Specifications of TSODL for C
Version 1.0.0 Draft

Laurent Lyaudet `https://lyaudet.eu/laurent/`

May 17, 2019

Abstract

TSODL (Tree Structured Orders Definition Language) is a family of languages for defining orders. These specifications define the dialect of TSODL compatible with the C programming language. A Tree Structured Order Definition (TSOD) using this dialect may be parsed to generate optimized C code for sorting using this order.

Contents

Chapter 1

Introduction

A result by Cantor (1895) shows that all countable orders may be sorted using any sorting algorithm suited for lexicographic sorting or any linear extension of the partiel order "Next". In less scientific terms, everything can be sorted as if it was string sorting. Tree structured orders and some of their suborders can be efficiently nextified/stringified/lexicographicalized (see Lyaudet (2018)).

The goal of the dialects of TSODL are:

- it could be used for order by clauses of SQL queries or other query languages,
- but it could also be used to generate code in your favorite programming language if you intend to sort objects without using a database,
- or it could be used as an input of a library that provides functions to prepare the array to sort (TSO-encoding) and sort accordingly.

Both generated code or dynamic code in library could provide a comparison function and a "nextification" function, computed from the tree structured order definition, and both code could switch between comparison model and lexicographic model, whichever is faster, according to the number of elements to sort and other parameters deduced from the tree structured order definition.

First each node of a tree structured order is centered around an object (or a structure in C language, etc.). The root node is centered on the main objects, the ones that correspond to the wanted level of granularity.

Let us explain our ideas using an example from our current work in business software for freight forwarders. Each night/morning, a freight forwarder receives trucks with freight from other freight forwarders. For each truck and each day, we have an "Arrival note" in the database. To this arrival note are linked shipments, each shipment has a certain number of handling units. We have these three levels of granularity: arrival note (corse grained), shipments, handling units (fine grained). Each of these three classes may have between a dozen and a few hundreds fields. Assume we want to sort shipments according to the name of the freight forwarder that brought his truck (field found in the arrival note), the weight of the freight in the truck by decreasing order (arrival note), the weight of the shipment by increasing order, the barcodes of the handling units.

A Tree Structured Order Definition for this sort would look like:

```
NEXT(
    CURRENT.arrival_note.forwarder_name VARCHAR(NULL),
    CURRENT.arrival_note.total_weight DOUBLE DESC,
```

```
    CURRENT.weight DOUBLE,
    LEX(
        1,
        0, //0 codes omega (countable infinity)
        CURRENT.number_of_handling_units, //the actual prelude length
        CURRENT.array_of_handling_units, //the array of pointers to redefine current
        ([
          //current has been redefined for the suborders.
          CURRENT.barcode VARCHAR(NULL)
        ])
    )
)
```

Note that we assumed denormalization or caching for the total weight of an arrival note. With TSO-encoding, that value, if not directly available, would be computed once and stocked into the TSO-encoding. With black box model and callbacks, you have to cache this value somehow, otherwise it will be computed on each comparison.

The goal of these specifications is to detail the definition of the Tree Structured Orders Definition Language (TSODL) in its dialect adapted to the C programming language (TSODL for C).

Chapter 2

Preview

Introduction chapter presented an exemple of TSOD. We give here more examples specific to C code.

Assume you have a C struct defined as follow:

```
typedef struct my_car {
  char* s_constructor;
  char* s_model;
  int i_year;
  int i_horse_power;
  float f_mileage;
} t_my_car;
```

Assume you want to sort an array of such structs by ascending horse power. Your TSOD would simply be:

```
CURRENT.i_horse_power INT
```

or it could be:

```
CURRENT.i_horse_power INT ASC
```

CURRENT is a reserved keyword for current item in the array, its meaning may vary when we go down in a hierarchy of structs. *INT* is a reserved keyword for specifying that the field i_horse_power is of int C type. *ASC* is a reserved keyword taken from SQL for specifying ascending order.

If you want to sort an array of pointers to such structs by horse power. Your TSOD would simply be:

```
CURRENT->i_horse_power INT
```

We do not reinvent the wheel, we use the C native distinction between '`.`' and '`->`'.

If you want to sort by descending horse power:

```
CURRENT->i_horse_power INT DESC
```

DESC is a reserved keyword taken from SQL for specifying descending order.

