SchemeDBC: Conexión a Bases de Datos Relacionales desde el Lenguaje Scheme

M. en C. Ariel Ortiz Ramírez
aortiz@campus.cem.itesm.mx
ITESM Campus Estado de México
Carr. Lago de Guadalupe Km. 3.5 Atizapán, Méx.
México

Resumen

SchemeDBC es un API de Scheme que soporta enunciados de SQL que permiten acceder y manipular datos y objetos complejos contenidos en bases de datos relacionales.

1 Introducción

El lenguaje de programación Scheme es una herramienta muy flexible que permite experimentar y desarrollar sobre conceptos avanzados de computación. Scheme soporta un consistente y poderoso esquema de manipulación de objetos en forma de listas, símbolos, procedimientos y otros tipos de datos sin paralelo en la gran mayoría de los lenguajes de programación contemporáneos. Esta gran versatilidad ha dado pie a que Scheme sea utilizado exitosamente en aplicaciones tan divergentes como son sistemas operativos, agentes inteligentes, análisis financiero, gráficas y construcción de traductores de lenguajes. Sin embargo, en contraposición a las múltiples bondades con las que cuenta el lenguaje, su soporte para datos persistentes resulta un tanto restringido.

Para desarrollar las aplicaciones que actualmente requiere la industria e investigación, resulta evidente que los lenguajes computacionales deben contar con mecanismos sofisticados de persistencia que permitan un eficiente almacenamiento y recuperación de las abstracciones directamente modeladas por el ambiente de programación.

En este artículo se presenta un modelo que extiende la semántica de Scheme a través de una biblioteca de procedimientos diseñados para acceder a bases de datos relacionales mediante enunciados de SQL. Se busca proporcionar una funcionalidad que permita la manipulación directa de todo los tipos de datos soportados por Scheme, superando de esta forma las limitaciones propias del lenguaje relacionadas a la persistencia.

2 Antecedentes de Scheme

La presente definición del lenguaje Scheme se encuentra en el documento titulado "The Revised[5] Report on the Algorithmic Language Scheme" [Kelsey, 1998]. A continuación se detallan tres aspectos importantes de la descripción del lenguaje que permitirán comprender las limitaciones inherentes asociados a su esquema de almacenamiento y recuperación.

2.1 Puertos de Entrada y Salida

En lo que respecta a entrada y salida, la definición del lenguaje documenta un esquema de *puertos*, el cual se parece un poco a los flujos o *streams* en lenguajes como C o C++. El siguiente ejemplo demuestra como escribir una lista con tres números a un archivo utilizando los procedimientos `call-with-output-file` y `write`:

```
(call-with-output-file
  "datos.txt"
  (lambda (puerto)
    (write '(10 24 -34) puerto)))
```

La lista del ejemplo anterior se almacena en el archivo `datos.txt` en forma de texto plano, la cual puede ser ahora leída utilizando los procedimientos `call-with-input-file` y `read`:

```
(call-with-input-file
  "datos.txt"
  (lambda (puerto)
    (read puerto))) ⇒ (10 24 -34)
```

Este esquema de puertos tiene una par de condicionantes importantes a mencionar:

- En un puerto se puede almacenar cualquier cantidad de datos, pero sólo de manera secuencial. Los datos no pueden ser accedidos de manera directa o aleatoria.
- En los puertos sólo se pueden almacenar datos que cuenten con una *representación externa*. Estos tipos de datos son: booleanos, toda la pirámide de números, símbolos, caracteres, cadenas de caracteres, listas, vectores y el objeto nulo. En particular, las *cerraduras léxicas*, encargadas de encapsular los procedimientos, no cuentan con representación externa, por lo que no pueden ser almacenados usando este esquema.

