Scheme-PG

David J. Neu
djneu at acm dot org

Version 0.2.0
September 16, 2004

1 Introduction

Scheme-PG is a MzScheme extension and set of associated Scheme procedures and macros that combine to provide a simple yet powerful interface between PLT Scheme and the PostgreSQL database management system. It strives to provide programmers with an interface to PostgreSQL whose syntax is natural for Scheme programmers, supports classic Scheme programming idioms, and shields them from the details of interacting with PostgreSQL and creating SQL statements.

The core Scheme-PG functionality is implemented as a MzScheme extension, that utilizes PostgreSQL's libpq C library. This extension provides users with the ability to open and close connections to a PostgreSQL database server, execute arbitrary SQL statements, execute SQL SELECT and FETCH statements that return rows into random access data structures, to access the data in these data structures, and to manage large objects.

Scheme-PG extends the core functionality to include access to procedures and data structures that allow programmers to access data using Scheme procedures that implement cursors and stream abstractions.

Finally, taking a cue from Scheme-QL, Scheme-PG provides a set of procedures and macros that allow programmers to create SQL statements using a little language implemented in Scheme.

2 Getting Started

In order to use Scheme-PG, PLT Scheme and PostgreSQL must be installed. The current compile and installation scripts have been tested on FreeBSD 5.2.1 with the PLT Scheme 208, installed in /usr/local/plt/ and with the PostgreSQL 7.4.5 port. They may need to be modified for other configurations. Specifically

(system-library-subpath), which is hardcoded to i386-freebsd will need to be modified for other systems.

1. Download the current tgz file from http://scheme-pg.sourceforge.net/ into a temporary directory.
2. Unpack the tgz file.
3. Move into the scheme-pg directory.
4. Run ./compile.sh
5. Run ./link.sh
6. As root run ./install.sh, to create a scheme-pg directory structure under /usr/local/plt/collect and to copy the the Scheme-PG into this structure.

An extensive set of example programs can be found in the included examples.ss file.

3 Connections

Prior to accessing a PostgreSQL database it is necessary to obtain a connection to the PostgreSQL server. The functions described in this section support the opening, closing and management of these connections.

(connection-open aconnection-string**)** PROCEDURE
connection-open: string → c-pointer
The connection-open procedure essentially wraps the libpq PQconnectdb function. It accepts a string that contains the information necessary to establish a connection with the PostgreSQL server and returns a c-pointer to a libpq PGconn C structure. The PGconn structure will be simply referred to as a connection in Scheme. The exact format of aconnection-string can be found in the libpq documentation of the PQconnectdb function.

(connection? aconnection**)** PROCEDURE
connection?: connection → boolean
The connection? procedure returns true if aconnection is a connection, i.e. a c-pointer to a PGconn structure, and false otherwise.

(connection-close aconnection**)** PROCEDURE
connection-close: connection → void
The connection-close procedure essentially wraps the libpq PQfinish function which closes the connection to the PostgreSQL server and frees any memory utilized by the the associated PQconn structure pointed to by aconnection.

4 SQL Statement Execution

(execute-sql aconnection astatement**)** PROCEDURE
execute-sql: connection string → integer
The execute-sql procedure accepts a connection and a string containing an SQL statement. It returns the number of rows affected by the execution of the SQL statement. It supports the execution of SQL statements that do not return query results. Attempts to execute SQL statements such as SELECT and FETCH that do return query results will raise an exception.

5 Results

The PostgreSQL libpq library includes a C structure called PGresult that, in conjunction with a family of associated libpq functions, provide random access to the rows returned by execution of an SQL SELECT or FETCH statement. In addition, information such as the number of rows returned and metadata including column names, data types, etc. is available. It is important to note that the PGresult structure does not maintain any state information about the "current position" in the set of rows. The PGresult structure will be simply referred to as a result in Scheme. Programmers will probably not want to use results directly, but will likely want to make use of the abstraction provided by cursors (see §6) or streams (see §7) so that they can easily switch the underlying implementation as an application's requirements evolve. The functions described in this section support the opening, closing and management of results.

(result-open aconnection aselect**)** PROCEDURE
result-open: connection string → c-pointer
The result-open procedure accepts a connection and a SQL SELECT or SQL FETCH statement as a string, and returns a connection object.