If you want to sort by descending horse power, and then break ties by ascending mileage:

```
NEXT(
  CURRENT->i_horse_power INT DESC,
  CURRENT->f_mileage FLOAT,
)
```

The line returns and the indentation convey no meaning, but it is nicer to read. *NEXT* is a reserved keyword for *Next* partial order (if current fields are equal, compare *Next* fields). Note that by default, TSODL for C is case sensitive. You may use "pragmas", to be defined later, so that the keywords are case insensitive. The comma after FLOAT is no accident, you may optionaly end a list with a comma. In particular, if you commit a TSOD in your application and add another field relevant to the order or comment one field, later on, you do not have to modify commas on other lines, and your blame view of the file will only show real modifications of the source lines.

If you want to sort by constructor, and then break ties by year:

```
NEXT(
  CURRENT->s_constructor VARCHAR(NULL),
  CURRENT->i_year INT,
)
```

VARCHAR is a reserved keyword taken from SQL meaning that the length of the string may vary. By default, varying length strings will be sorted using lexicographic order. *VARCHAR(NULL)* means that the end of the string is known thanks to a null byte (value 0). If you fancy your own strings terminated by a byte of value 3, you can use *VARCHAR(NULL(3))*. *VARCHAR(NULL(0))* and *VARCHAR(NULL)* have the same meaning.

Assume that the C struct is slightly different:

```
typedef struct my_car {
  char* s_constructor;
  int i_s_constructor_size;
  char* s_model;
  int i_s_model_size;
  int i_year;
  int i_horse_power;
  float f_mileage;
} t_my_car;
```

Instead of null-terminated strings, you have explicit length counters for the strings. If you want to sort by constructor, and then break ties by year:

```
NEXT(
  CURRENT->s_constructor VARCHAR(CURRENT->i_s_constructor_size INT),
  CURRENT->i_year INT,
)
```

Assume that the C struct has fixed length strings:

```
typedef struct my_car {
  char[31] s_constructor;
  char[31] s_model;
  int i_year;
  int i_horse_power;
  float f_mileage;
} t_my_car;
```

You will use the TSOD:

```
NEXT(
  CURRENT->s_constructor CHAR(31),
  CURRENT->i_year INT,
)
```

Until now, we assumed that the collation used for sorting the strings was a no-op on byte values. You may want to group first the corresponding uppercase and lowercase letter. If both 'A' and 'a' gets the same value, etc., so that 'bCBc', 'BAba', 'baBA', are sorted like 'baBA', 'BAba', 'bCBc', then you need to sort first using your grouping collation, and sort later using a total order on the byte values. Let us assume that you have a C function with the following signature:

```
char my_collation(char* s_string, size_t* p_i_current_offset);
```

my_collation groups lowercase and uppercase letters. You may use the TSOD:

```
NEXT(
  CURRENT->s_constructor CHAR(31) COLLATION(CHAR my_collation),
  CURRENT->s_constructor CHAR(31),
  CURRENT->i_year INT,
)
```

The second sort on s_constructor is necessary to break ties ('baBA', 'BAba' instead of 'BAba', 'baBA' assuming that the first sort was stable). If it is UTF-8 strings, your collation may return an int and read many chars at once.

```
NEXT(
  CURRENT->s_constructor CHAR(31) COLLATION(INT32 my_collation),
  CURRENT->s_constructor CHAR(31) COLLATION(INT32 my_collation_break_ties),
  CURRENT->i_year INT,
)
```

You may optimze the produced code by giving range indications in your TSOD:

```
NEXT(
  CURRENT->s_constructor CHAR(31) COLLATION(INT32 RANGE(0, 2500000) my_collation),
  CURRENT->s_constructor CHAR(31)
    COLLATION(INT32 RANGE(0, 2500000) my_collation_break_ties),
  CURRENT->i_year INT RANGE(1900, 2100),
)
```

With *RANGE(1900, 2100)*, TSODLULS knows that only a single byte is required to sort by year instead of 2 or 4 bytes. With *RANGE(0, 2500000)*, TSODLULS knows that only three bytes are required for each unicode character. Shortening the key used for sorting is crucial for good performances.

Chapter 3

Production rules

Chapter 4

String collations

Collations used for sorting string may be arbitrary functions if you want to generate code. TSODLULS will simply assume that you will provide the appropriate collation function when you will compile the code. TSODLULS defines standards collations that are already given, ready to use.

4.1 Standard collations for ASCII strings

4.2 Standard collations for Unicode strings

We thank God: Father, Son, and Holy Spirit. We thank Maria. They help us through our difficulties in life.

Bibliography

G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Math. Ann.*, 46:481–512, 1895.

L. Lyaudet. A class of orders with linear? time sorting algorithm. *CoRR*, abs/1809.00954, 2018. URL `http://arxiv.org/abs/1809.00954`.