2.2 Cerraduras Léxicas

A diferencia de la mayoría de los lenguajes de programación, en Scheme los procedimientos son creados de manera dinámica a tiempo de corrida. Cuando un procedimiento es creado, a partir de la evaluación de una

expresión lambda, se produce una cerradura léxica, la cual encapsula tres elementos: código, parámetros formales y ambiente, el cual consiste de todas las ligas de las variables que estaban visibles al momento de creación del procedimiento [Friedman, et *al.*, 1992].

Las cerraduras léxicas pueden responder al paso de mensajes y mantener/modificar un estado local. Por ejemplo, el siguiente fragmento de código contiene una cerradura léxica que representa un contador que empieza con un valor inicial dado:

```
(define crea-contador
  (lambda (inicial)
    (lambda (mensaje)
      (case mensaje
        ((inc) (set! inicial
                     (+ inicial 1))
               inicial)
        ((cero) (set! inicial 0)
                inicial)))))

(define c (crea-contador 10))
```

Cada vez que se invoca el procedimiento `crea-contador` se crea una nueva cerradura léxica, correspondiente a la expresión lambda más interna. La figura 1 muestra la cerradura léxica ligada a la variable `c` del ejemplo anterior.

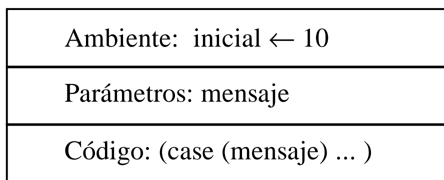

Figura 1: Cerradura Léxica ligada a la variable `c`.

Esta cerradura está diseñada para poder responder a dos mensajes: uno de incremento y otro de reiniciación; dichos mensajes modifican el estado interno de la cerradura léxica:

```
(c 'inc)    ⇒ 11
(c 'inc)    ⇒ 12
(c 'inc)    ⇒ 13
(c 'cero)   ⇒ 0
(c 'inc)    ⇒ 1
```

Cada vez que se invoca el procedimiento `crea-contador` se produce una nueva cerradura léxica que cuenta con un ambiente independiente de cualquier otra cerradura. Por tanto, lo que aquí se tiene es un esquema para generar "instancias" del tipo de dato abstracto "contador". Este esquema de cerraduras léxicas es el fundamento para la programación basada en objetos en Scheme. Sin embargo, como ya se mencionó, las cerraduras léxicas no tienen representación externa, significando que no existe un soporte directo para objetos persistentes utilizando puertos en Scheme.

2. 3 Estructuras Complejas

Una lista en Scheme puede tener una estructura irregular que evite que pueda ser escrita en un puerto. Por ejemplo, el siguiente código produce una lista circular que generará algún tipo de problema, posiblemente un desbordamiento de pila, cuando se intente imprimir:

```
(call-with-output-file
  "circular.txt"
  (lambda (puerto)
    (let ((lista (list 'a)))
      (set-cdr! lista lista)
      (write lista puerto))))
```

Otro problema que surge es cuando una lista está conformada por sublistas que en algún momento se comparten internamente. Por ejemplo:

```
(call-with-output-file
       "sublistas.txt"
       (lambda (puerto)
         (let* ((una-lista '(a b))
                (otra-lista
                  (list una-lista
                        una-lista)))
           (write otra-lista puerto))))
```

La lista original que se está escribiendo al archivo tiene la forma de la figura 2. Sin embargo, cuando sea leída nuevamente a memoria, el sistema no tiene forma de saber que una misma lista se escribió dos veces, por lo que la estructura que se obtendrá será la de la figura 3. Es evidente que no se está regenerando correctamente la estructura de la lista original, lo que nos lleva a concluir que Scheme no soporta correctamente persistencia de listas con estructuras complejas.

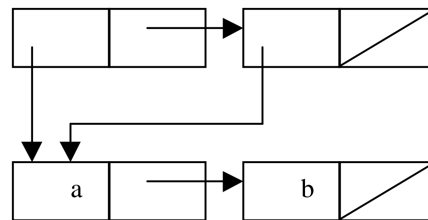

Figura 2: Lista antes de ser escrita a un archivo.