(result? aresult**)** PROCEDURE
result?: result → boolean
The result? procedure returns true if aresult is a result, i.e. a c-pointer to a PGresult structure, and false otherwise.

(result-ref aresult aindex**)** PROCEDURE
result-ref: result integer → association-list
The result-ref procedure accepts a result and a non-negative integer, and returns an association list containing the row in position $aindex + 1$ (i.e. 0 corresponds to the first row) in the result. The keys of the association list are the column names given as Scheme symbols and the values are the column values. If a column value is NULL then the symbol 'null is returne. If aindex is negative or greater than one less the number of row retrieved, an exception is raised.

(result-length aresult**)** PROCEDURE
result-length: result → integer
The result-length procedure accepts a result and returns a non-negative integer containing the number of rows returned by the SQL SELECT or FETCH statement.

(result-metadata aresult**)** PROCEDURE
result-metadata: result → association-list
The result-metadata procedure accepts a result and returns an association list containing metadata for the rows returned by the SQL SELECT or FETCH statement. The keys of the association list are the column names given as Scheme symbols and the values are themselves association-lists with keys 'table, 'column and 'type, corresponding to the table name, column name and column type respectively.

(result-close aresult**)** PROCEDURE
result-close: result → void
The result-close procedure essentially wraps the libpq PQclear function which frees any memory utilized by the the associated PQresult structure pointed to by aresult.

6 Cursors

PostgreSQL provides the ability to associate SQL SELECT statements with named server-side constructs known as CURSORs. Referring to CURSORs by name, programmers can iterate over the rows returned by the associated SELECT statement using commands such as FIRST, NEXT, PREVIOUS, and LAST, as well as commands that allow the retrieval of rows by row number. PostgreSQL CURSORs are complemented by the client-side result data structures that were discussed in §5. Scheme-PG allows programmers to specify whether the rows retrieved by a query should be be maintained as a result, or as a CURSOR, and provides a single set of Scheme procedures to manage the rows regardless of the which option is used. This set of procedures operate on a Scheme struct called a cursor that contains the information needed store the rows as either results or CURSORs. It is important to note that while the set of procedures available to manage rows is independent of the underlying storage mechanism, the efficiency of many of these procedure varies significantly depending on whether results or CURSORs. This fact will be will be further discussed below.

(make-cursor
aconnection aname aclient/server aposition aresult aselect**)** PROCEDURE
make-cursor: connection symbol symbol integer result string → struct
The cursor struct contains all information necessary to manage rows retrieved by a query, maintaining the rows either as a result or as a CURSOR. The

cursor-connection field contains a connection, i.e. a c-pointer to a PGconn structure that maintains a connection to a PostgreSQL database. The cursor-select field is a string that contains the SQL SELECT statement to be executed. The client/server field is a symbol that can have either value 'client or 'server. A value of 'client for the client/server field indicates that Scheme-PG should maintain the rows as a client-side result, while a value of 'server indicates that Scheme-PG should maintain the rows as a server-side CURSOR. When client/sever is 'client, the cursor-result field contains a result, i.e. a c-pointer that points to a PGresult structure, that maintains the retrieved rows. When client/sever is 'server, the cursor-name field contains a symbol that is used to reference a CURSOR that maintains the retrieved rows. The cursor-position field is a non-negative integer that maintains the current row number in the retrieved rows. The reason that cursors maintain cursor-position is to facilitate the implementation of the cursor-next and cursor-previous procedures when cursor-client/server has value 'client. As mentioned, the result data structure is like a Scheme vector, it provides random access to its elements, but does not maintain any state information about the "current position" in the set of rows. Rows in PostgreSQL results are numbered from 0 to one less the number of rows retrieved by the query, however, rows in PostgreSQL cursors are numbered from 1 to the number of rows retrieved by the query. The cursor-position in the cursor struct utilizes the numbering scheme employed by PostgreSQL results, so $0 \leq$ cursor-position $\leq$ number of results retrieved $- 1$. It should be noted that the make-cursor procedure should not be utilized by programmers to create a new cursor, this is the purpose of the cursor-open procedure.

