga.

Las Listas (*cons-cells*)

Un *cons-cell* es una tupla, que tradicionalmente tiene dos componentes denominados `car` y `cdr` que corresponden, respectivamente, al primer elemento de la lista y al resto de la lista. Como un elemento puede ser un valor concreto o bien otra lista, entonces tanto `car` como `cdr` han de ser *apuntadores* bien sea a una celda con un valor concreto (número o símbolo) o bien a una celda con un *cons-cell*.

Así, será necesario que comience por proveer la clase `<ptr>` para modelar apuntadores como referencias a posiciones de memoria *física*, de manera que cualquier instancia de la clase debe tener un atributo `points-to` que corresponde a la posición de memoria *física* a la cual hace referencia. Además, debe contar con un atributo `is-null` que indique si el apuntador es nulo o no. Asegúrese que el método de acceso para lectura se comporte como un predicado, i.e. `is-null?`, pero que el método de acceso para escritura se comporte como un mutador, i.e. `is-null!`. Piense con mucho cuidado cuál es la representación Scheme de una *cons-cell* que tiene **ambos** apuntadores nulos.

Luego, aproveche la clase `<ptr>` para construir la clase `<cons>` con atributos `car` y `cdr`, y que para evitar confusiones con las funciones homónimas de Scheme empleará los nombres `head` y `tail` tanto para los inicializadores como para los métodos de acceso.

Memoria Administrada

Aprovechando la memoria física y el modelo de representación de estructuras recursivas construido hasta ahora, podremos construir la última capa de abstracción en nuestra simulación. La Memoria Administrada constituye, en efecto, un pequeño manejador de memoria que se encargará de gestionar la reserva y recuperación de memoria.

La memoria administrada es una especialización de la memoria física, implantada en la clase `<managed-memory>`, y que incorpora dos elementos mínimos para la gestión del espacio: un atributo `freelist` para modelar la lista de bloques libres, y un atributo `roots` para modelar la lista de objetos presentes en la memoria.

El atributo `freelist` no es más que una lista de números enteros que incluye aquellos bloques de memoria física disponibles para almacenamiento; como todos los elementos de memoria son del mismo tamaño, tanto la búsqueda como la incorporación de bloques libres a la lista puede hacerse por el principio.

El atributo `roots` es una lista de asociación que indica el nombre de un objeto y la posición de memoria donde se encuentra su primera celda; esto es, si en la memoria hubiese un objeto denominado `foo` almacenado en la posición `42`, y otro objeto denominado `bar` en la posición `69`, el atributo `roots` contendría `((foo 42) (bar 69))`.

Con esta infraestructura, será necesario proveer tres métodos de alto nivel para nuestra simulación:

- El método `store-object!` que opera sobre un objeto `<managed-memory>`, un símbolo Scheme para identificar el objeto almacenado y cualquier valor Scheme sujeto a las restricciones descritas anteriormente. Este método debe utilizar tanta memoria simulada como sea necesario para representar el valor Scheme usando objetos `<cons>` y `<val>` según sea necesario. Esto es, una invocación como

  ```
  (store-object! memory 'foo '(1 bar (2 3)))
  ```

  debe

  1. Buscar en la lista de bloques libres tanta memoria como sea necesaria para representar la lista `(1 bar (2 3))` usando `<cons>` y `<val>`. Si no hay suficientes, debe recoger basura con la intención de recuperar espacio libre; Ud. debe implantar el algoritmo *mark and sweep* inocente[2]. Si aún después no hay suficientes bloques libres, debe reportar el error `"out of memory"`.
  2. Suponiendo que fue capaz de representar el objeto, entonces debe registrar en la tabla `roots` la presencia del objeto de nombre `foo` indicando su primera posición de memoria, y retornar esta posición como evidencia de éxito. Note que los nombres de los objetos son únicos, así que si se registra un segundo objeto con el mismo nombre, se reemplaza la pareja en la lista de raíces. Recuperar el espacio de la reemplazada es trabajo del recolector de basura cuando sea necesario.

- El método `fetch-object` que opera sobre un objeto `<managed-memory>` y un símbolo Scheme, produciendo como resultado el objeto Scheme representado. Si el objeto no existe, debe reportar el error `"undefined symbol"`. Esto es, un ejemplo de uso podría ser

  ```
  > (store-object! memory 'foo 42)
  1
  > (store-object! memory 'foo '(foo bar (42 baz)))
  2
  > (fetch-object memory 'foo)
  (foo bar (42 baz))
  > (fetch-object memory 'bar)
  undefined symbol
  ```

- El método `forget-object!` que opera sobre un objeto `<managed-memory>` y un símbolo Scheme, tal que el objeto en cuestión sea eliminado de la tabla de raíces. Si el objeto no existe, debe reportar el error `"undefined symbol"`.