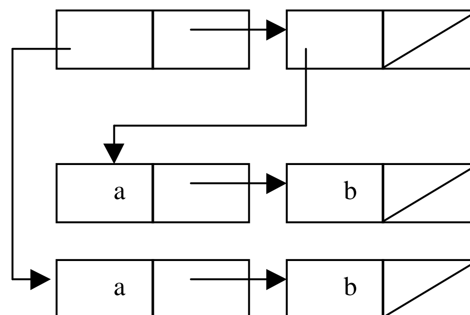

Figura 3: Lista después de ser leída de un archivo.

3 SchemeDBC

SchemeDBC es un API que permite acceder a bases de datos relacionales a través de enunciados de SQL incrustados en el código de Scheme.

La implementación actual de SchemeDBC se construyó sobre equipos Sun Ultra 5 corriendo bajo SunOS 5.6. El ambiente de programación utilizado fue Chez Scheme 6.1 de Cadence Research Systems [Dybvig, 1998], accediendo a través de TCP/IP al servidor de base de datos Mini SQL 2.0.1 de Hughes Technologies [Hughes, 1998], el cual soporta un subconjunto significativo de ANSI SQL. Estas herramientas se seleccionaron por su disponibilidad y facilidad de integración, sin embargo, SchemeDBC potencialmente puede ser implementado sin demasiada dificultad en otras plataformas.

Además de incorporar el acceso a bases de datos relacionales con SQL, SchemeDBC se enfoca a superar las limitaciones expuestas en la sección 2 respecto a objetos persistentes de Scheme.

3.1 Arquitectura de SchemeDBC

La figura 4 muestra esquemáticamente la arquitectura de la presente implementación de SchemeDBC.

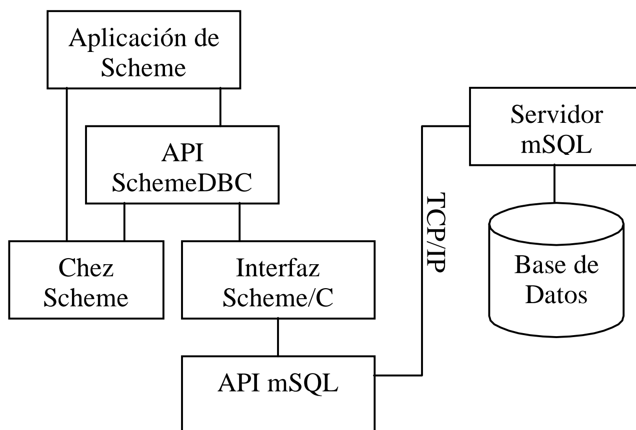

Figura 4: Arquitectura de SchemeDBC

El servidor mSQL es un demonio (*deamon*) que proporciona los servicios necesarios para que cualquier cliente tenga acceso a la base de datos. mSQL proporciona un API para lenguaje C con el que se pueden escribir los programas clientes.

La interfaz Scheme/C, descrita a más detalle en la sección 3.2, es una biblioteca de enlace dinámico escrita en lenguaje C que sirve como puente entre el código de Scheme y el API de mSQL. El ambiente de programación Chez Scheme carga la biblioteca dinámica y la añade a su montículo (*heap*). El módulo del API de SchemeDBC incluye las definiciones necesarias para la importación de las rutinas externas contenidas en la biblioteca dinámica, así como los procedimientos que conforman el API propiamente dicho. Una aplicación de Scheme se convierte en un cliente de mSQL al utilizar el API de SchemeDBC.

