Sorting Index Entries with xindy revisited

Roger Kehr
kehr@iti.informatik.tu-darmstadt.de
Darmstadt University of Technology

October 20, 1997

This chapter discusses how sorting of index entries for arbitrary languages can be achieved with xindy. It presents the recently implemented sorting scheme and describes two example specifications for German and French.

1 Motivation

Sorting index entries is not a trivial task as one might think at first. This is especially the case in many non-English languages. Sorting English words is rather trivial compared to these languages, which is probably one reason why some systems simply ignore the pecularties of these languages.

To emphasize these introductory sentences we outline a simple example. In German some words such as *arm* (poor) and *Arm* (arm) are distinguished by their case. Sorting German is done roughly as follows (in fact there exists more than one ordering scheme):

1. Sort all words independent of their case. Hence, *arm* and *Arm* are first considered to be equal.

2. Then, in a new sorting phase, all remaining and unresolved orders are sorted according to a new criteria. This could be: sort all lowercase letters before the corresponding uppercase ones. In this case the word *arm* appears before *Arm* and a total order is achieved.

The most important result of this example is that a total order can only be obtained if we consider the concept of *sorting phases*. Hence, sorting index entries is a task that has to be performed in several, largely independent phases.

2 The Sorting Model

The sorting model implemented in xindy can be seen as pipeline of filters. It starts from the top with the so-called *merge key*, which is a string. The merge key basically defines the *identity* of an index entry. Two index entries with the same merge key are considered to be *equal* and are unified (merged) prior to sorting.

Starting from the merge key several sorting phases determine the order of the index entries. A sorting phase consists of two steps:

1. The first step is the so-called *sort mapping*. The sort mapping consists of a string rewriting mechanism. With the so-called *sort rules* the user can specify a set of character, string, and regular expression substitutions that are applied to the merge key. The result of this rewriting mechanism is the *sort key*.

2. The second step takes all sort keys and sorts them according to the underlying encoding scheme (i.e. ISO Latin-1) in some order, usually from left to right. Since more than one merge key may be mapped onto the same sort key (i.e. to ignore cases), the result of the sorting process may still contain ambiguities w.r.t. a total order of all index entries.

To achieve a total order on the index entries or to implement all the rules of a particular language, the previous steps have to be done repeatedly with different sort mappings to implement the full sorting scheme. In the example of German, the first phase must map all uppercase letters to their lowercase counterparts ('A'→'a', 'B'→'b', etc.). This makes both indistinguishable in the first phase, though the relative order to other letters is preserved. In the second phase (if there are remaining ambiguities at all) the mapping must state that lowercase letters precede uppercase ones. Since in the usual ASCII alphabet uppercase letters precede lowercase ones, we can, for example, replace all lowercase letters by uppercase ones and vice versa ('A'→'a', 'a'→'A', etc.). Since sorting is done using the underlying encoding, we actually obtain the desired sorting order. Other than the proposed mapping schemes are perfectly possible, as long as the obtained sort key reflects the correct sorting order.

There exists one command for defining sort rules. It full syntax is as follows:

```
(sort-rule pattern replacement
           [:run n]
           [:string | :bregexp | :eregexp])
```

With the parameter `:run` *n* one declares a rule to be in effect only in a certain sorting phase. If it is omitted, phase 0 (the first phase) is assumed.

As already stated, the usual sort key comparisons work from left to right of a string, therefore operate on a *lexicographic* sorting order. There are some sorting rules, for example the French ones, that in some phases operate from right to left. Hence, an additional *orientation* is needed for each sorting phase, to obtain the desired behaviour. The command

```
(define-sort-rule-orientations (orientations...))
```

defines the order for all sorting phases. The currently implemented *orientations* are `forward` and `backward`. This command must precede all `sort-rule` commands in an index style. It defines the orientations and implicitly sets the maximum number of sorting phases performed.

That's the whole magic in defining sort rules with `xindy`. The mechanism is simple, flexible, and powerful. The only disadvantage is that a complete specification may be a bit too lengthy, as one can see from the examples in the following section.

3 Example Specifications

3.1 Sorting German

There exist two main sorting schemes in German. One sorts the umlaut letters (those with a trema) in a first phase as if they were written without a trema but having the letter *e* appended. This is because the letters with umlauts were written in this form in ancient times. The other rule (sorting uppercase and lowercase letters) has already been described earlier. Another rule is to expand *ß* into *ss* in a first phase.