(cursor-open aconnection aselect aclient/server**)** PROCEDURE
cursor-open: connection string symbol → struct
The cursor-open procedure is used to create a new cursor. It accepts a connection, a string containing an SQL SELECT command to run and a symbol having value either 'client or 'server, executes the SQL SELECT command in the manner indicated by aclient/server, and returns a new cursor. It should be noted that open-cursor is the only function in the cursor-xxx family in which we check if cursor-client/server has a valid value, i.e. whether it is 'client or 'server. In all other cursor-xxx functions, we check if cursor-client/server is 'client and if not, we assume that it's 'server.

(cursor-metadata acursor**)** PROCEDURE
cursor-metadata: acursor → association list
The cursor-metadata procedure accepts a cursor and returns an association list containing metadata in the same format as the result-metadata procedure returns.

(cursor-ref! acursor aindex**)** PROCEDURE
cursor-ref!: cursor integer → association list
The cursor-ref! procedure accepts a cursor and a non-negative integer, aindex and returns the aindexth row as an association list and sets cursor-position to

aindex. The keys of the association list are the column names of the columns returned by the query given as symbols and the values are the column values. When aindex is not between 0 and one less than the number of rows retrieved, including the specific case when the query is empty, an exception is raised.

(cursor-ref acursor aindex**)** PROCEDURE
cursor-ref: cursor integer → association list
The cursor-ref procedure accepts a cursor and a non-negative integer, aindex and returns the aindexth row as an association list without modifying cursor-position. The format of the association list is described in the documentation for the cursor-ref! procedure. When aindex is not between 0 and one less than the number of rows retrieved, including the specific case when the query is empty, an exception is raised.

When cursor-client/server is 'client, both the client-ref! and client-ref procedures are quite efficient as they capitalize on the random access methods of PostgreSQL results and simply retrieve the aindex row. When cursor-client/server is 'server, these procedures are in the worst case less efficient. For example, client-ref! calls executes a SQL FETCH ABSOLUTE command. The worst case performance of this call involves traversing, albeit on the server, $O(|\text{rows retrieved}|)$ rows. The performance of client-ref is even worse, since in order to avoid resetting cursor-position, it makes two calls to client-ref!, one to retrieve the aindex row and another to reset cursor-position (and the CURSOR itself) to the position prior to the first call.

(cursor-next acursor**)** PROCEDURE
cursor-next: cursor → association list
The cursor-next procedure sets the cursor-position to cursor-position + 1 and returns the cursor-position + 1^{th} row. The format of the association list is described in the documentation for the cursor-ref! procedure. On attempts to retrieve beyond the last row or from an empty query, an 'eoc-object is returned.

(cursor-previous acursor**)** PROCEDURE
cursor-previous: cursor → association list
The cursor-previous procedure sets the cursor-position to cursor-position − 1 and returns the cursor-position − 1^{th} row. The format of the association list is described in the documentation for the cursor-ref! procedure. On attempts to retrieve before the first row or from an empty query, an 'eoc-object is returned.

(cursor-first acursor**)** PROCEDURE
cursor-first: cursor → association list
The cursor-first procedure sets the cursor-position to 0 and returns the first row as an association list. The format of the association list is described in the documentation for the cursor-ref! procedure. On attempts to retrieve from an empty query, an 'eoc-object is returned.

(cursor-last acursor**)** PROCEDURE
cursor-last: cursor → association list
The cursor-last procedure sets the cursor-position to one less than the number of rows retrieved and returns the last row as an association list. The format of the association list is described in the documentation for the cursor-ref! procedure. On attempts to retrieve from an empty query, an 'eoc-object is returned.

(cursor-length acursor**)** PROCEDURE
cursor-length: cursor → integer
The cursor-length procedure accepts a cursor and returns a non-negative integer which is the the number of rows retrieved by the underlying query. It does this without changing the cursor-position of acursor. When client/server is 'server this operation loops through the entire set of results.

(eoc-object? arow**)** PROCEDURE
eoc-object?: association-list → boolean
The eoc-object? procedure returns #t if arow is 'eoc-object and #f otherwise.

(cursor-close acursor**)** PROCEDURE
cursor-close: cursor → void
The cursor-close "closes" the given cursor by either calling calling result-close in the case that cursor-client/server is 'client, or calling using the appropriate SQL commands to close the associated CURSOR.