3.2 Invocando el API de mSQL desde Scheme

Chez Scheme permite cargar y ligar dinámicamente código binario que: 1) utilice la convención de llamadas de lenguaje C, y 2) esté contenido en una biblioteca de enlace dinámico (*shared object* de UNIX). El API de mSQL se proporciona en forma de una biblioteca estática, y para utilizarla se requieren elementos de lenguajes de bajo nivel que no tienen una correspondencia directa en lenguaje Scheme. Para contrarrestar estas limitaciones, SchemeDBC cuenta con un módulo de interfaz que: a) se proporciona en forma binaria como biblioteca de enlace dinámico, y b) contiene procedimientos en lenguaje C que invocan el API de mSQL y que su vez pueden ser invocados directamente por programas escritos en Scheme. Estos procedimientos intermediarios esconden el uso de apuntadores de lenguaje C, sustituyéndolos en su lugar por controladores (*handles*) que son administrados a través de una tabla de asociación interna.

Los procedimientos del módulo de interfaz solamente pueden recibir y devolver ciertos tipos compatibles con Scheme: `void`, `char`, `short`, `int`, `long`, `float`, `double` y `char*`. Esta restricción no es tan grave como pudiera parecer, ya que el API de SchemeDBC proporciona un esquema de manipulación de datos de mucho más alto nivel. Por ejemplo, a la hora de solicitar una fila, el módulo de interfaz solamente puede leer un campo a la vez, sin embargo SchemeDBC esconde este detalle y proporciona al usuario una funcionalidad en donde la fila completa se presenta como una lista.

A continuación se muestra la manera de definir e importar rutinas externas en Scheme. Se ejemplifica mediante la rutina que se encarga de comunicar los mensajes de error que reporta el servidor de mSQL en un momento dado. El API de mSQL define una variable global llamada `msqlErrMsg` de tipo `char*` en el archivo `msql.h`. El módulo de interfaz `schemedbc.c` tiene entonces una rutina de la siguiente forma que permite devolver el valor de dicha variable:

```
#include "msql.h"

char * msql_errorMessage(void) {
    return msqlErrMsg;
}
```

Para construir el archivo de biblioteca de enlace dinámico bajo SunOS, se utilizó el compilador `gcc` de la siguiente manera:

```
gcc schemedbc.c -o schemedbc.so -fPIC -G
    -lmsql -lsocket -lnsl
```

La opción `-G` informa al compilador que debe producir una biblioteca dinámica, mientras que la opción `-fPIC` sirve para indicarle que el código de la biblioteca debe ser independiente de posición. Las opciones que inician con `-l` indican los nombres de las bibliotecas con las que se necesita ligar el módulo: la biblioteca de mSQL y las bibliotecas que proporcionan las funciones de red TCP/IP.

El módulo que implementa el API de SchemeDBC carga la biblioteca dinámica mediante el siguiente comando de Scheme:

```
(load-shared-object "./schemedbc.so")
```

Posteriormente, se declara el procedimiento en cuestión como externo:

```
(define msql-error-message
  (foreign-procedure
    "msql_errorMessage"
    ()
    string))
```

En esta caso se está definiendo a la variable `msql-error-message` como un procedimiento foráneo, cuyo nombre en la biblioteca dinámica es `"msql_errorMessage"`. Adicionalmente, se establece que dicho procedimiento no recibe argumentos y devuelve una cadena de caracteres como resultado. Una vez hecha la definición, la forma de usar el recién importado procedimiento es la siguiente:

```
(msql-error-message) ⇒ ""
```

Una cadena vacía se devuelve cuando no se ha generado ningún mensaje de error por parte del servidor de mSQL.

3.3 SQL y Scheme

A continuación se muestra un fragmento de código de SchemeDBC que establece una conexión con el servidor de base de datos, abre una base de datos, crea una tabla llamada `alumnos` e inserta tres filas:

```
(call-with-database
  "dbserver" "datos"
  (lambda (db)
    (database-update db "create table
      alumnos (matricula int, nombre
      char(30), carrera char(4))")
    (database-update db "insert into
      alumnos values (441523, 'Juan
      Pérez', 'LAE')")
    (database-update db "insert into
      alumnos values (441823, 'María
      López', 'IEC')")
    (database-update db "insert into
      alumnos values (450012, 'Eva
      Mata', 'LAE')")))
```