Traza y Estadística

Una vez completada la funcionalidad de `<managed-memory>` Ud. deberá agregar funcionalidad de trazado y estadística *sin* cambiar los métodos primarios para todo el resto de la implantación. Esto va a requerir que emplee métodos `:before`, `:after` o `:around` según le parezca conveniente en los diversos casos. Así, es necesario que Ud. provea las definiciones y capacidades para varias clases adicionales.

- `<traced-memory>` especializa a `<managed-memory>` y tiene la capacidad de reportar todas y cada una de las operaciones de almacenamiento y búsqueda, manteniendo un acumulado total. Esto es:
  - Cada operación `store-object!` debe indicarse como `"T: attempting store-object! for foo"` donde `foo` es el símbolo particular a almacenar. Cada uno de los pasos del proceso debe ser anunciado, i.e. `"T: searching for a free page"`, `"T: storing on page 42"`, `"T: collecting garbage"`, `"T: updating root list",` además de las operaciones sobre memoria *física* hasta culminar la inserción.
  - Cada operación `fetch-object` debe indicarse como `"T: attempting fetch-object for foo"` donde `foo` es el nombre del símbolo particular a buscar. Cada uno de los pasos del proceso debe ser anunciado, i.e. `"T: looking for foo in root list"`, `"T: fetching from page 42"`, además de las operaciones sobre memoria *física* hasta culminar la búsqueda.
  - Cada operación `forget-object!` debe indicarse de manera similar a `fetch-object`.
  - Al culminar un `store-object!`, `fetch-object` o `forget-object!` debe reportarse la cantidad total de operaciones según su tipo `"T: 32 store-object! / 42 fetch-object / 5 forget-object! / 437 store! / 1563 fetch"`.
- `<profiled-memory>` especializa a `<managed-memory>` y tiene la capacidad de reportar la cantidad de memoria libre y usada en todo momento así como la cantidad de objetos almacenados después de cada operación según su tipo, incluyendo información sobre la actividad del recolector de basura. Esto es:
  - Después de cada operación, debe indicarse la cantidad de memoria diferenciando su uso, i.e. `"P: 100 cells / 28 used (16 values / 12 cons) - 72% free"`.
  - Después de cada corrida del recolector de basura, debe indicarse la cantidad de memoria recuperada en la operación, i.e. `"P: reclaimed 42 cells -- free list holds 67 cells"`.
  - Después de cada `store-object!`, debe reportarse la modificación a la lista de libres y de raíces, i.e. `"P: free list holds 54 cells -- root list stores 17 symbols"`.
  - Después de cada `forget-object!`, debe reportarse la modificación a la lista de raíces, i.e. `"P: root lists stores 16 symbols"`.
- `<traced-and-profiled-memory>` debe combinar la funcionalidad de `<traced-memory>` y `<profiled-memory>`, incorporando además un resumen global de inserciones, reemplazos (cuando se hace `store-object!` de un objeto que ya existe) y eliminaciones.

Finalmente, debe proveer un método `dump-memory` que debe mostrar en pantalla los contenidos actuales de la memoria en un formato similar a

```
Size: 5 cells
[0] empty
[1] value: 42
[2] cons:  head points-to 3 / tail null
[3] value: 42
[4] cons:  head null / tail null
Free List: 1 0
Root List:
- foo stored in 2
- bar stored in 3
```

No hay ningún orden implícito en la lista de libres ni en la lista de raíces, simplemente muéstrelas tal como están en ese momento. Note que al invocar `dump-memory` **no** deben interferir las operaciones de traza ni perfilado.

Esto va a requerir, entre otras cosas, que Ud. implante métodos `print-object` para todas las clases involucradas.

Entrega de la Implantación

Ud. debe entregar un archivo `.tar.gz` o `.tar.bz2` (**no** puede ser `.rar`, ni `.zip`) que al expandirse genere un directorio con el nombre de su grupo (e.g. `G42`) dentro del cual encontrar **solamente**:

- El archivo `gc.scm` conteniendo el código fuente Scheme para implantar todas las clases, métodos y funciones auxiliares que considere menester.
- Se evaluará el correcto estilo de programación:
  - Indentación adecuada y consistente en *cualquier* editor – la excusa "profe, en el mío se ve bien" es inaceptable.
  - Las clases, métodos y funciones deben estar completa y correctamente documentado. Es **inaceptable** que la documentación tenga errores ortográficos.
- **Fecha de Entrega.** Viernes 2011-04-01 (Semana 12) hasta las 17:59 VET
- **Valor de Evaluación.** Veinte (20) puntos.

Referencias

[1] Celdas CONS según Wikipedia
http://en.wikipedia.org/wiki/Cons

[2] Recolección de Basura según Wikipedia
http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)