It is important to note that procedures cursor-first, cursor-next, cursor-previous, cursor-last and cursor-ref! have the side-effect of setting cursor-position to the position of the row that they return. While this behavior seems to be what a programmer would expect from the cursor-first, cursor-previous, cursor-next and cursor-last procedures, it seems to be unusual behavior for a xxx-ref function. For example, consider list-ref or vector-ref. Hence the procedure name cursor-ref! rather than cursor-ref. The decision to include procedure cursor-ref! with this side-effect was made because the PostgreSQL FETCH function utilized to implement it when cursor-aclient/server is 'server, essentially also has this side-effect and as will be seen, the work around includes a performance penalty. The procedure call (cursor-ref the-cursor 10) is implemented using the SQL command FETCH ABSOLUTE 10 FROM mycursor, after execution of this command, the cursor position, on the PostgreSQL server, is set to the 10^{th} row. To see this note that if a subsequent call to (cursor-next the-cursor), which is implemented using the SQL command FETCH NEXT FROM mycursor, will result in the 11^{th} row being return. To avoid this side-effect in, one strategy would be to make two FETCH calls. The first call would retrieve the desired data, and the second call would reposition the cursor to its position prior to the first call and simply discard any data retrieved. The cursor-ref procedure utilizes this strategy, thereby leaving cursor-position unchanged.

7 Streams

Streams are Scheme data structures that represent sequences of data of infinite or unknown size and that support a similar set of procedures to those that are defined on Scheme lists. They have the advantage of delaying generation of list components until they are needed. Therefore, in Scheme-PG, when using cursors with cursor-client/server set to 'server, this behavior means that we can use list-like procedures on query results without moving all retrieved rows from the server to the client.

The stream implementation included with Scheme-PG is similar to that presented in many classic Scheme references and therefore only special features will be discussed here.

Most streams are implemented as promises that when forced return another promise, with the car of which is a value of interest and the cdr of which is another promise. Scheme-PG streams are similar except that the aforementioned car is a cons cell whose car is the value of interest, i.e. the next row in the set of retrieved rows, and whose cdr is the cursor used to manage the set of retrieved rows. Scheme-PG streams are of the form

$$((\text{row1 cursor}) \ . \ \text{promise1}) \ ((\text{row2 cursor}) \ . \ \text{promise2}) \ \cdots$$

The state, e.g. the cursor-position, of the cursor varying in each stream component. For example, the first component of the stream, the car will be a cons cell containing the first row retrieved and the underlying cursor, with cursor-position set to 0, and the cdr will be promise that when forced will have a car containing the second row and underlying cursor, with cursor-position set to 1, etc. As will be seen below, calling stream-cursor, which simply calls cdar on a stream returns a cursor on which cursor-ref, cursor-ref!, etc. can be called.

Making the cursor available as shown above, was an important design choice, since this allows procedures such as stream-ref, stream-ref!, stream-length and stream-metadata to be conveinently implemented by simply calling the obvious cursor procedures. In the case of the first three procedures this design also results in a much more efficient implementations. For example, a classic stream-ref implementation would loop through all rows until it reached the requested one, but since the cursor is available this can be avoided – when cursor-client/server is 'client the operation is of constant time.

So, Scheme-PG streams have the best of both worlds: delayed list operations as well as fast stream-ref! and stream-ref procedures that don't involve looping when cursor-client/server is 'client and even when cursor-client/server is 'server, the FETCH ABSOLUTE SQL command does the looping on the PostgreSQL server rather than bringing all the data over to the client.

8 Utility Procedures

(escape-string astring**)** PROCEDURE
escape-string: string → string

The escape-string procedure accepts a string and returns the string properly prepared for use by PostgreSQL.

9 SQL Statement Creation

In order to create SQL statements, programmers frequently utilize error prone and often syntactically unappealing string formatting operations. Taking a cue from Scheme-QL, Scheme-PG provides a set of procedures and macros that allow programmers to create SQL statements using a little language implemented in Scheme. While this little language is still under development, it implements enough of SQL that it should prove useful for many applications.