El procedimiento `call-with-database` requiere tres argumentos: una cadena de caracteres que contenga el nombre o dirección IP en donde se encuentra el *host* en el que está corriendo el servidor de mSQL, una cadena de caracteres que contenga el nombre de la base de datos a la que se desea acceder, y un procedimiento *proc* de un parámetro. `Call-with-database` establece la conexión con el servidor y selecciona la base de datos indicada, para posteriormente invocar el procedimiento *proc* enviándole como argumento un controlador con el que se podrán realizar las operaciones sobre la base de datos. Una vez que *proc* termina, se cierra la conexión al servidor. El resultado que devuelve `call-with-database` es el mismo que devuelve *proc*.

El procedimiento `database-update` se utiliza para llevar a cabo operaciones que modifican la base de datos. Para el caso de mSQL, son los enunciados que contienen las cláusulas `create`, `drop`, `insert`, `delete` y `update`.

Para llevar a cabo consultas mediante la cláusula `select`, se debe utilizar el procedimiento `database-query`. Este procedimiento devuelve un objeto "result", al cual se le puede solicitar que indique qué campos y qué filas resultaron de la consulta. Por ejemplo, para determinar los campos de la tabla `alumnos`, se tiene el siguiente código:

```
(call-with-database
  "dbserver" "datos"
  (lambda (db)
    (result-fields (database-query db
      "select * from alumnos"))))
⇒ (("matricula" "alumnos" int 4 ())
   ("nombre" "alumnos" char 30 ())
   ("carrera" "alumnos" char 4 ()))
```

El resultado final obtenido es una lista conformada por sublistas, cada una de las cuales representa la información de uno de los campos conteniendo la siguiente información: el nombre del campo (cadena de caracteres), el nombre de la tabla (cadena de caracteres), el tipo de campo (símbolo), la longitud del campo (número) y una lista con cero, uno o dos atributos. Los atributos son símbolos cuya interpretación es la siguiente:

- `not-null` si el campo no debe estar vacío; y
- `unique` si el campo es parte de un índice, razón por la cual no debe repetirse.

Así como el procedimiento `result-fields` sirve para obtener los campos o columnas resultantes de una consulta, el procedimiento `result-rows` se utiliza para obtener las filas. Este último procedimiento devuelve un flujo (*stream*) con todas las filas de la consulta. Un flujo de Scheme, tal como lo define Friedman et *al.* [1992], es una secuencia similar en su forma de manipulación a una lista pero con la diferencia de que cada uno de sus elementos se calcula justo hasta el momento en que es realmente requerido. Lo anterior se logra mediante un proceso de evaluación postergada (*lazy evaluation*). Para avanzar secuencialmente entre las filas resultantes de una consulta, basta utilizar los procedimientos `head` y `tail`, los cuales son equivalentes a `car` y `cdr` para listas convencionales.

A continuación se muestra un ejemplo que realiza una consulta sobre la base de datos y devuelve la segunda fila (asumiendo que ésta existe) de un conjunto de alumnos que estudian la carrera de `'LAE'`:

```
(call-with-database
  "dbserver" "datos"
  (lambda (db)
    (head (tail (result-rows
      (database-query db
        "select * from alumnos where
         carrera = 'LAE'"))))))
⇒ (450012 "Eva Mata" "LAE")
```

Hay que notar varias cosas aquí:

- El procedimiento `head` devuelve una lista la cual representa toda una fila de la consulta.
- Dicha lista está compuesta por cadenas de caracteres y números. Cualquier campo que esté definido en la base de datos como de tipo numérico (`int`, `uint`, `real` y `money` para mSQL) se convierte automáticamente a un número de Scheme. Todos los demás campos se preservan como cadenas de caracteres.
- No se requiere un comando separado para avanzar a la siguiente fila, sino que al aplicar el procedimiento `tail` se obtienen automáticamente la siguiente fila del servidor de base de datos o de un caché interno según se vaya necesitando.
- Al aplicar `head` al resultado devuelto de `tail` se obtiene el segundo elemento de un flujo.