The sort rules for the first phase are then as follows:

```
(sort-rule "A" "a"  :run 0)
 ...
(sort-rule "Z" "z"  :run 0)

(sort-rule "Ä" "ae" :run 0)
(sort-rule "Ö" "oe" :run 0)
(sort-rule "Ü" "ue" :run 0)
(sort-rule "ä" "ae" :run 0)
(sort-rule "ö" "oe" :run 0)
(sort-rule "ü" "ue" :run 0)

(sort-rule "ß" "ss" :run 0)
```

In the second phase we have to separate the index entries that are only distinguished by the umlaut stuff. Hence we still have to distinguish between lowercase and uppercase letters.

```
(sort-rule "A"  "a"  :run 1)
 ...
(sort-rule "Z"  "z"  :run 1)
```

The umlaut letters always follow the non-umlauts as follows:

```
(sort-rule "ae" "ae " :run 1)
(sort-rule "ä"  "ae+" :run 1)
(sort-rule "Ä"  "ae+" :run 1)
```

```
(sort-rule "oe" "oe " :run 1)
(sort-rule "ö"  "oe+" :run 1)
(sort-rule "Ö"  "oe+" :run 1)

(sort-rule "oe" "oe " :run 1)
(sort-rule "ü"  "ue+" :run 1)
(sort-rule "Ü"  "ue+" :run 1)
```

Here, we have simply used *tokens*, composed of three letters to indicate this sorting scheme. The rule-of-thumb is to expand the pseudo-token with a blank (preceding all other letters in the ISO Latin-1 alphabet) and to add to the other letters a character that comes behind it somewhere, such as a plus sign. Keep in mind that these rules are applied to only a few words of an index. Only index entries that are mapped onto the same sort key are of interest. Hence, the number of index entries involved in further phases is usually small.

Additionally we can resolve the sharp-s dilemma. Note the blank in this solution that artificially expands all *s* letters. This solves problems such as sorting the two different words *Schloss* and *Schloß*.[1]

```
(sort-rule "s" "s " :run 1)
(sort-rule "ß" "ss" :run 1)
```

In the third an last run we sort the lowercase before the uppercase ones:

```
(sort-rule "A" "a"   :run 2)
 ...
(sort-rule "Z" "z"   :run 2)

(sort-rule "a" "A"   :run 2)
 ...
(sort-rule "z" "Z"   :run 2)
```

And finally resolve the umlaut ambiguity as follows:

```
(sort-rule "ä" "ae " :run 2)
(sort-rule "Ä" "ae+" :run 2)

(sort-rule "ö" "oe " :run 2)
(sort-rule "Ö" "oe+" :run 2)

(sort-rule "ü" "ue " :run 2)
(sort-rule "Ü" "ue+" :run 2)
```

[1] I hope that it still makes a difference in our new German orthographic rule set, who knows, how it's written today?

3.2 Sorting French

We do not specify all the rules necessary to sort French. Instead we present a solution to a very peculiar problem when sorting words in this language. The sorting rules for French say that in a first phase words have to be sorted without considering accents. Hence, all letters with diacritical marks are sorted as if they had none. We show this for the *e* and *o* letters.

```
(sort-rule "è" "e" :run 0)
(sort-rule "é" "e" :run 0)
(sort-rule "ê" "e" :run 0)
(sort-rule "ë" "e" :run 0)

(sort-rule "ô" "o" :run 0)
(sort-rule "ò" "o" :run 0)
(sort-rule "ó" "o" :run 0)
```

The previous rules map the keywords *cote*, *côte*, *côté*, and *coté* onto the sort key *cote*. Therfore further rules have to be defined.

In the next run we define an order on the diacritical marks. The French rules say that in this case the word is compared letter to letter from right to left. Hence, in this run we must sort backwards. Now we define an order on the accents. Since the comparison now is from right to left (backwards) the tokens define their order number *before* the character.

```
(sort-rule "e" " e" :run 1)
(sort-rule "è" "1e" :run 1)
(sort-rule "é" "2e" :run 1)
(sort-rule "ê" "3e" :run 1)
(sort-rule "ë" "4e" :run 1)

(sort-rule "o" " o" :run 1)
(sort-rule "ò" "1o" :run 1)
(sort-rule "ó" "2o" :run 1)
(sort-rule "ô" "3o" :run 1)
```

To sort backwards, the command

```
(define-sort-rule-ordering (forward backward))
```

must be issued *before* the actual sort rules are defined.

This sorting scheme guarantees, that the result is sorted according to the French sorting rules. Actually, I'm not quite sure about the exact order of the diacritical marks, but the example words from above should now occur in an index in the same order as they were introduced.

4 Summary

I hope one gets the point of this system. It has some similiarities with the ISO Standard 14651 *International sorting order*. Other ideas were borrowed from discussions on the xindy mailing list. It's not very elegant but powerful, since it allows very complex rewriting systems. Especially if multilingual indexes have to be processed, that actually need a completely new sorting scheme generalizing more than one language, it may show its flexibilty.

Send me examples of your solutions—I want learn more about it.