Scheme-PG introduces a set of new forms that mirror a subset of the SQL language functions such as SELECT, INSERT, UPDATE and DELETE. These new forms return valid SQL statements as Scheme strings, are not dependent on other Scheme-PG functionality, and therefore can be used by themselves to support SQL statement creation for use with other Scheme database packages.[1]

In the new forms introduced by Scheme-PG

- SQL keywords such as SELECT, WHERE, UPDATE and DELETE are macro literals
- SQL objects such as columns, tables, views, etc. are represented as Scheme symbols
- SQL values such as the value to be inserted into a database or the value used in a condition in a SQL WHERE clause can be Scheme strings or Scheme numbers in Scheme-PG

Scheme-PG requires that Scheme symbols appear in positions where SQL objects should appear (e.g. a column list in a SELECT statement) and it requires that Scheme strings or numbers appear in positions where SQL values should appear (e.g. a value in an INSERT statement). Scheme-PG handles proper formatting of values. Specifically, the escape-string procedure, (see §8) is called on strings and the result is surrounded by single quotes. Scheme-PG also formats SQL objects by surrounding them with double quotes so that table or column names can for example contain spaces or dashes. Positions in the forms where Scheme-PG expects an SQL object or SQL value are are quasiquoted, so unquoting (e.g. ,first-name ,(get-first-name)) can be utilized.

(where condition**)** SYNTAX
where: condition → string
where: and (listof conditions) → string
where: or (listof conditions) → string
where: not (listof conditions) → string

[1] To use the little language feature by itself simply (require (lib "sql.ss" "scheme-pg")).

The where macro supports the creation of WHERE clauses for use in SQL statements such as SELECT, UPDATE and DELETE. Fundamental to the where clause is a condition which is a length three proper list (operator column value). In a condition, a valid operator is a symbol, a valid column is either a symbol representing a column name or a length two proper list of symbols representing a table name and a column name, and a valid value is either a valid column or a string or number. The case of value being a valid column occurs in a condition in which the values of two database columns are compared. For example,

- SELECT * FROM pers WHERE first-name = last-name
- SELECT * FROM pers,addr WHERE pers.id = addr.id.

The case of value being a string or number occurs in a condition in which the value of a database column is compared to a constant. For example,

- SELECT * FROM pers WHERE last-name LIKE 'Do%'
- SELECT * FROM pers WHERE age < 22.

If value is a valid column it is formatted as described above. If value is not a valid column then, if it is a string it is formatted as described above, if it is a number it is returned unformatted and in all other cases an exception is raised. An example of an additional Scheme data type that could be supported as a value in future releases is a list, which could be used in an SQL IN, e.g. SELECT * FROM addr WHERE state IN ('NH', 'NJ', 'NY').

Scheme-PG tranforms a condition (operator column value) into the string "column operator value" formatting its components as described above. The following examples correspond to the four SELECT statements listed above

- (= first-name last-name) → first-name = last-name
- (= (pers id) (addr id)) → pers.id = addr.id
- (like last-name "Do%") → last-name LIKE 'Do%'
- (< age 22) → age < 22

The where macro accepts either a single condition or the literals and, or and not and a list of conditions. Programmers can build where clauses that contain arbitrarily complex Boolean expression through use of and, or and not as shown by the following examples

- (where (and (like last-name "Do%") (< age 22))) →
  WHERE last-name LIKE 'Do%' AND age < 22
- (where (or (and (> age 18) (< age 22)) (like last-name "Do%"))) →
  WHERE (age > 18 AND age < 22) OR last-name LIKE 'Do%'

(select acolumns atables awhere**)** SYNTAX
select: (listof symbol) (listof symbol) where → string
select: all (listof symbol) where → string
The select macro supports the creation of SQL SELECT statements. It accepts a list of column names as symbols, a list of table names as symbols and an optional where clause. In lieu of the list of column names the single literal all can be provided to generate the * in the common SELECT * ··· statement.

(insert atable avalues**)** SYNTAX
insert: symbol (listof strings and/or numbers) → string

(insert atable acolumns avalues**)** SYNTAX
insert: symbol (listof symbols) (listof strings and/or numbers) → string

(insert atable acolumns-values**)** SYNTAX
insert: symbol (listof (symbols . string and/or number)) → string

(insert atable acolumns-values**)** SYNTAX
insert: symbol (listof (symbols string and/or number)) → string

The insert macro has four rules that support the creation of INSERT statements as shown in the examples below. The following form implements the values-only format of an INSERT statement.