Cualquier dato de Scheme puede ser convertido, con el cuidado debido, a una cadena de caracteres para poder ser almacenado en un campo de tipo `char` o `text` dentro de una base de datos de mSQL. Para facilitar el proceso, SchemeDBC cuenta con un procedimiento llamado `make-query-string`. El siguiente ejemplo muestra el uso de dicho procedimiento en el contexto de una actualización a partir de datos contenidos en dos variables arbitrarias `nom` y `mat`:

```
(call-with-database
  "dbserver" "datos"
  (lambda (db)
    (let* ((nom "Pedro Pérez")
           (mat 441600)
           (consulta
             (make-query-string
               "update alumnos set
                nombre = ? where
                matricula < ?"
               nom
               mat)))
      (database-update db consulta))))
```

Cada aparición del carácter `?` en la cadena original es reemplazada por la conversión a cadena de caracteres de los sucesivos argumentos del procedimiento `make-query-string`. Dicha conversión incluye algunos caracteres de control que permiten la correcta recuperación posterior de los datos.

3.4 Cerraduras Léxicas en SchemeDBC

SchemeDBC permite el almacenamiento de cerraduras léxicas usando el mismo procedimiento `make-query-string` mencionado en la sección 3.3. Para este propósito, el procedimiento no estándar `inspect/object` de Chez Scheme se utilizó para generar una representación externa en forma de lista. Dicha representación externa está diseñada para que cuando se le aplique el procedimiento primitivo `eval` se obtenga una imagen fidedigna de la cerradura léxica original.

Asumiendo que se tiene disponible el procedimiento `crea-contador` introducido de la sección 2.2, la siguiente porción de código muestra como se pueden almacenar objetos propios de Scheme en una base de datos:

```
(let* ((alfa (crea-contador 20))
       (beta (crea-contador 10))
       (sigma (list car beta cdr)))
  (alfa 'inc)
  (beta 'cero)
  (call-with-database
    "dbserver" "datos"
    (lambda (db)
      (database-update db "create table
        objetos (nombre char(20), objeto
        text(1024))")
      (database-update db
        (make-query-string "insert into
          objetos values ('Alfa', ?)"
          alfa))
      (database-update db
        (make-query-string "insert into
          objetos values ('Sigma', ?)"
          sigma)))))
```

El ejemplo crea una nueva tabla con dos campos, `nombre` y `objeto`, e inserta dos filas. La primera fila contiene la cerradura léxica asociada a la variable local `alfa`, mientras que la segunda fila contiene una lista conformada por tres procedimientos: el procedimiento primitivo `car`, la cerradura léxica asociada a la variable local `beta` y el procedimiento primitivo `cdr`.

El siguiente código muestra como se recupera la cerradura léxica asociada al nombre `'Alfa'` de la tabla recién creada:

```
(define alfa-recuperada
  (call-with-database
    "dbserver" "datos"
    (lambda (db)
      (get-object (head (result-rows
        (database-query db "select
          objeto from objetos where
          nombre = 'Alfa'")))
        0))))

(alfa-recuperada 'inc) ⇒ 22
```

El procedimiento `get-object` recibe dos argumentos: una lista que representa una fila de la consulta realizada y un número entero no negativo que representa la posición del campo (comenzando en cero) que se desea obtener. `Get-object` convierte el dato de la posición indicada a un objeto de Scheme.