- (insert pers (1 "John" "Doe" 20)) →
  INSERT INTO "pers" (1,'John','Doe',20)

The following forms all generate the same INSERT statement.

- (insert pers (id first-name last-name age) (1 "John" "Doe" 20))
- (insert pers ((id . 1) (first-name . "John") (last-name . "Doe") (age . 20)))
- (insert pers ((id 1) (first-name "John") (last-name "Doe") (age 20))) →
  INSERT INTO "pers" ("id","first-name","last-name","age")
  VALUES (1,'John', 'Doe',20)

(delete atable**)** SYNTAX
delete: symbol → string

(delete atable awhere**)** SYNTAX
delete: symbol where → string

The delete macro has two rules that support the creation of DELETE statements as shown in the examples below.

- (delete pers) → DELETE FROM "pers"

- (delete pers (where (and (< age 45) (= state "NJ")))) →
  DELETE FROM "pers" WHERE "age" < 45 AND "state" = 'NJ'

(update atable acolumns-values**)** SYNTAX
update: symbol (listof (symbols . string and/or number)) → string

(update atable acolumns-values**)** SYNTAX
update: symbol (listof (symbols string and/or number)) → string

(update atable acolumns avalues**)** SYNTAX
update: symbol (listof symbols) (listof strings and/or numbers) → string

(update atable acolumns-values where**)** SYNTAX
update: symbol (listof (symbols . string and/or number)) where → string

(update atable acolumns-values where**)** SYNTAX
update: symbol (listof (symbols string and/or number)) where → string

(update atable acolumns avalues where**)** SYNTAX
update: symbol (listof symbols) (listof strings and/or numbers) where → string

The update macro has six rules that support the creation of UPDATE statements as shown in the examples below. The following three forms all construct the same UPDATE statement:

- (update pers ((first-name "John") (age 20)))
- (update pers ((first-name . "John") (age . 20)))
- (update pers ((first-name age) ("John" 20))) →
  UPDATE pers SET first-name='John', age=20

The following three forms all construct the same UPDATE statement:

- (update pers ((first-name "John") (age 20)) (where (= last-name "Doe")))
- (update pers ((first-name . "John") (age . 20)) (where (= last-name "Doe")))
- (update pers ((first-name age) ("John" 20)) (where (= last-name "Doe"))) →
  UPDATE pers SET first-name='John', age=20 WHERE age < 20

(limit aselect offset number**)** SYNTAX
limit: string (non-negative-integer non-negative-integer) → string
The limit macro appends a LIMIT clause to the end of a SELECT. The offset argument is the number of rows to be skipped, so an offset of 0 means that the first row returned will be the first row of the result. The number argument is an upper bound on the number of rows to return. Less than number rows can

be returned depending on the number of rows in the result and the value of offset.

(order-by aselect column asc/desc**)** SYNTAX
order-by: string (listof (symbol 'asc or 'desc))) → string
The order-by macro appends an ORDER BY clause to the end of a SELECT. It accepts a SELECT statment and a proper list of length two lists that consists of a symbol representing a column name and either the symbol 'asc (to indicate the result should be put in ascending order) or the symbol 'desc (to indicate the result should be put in descending order).

(distinct aselect**)** SYNTAX
distinct: string → string
The distinct procedure checks to the aselect is a string that begins with "SELECT". If so, it replaces "SELECT" with "SELECT DISTINCT", otherwise it raises an exception.

10 Data Type Conversions

This section provides details on the PostgreSQL to Scheme data type conversion performed by Scheme-PG. The reference material used to develop these conversions included PostgreSQL 7.4.3 Documentation, Chapter 8. Data Types, Inside PLT Scheme, The C Programming Language and the include file /usr/local/include/postgresql/server/catalog/pg_type.h. Programmers should note that s-pg-types.h contains data type constants and was manually constructed from the previously mentioned include file. It should be noted that Scheme symbols will be treated as strings. The PostgreSQL Modifier field indicates the value returned by the PQfmod() function.

PostgreSQL Name	bigint
PostgreSQL Alias	int8
PostgreSQL OID	INT8OID = 20
PostgreSQL Modifier	-1
Description	signed eight-byte integer
C Type	.
C function for string to C type conversion	.
Scheme symbol	bigint
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	bigserial
PostgreSQL Alias	serial8
PostgreSQL OID	INT8OID = 20
PostgreSQL Modifier	-1, How do you know that this is unsigned?
Description	autoincrementing eight-byte integer
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	bit
PostgreSQL Alias	N/A
PostgreSQL OID	BITOID = 1560
PostgreSQL Modifier	1
Description	fixed-length bit string
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	bit varying(n)
PostgreSQL Alias	varbit(n)
PostgreSQL OID	VARBITOID = 1562
PostgreSQL Modifier	10
Description	variable-length bit string
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	boolean
PostgreSQL Alias	bool
PostgreSQL OID	BOOLOID = 16
PostgreSQL Modifier	-1
Description	logical Boolean (true/false)
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	box
PostgreSQL Alias	.
PostgreSQL OID	BOXOID = 603
PostgreSQL Modifier	-1
Description	rectangular box in the plane
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	bytea
PostgreSQL Alias	.
PostgreSQL OID	BYTEAOID = 17
PostgreSQL Modifier	-1
Description	binary data
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

*PostgreSQL Name	character varying(n)
PostgreSQL Alias	varchar(n)
PostgreSQL OID	VARCHAROID = 1043
PostgreSQL Modifier	14
Description	variable-length character string
C Type	char *
C function for string to C type conversion	None
Scheme symbol	character-varying
Scheme function to determine Scheme type from Scheme_Object	SCHEME_STRINGP
Scheme function to extract C type from Scheme_Object	SCHEME_STR_VAL, SCHEME_STRLEN_VAL
Scheme function to create Scheme_Object from C type	scheme_make_string

PostgreSQL Name	character(n)
PostgreSQL Alias	char(n)
PostgreSQL OID	BPCHAROID = 1042
PostgreSQL Modifier	14
Description	fixed-length, blank padded character string
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	cidr
PostgreSQL Alias	.
PostgreSQL OID	CIDROID = 650
PostgreSQL Modifier	-1
Description	IPv4 or IPv6 network address
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	circle
PostgreSQL Alias	.
PostgreSQL OID	CIRCLEOID = 718
PostgreSQL Modifier	-1
Description	circle in the plane
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	date
PostgreSQL Alias	.
PostgreSQL OID	DATEOID = 1082
PostgreSQL Modifier	-1
Description	calendar date (year, month, day)
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	double precision
PostgreSQL Alias	float8
PostgreSQL OID	FLOAT8OID = 701
PostgreSQL Modifier	-1
Description	double precision floating-point number
C Type	double
C function for string to C type conversion	atof
Scheme symbol	double-precision
Scheme function to determine Scheme type from Scheme_Object	SCHEME_DBLP
Scheme function to extract C type from Scheme_Object	SCHEME_DBL_VAL
Scheme function to create Scheme_Object from C type	scheme_make_double

PostgreSQL Name	inet
PostgreSQL Alias	.
PostgreSQL OID	INETOID = 869
PostgreSQL Modifier	-1
Description	IPv4 or IPv6 host address
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	integer
PostgreSQL Alias	int, int4
PostgreSQL OID	INT4OID = 23
PostgreSQL Modifier	-1
Description	signed four-byte integer
C Type	int
C function for string to C type conversion	atoi
Scheme symbol	integer
Scheme function to determine Scheme type from Scheme_Object	SCHEME_INTP
Scheme function to extract C type from Scheme_Object	SCHEME_INT_VAL
Scheme function to create Scheme_Object from C type	scheme_make_integer