De manera similar, se puede obtener y utilizar el procedimiento primitivo `cdr` contenido en la tercera posición de la lista asociada al nombre `'Sigma'` de la tabla `objetos`:

```
((caddr
   (call-with-database
     "dbserver" "datos"
     (lambda (db)
       (get-object (head (result-rows
         (database-query db "select
           objeto from objetos where
           nombre = 'Sigma'")))
         0))))
 '(uno dos tres))
⇒ (dos tres)
```

3.4 Estructuras Complejas en SchemeDBC

Para los dos tipos de listas con estructuras complejas mencionadas en la sección 2.3, SchemeDBC utiliza un algoritmo que detecta estructuras compartidas y produce una representación externa con notación especial al momento de convertir las listas a cadenas de caracteres, previo a su almacenamiento. Dicha notación emplea una sintaxis del tipo `"#`*`n`*`=`*`obj`*`"`, donde *`n`* es un número entero no negativo y *`obj`* es la representación externa de un objeto, para etiquetar la primera ocurrencia de *`obj`* en la salida. La sintaxis `"#`*`n`*`#"` se usa para referirse al objeto etiquetado por *`n`* en las ocurrencias sucesivas. Por ejemplo, dado el siguiente código:

```
(define a (list 1 2 3))
(set-cdr! (cddr a) a)
```

La representación externa de la lista asociada a la variable `a` sería:

```
#0=(1 2 3 . #0#)
```

Para el código siguiente:

```
(define b '(1 2 3))
(define c (list b (list b b)))
```

La lista asociada a `c` tendría la siguiente representación externa:

```
(#0=(1 2 3) (#0# #0#))
```

La notación de representación externa aquí expuesta permite leer y reconstruir a partir de una cadena de caracteres la estructura original de cualquier lista.

4 Trabajos Relacionados

Varios esquemas que incorporan acceso a bases de datos relacionales a través de SQL con Scheme han sido propuestos e implementados, por ejemplo Kiselyov [1998] y Lewis [1997]. Sin embargo, ningún sistema conocido por el autor ha incorporado un esquema que permita fácilmente almacenar y recuperar objetos de primera clase de Scheme, incluyendo cerraduras léxicas (procedimientos), símbolos, listas y vectores heterogéneos, etc.

5 Conclusiones

SchemeDBC corrige las limitaciones de Scheme en cuanto a manejo de objetos persistentes. SchemeDBC utiliza SQL para acceder y manipular la información contenida en un sistema manejador de base de datos. Esto tiene importantes ventajas sobre el manejo de archivos secuenciales convencionales: accesos más eficientes, seguridad, integridad, control de concurrencia, etc. Así mismo, SchemeDBC permite escribir y leer todos los tipos de datos de Scheme, incluyendo listas con estructuras complejas y procedimientos con ambiente local.

La implementación actual de SchemeDBC ha sido exitosamente utilizada para desarrollar guiones CGI escritos en el lenguaje Scheme que requieren de acceso a información contenida en bases de datos.

A corto plazo se espera portar SchemeDBC a la plataforma Win32. En este caso se tiene pensado utilizar el estándar de Microsoft ODBC para acceder a las bases de datos desde Windows.

Los archivos fuente de SchemeDBC pueden ser obtenidos del siguiente URL:

http://webdia.cem.itesm.mx/dia/ac/aortiz/sdbc.tar.gz

Referencias

[Dybvig, 1998] R. Kent Dybvig. *Chez Scheme User's Guide.* Cadence Research Systems, 1998.

[Friedman, et *al.*, 1992] Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, 1992.

[Hughes, 1998] David J. Hughes and Brian Jepson. *Official Guide to Mini SQL 2.0.* John Wiley & Sons, 1998.

[Kelsey, 1998] Richard Kelsey, William Clinger and Jonathan Rees, eds. *Revised^5 Report on the Algorthmic Language Scheme.* 1998. http://www-swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps

[Kiselyov, 1998] Oleg Kiselyov. *The Most Primitive and Nearly Universal Database Interface.* 1998. http://www.lh.com/~oleg/ftp/Scheme/index.html#db.scripting

[Lewis, 1997] Bruce Lewis. *The Beautiful Report Language.* 1997. http://w3.mit.edu/wwwdev/brl/index.html