PostgreSQL Name	interval(p)
PostgreSQL Alias	.
PostgreSQL OID	INTERVALOID = 1186
PostgreSQL Modifier	-1
Description	time span
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	line
PostgreSQL Alias	.
PostgreSQL OID	LINEOID = 628
PostgreSQL Modifier	-1
Description	infinite line in the plane (not fully implemented)
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	lseg
PostgreSQL Alias	.
PostgreSQL OID	LSEGOID = 601
PostgreSQL Modifier	-1
Description	line segment in the plane
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	macaddr
PostgreSQL Alias	.
PostgreSQL OID	MACADDROID = 829
PostgreSQL Modifier	.
Description	MAC address
C Type	-1
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	money
PostgreSQL Alias	.
PostgreSQL OID	CASHOID = 790
PostgreSQL Modifier	-1
Description	currency amount
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	numeric [(p, s)]
PostgreSQL Alias	decimal [(p, s)]
PostgreSQL OID	NUMERICOID = 1700
PostgreSQL Modifier	655367
Description	exact numeric with selectable precision
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	path
PostgreSQL Alias	.
PostgreSQL OID	PATHOID = 602
PostgreSQL Modifier	-1
Description	open and closed geometric path in the plane
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	point
PostgreSQL Alias	.
PostgreSQL OID	POINTOID = 600
PostgreSQL Modifier	-1
Description	geometric point in the plane
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	polygon
PostgreSQL Alias	.
PostgreSQL OID	POLYGONOID = 604
PostgreSQL Modifier	-1
Description	closed geometric path in the plane
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	real
PostgreSQL Alias	float4
PostgreSQL OID	FLOAT4OID = 700
PostgreSQL Modifier	-1
Description	single precision floating-point number
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	smallint
PostgreSQL Alias	int2
PostgreSQL OID	INT2OID = 21
PostgreSQL Modifier	-1
Description	signed two-byte integer
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	serial
PostgreSQL Alias	serial4
PostgreSQL OID	INT4OID = 23
PostgreSQL Modifier	-1
Description	autoincrementing four-byte integer
C Type	.
C function for string to C type conversion	.
Scheme symbol	integer
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	text
PostgreSQL Alias	.
PostgreSQL OID	TEXTOID = 25
PostgreSQL Modifier	-1
Description	variable-length character string
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	time [(p)] [without time zone]
PostgreSQL Alias	.
PostgreSQL OID	TIMEOID = 1083
PostgreSQL Modifier	-1
Description	time of day
C Type	.
C function for string to C type conversion	.
Scheme symbol	time-without-time-zone
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	time [(p)] with time zone
PostgreSQL Alias	timetz
PostgreSQL OID	TIMETZOID = 1266
PostgreSQL Modifier	-1
Description	time of day, including time zone
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	timestamp [(p)] without time zone
PostgreSQL Alias	timestamp
PostgreSQL OID	TIMESTAMPOID = 1114
PostgreSQL Modifier	-1
Description	date and time
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

PostgreSQL Name	timestamp [(p)] [with time zone]
PostgreSQL Alias	timestamptz
PostgreSQL OID	TIMESTAMPTZOID = 1184
PostgreSQL Modifier	-1
Description	date and time, including time zone
C Type	.
C function for string to C type conversion	.
Scheme symbol	.
Scheme function to determine Scheme type from Scheme_Object	.
Scheme function to extract C type from Scheme_Object	.
Scheme function to create Scheme_Object from C type	.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, second edition, The MIT Press, Cambridge, Massachusetts, 1996.

[2] Philip L. Bewig, *SRFI 40: A Library of Streams*, http://srfi.schemers.org/srfi-40/, 2003.

[3] Ryan Culpepper, *spgsql: A PostgreSQL Database Library*, http://schematics.sourceforge.net/spgsql.html.

[4] R. Kent Dybvig, *The Scheme Programming Language*, third edition, The MIT Press, Cambridge, Massachusetts, 2003.

[5] Daniel P. Friedman and Matthias Felleisen, *The Little Schemer*, fourth edition, The MIT Press, Cambridge, Massachusetts, 1996.

[6] Daniel P. Friedman and Matthias Felleisen, *The Seasoned Schemer*, The MIT Press, Cambridge, Massachusetts, 1996.

[7] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, second edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[8] Noel Welsh, Francisco Solsona and Ian Glover, *SchemeUnit and SchemeQL: Two Little Languages*, Scheme 2002 Workshop, Pittsburgh, Pennsylvania, October 3, 2002.

[9] PLT Scheme, http://www.plt-scheme.org.

[10] PostgreSQL, http://www.postgresql.org.