xindy

A Flexible Indexing System

Roger Kehr, Institut für Theoretische Informatik, TH-Darmstadt
Draft Discussion Paper on International Sorting

This paper dicusses some of the problems that must be solved to sort international alphabets with **xindy**. *Any fruitful comments, further hints and suggestions are welcome. Please contact the* **xindy** *mailing list for your ideas.*

Contents

1 Problem

One of the most interesting new features of xinfy is its ability to offer mechanisms to sort arbitrary keyword using the *sort mapping*. This scheme was first proposed in [1] and implemented in the *International MakeIndex* [2]. This scheme has been integrated into the current version of **xindy**.

While playing around with the `sort-rule` mechanisms of **xindy** I was confronted with the problem that there often exist many ways to accomplish as particular sorting order but I was not aware of the underlying *rules* how to write a correct sorting specification.

Therefore, I wrote down a simple guideline which is presented.

1.1 Initial guideline

Note: This guideline was a first attempt to understande the meta-rules of writing sort rules and I'll give it here to make you able to get into the discussion!

How to write correct sort and merge rules?

One of the biggest hurdle in mastering **xindy** is the correct declaration of sort and merge rules. To bring a little bit of structure into this problem area we have prepared a set of rules-of-thumb that may help you to correclt specify rules.

First of all, if you have ever written a set of rules and the result is what you expect, then your rules are correct, since correctness is only a question of the result and not the way you did it.

But you may run into difficulties later if the rule set is not correct meaning that your rules may fail under certain circumstances.

1.1.1 Sort Rules

What you always must keep in mind is that the sort rules are applied to the result of the merge mapping. Thus, if you have already made a mistake in the merge mapping the sort rules often do not work as expected but actually may be correct!

1.1.2 Inserting Letters

The most common problem is to define an alphabet that is based on the latin alphabet but is extended by some letters. Inserting a letter often actually means to sort it before and after a letter, thus it is relative to an existing letter. For example the letter á must in some languages be sorted before a. In this case we should specify the following sort rule using the special letter `~b` having a sorting order which is lower than all other characters. Actually it is implemented as character No. one (zero is reserves as the C string end delimiter character).

With the rule

á → a˜b

we would obtain the correct order

barábas → bara˜bbas

barabas → barabas

But what happens with

bará → bara˜b

bara → bara

This is obviously incorrect, since the second string is shorter than the first one. Thus inserting letters without considering the length of the result is not always a good idea.

1.1.3 The principle of tokenising

One answer to correctly spcifying rules lies in tokenising.

What we actually have is a group of letters that must be sorted correctly. The solution is to map all members of this group as follows:

á → a0

a → a1

à → a2

ä → a3

(Never use :again in these rules! This would lead replace a by a1 by a11 by a111...)

Thus we have produced a set of tokens a1,...,a4 that represent the members of the a-group.

Appending letters to an alphabet can be done exactly the same way:

For Finnish:

z → z˜b (or z0)

ä → za (or z1)

ö → zo (or z2)

1.1.4 Upper and lowercase letters

In German 'Arm' and 'arm' are two different words (meaning 'arm' and 'poor' resp.), but usually the lowercase variant preceeds the uppercase one. Both form a token group.

á → a0 (or a00)

Á → a1 (or a01)

a → a2 (or a02)

A → a3 (or a03)

à → a4 (or a10)

À → a5 (or a11)

ä → a6 (or a12)

Ä → a7 (or a13)

In the example of the Finnish alphabet one might write

Z → zZ

z → zz

Ä → zA

ä → za

Ö → z0

ö → zo

Thus, there are several possibilites to resolve this ambiguity but it is essential to do some form of tokenising to obtain predictable results.

1.1.5 The Problem with Hyphens

Leading hyphens should usually disappear:

```
star
   -gate
   field
```

This can be accomplished with the follwing rule:

```
(sort-rule "-" "")
```

but for the both (different) German words ‘Druck-Erzeugnis’ and ‘Drucker-Zeugnis’ one would expect the first one to appear before the second one. Above rule would in conjunction with case insensitive mapping for both words yield the same sort key, therefore the order of appearance would be not deterministic.

1.2 More Problems

After these initial ideas we were pointed to the French sorting rules (thanks Chris), which can be roughly described as follows (for a detailed description refer to [3])

The French sorting rules sort words in a first run as if they contained no accents. There may some words left that are equal if accents are not considered. For example, the words *cote*, *côte*, *coté*, and *côté* are equal in this sense. In the second sorting phase each set of words being equal in this sense is ordered considering the accents but sorting lexicographically from right to left (!). The resulting order is then the same order as listed above, which is not obviuos.

1.3 First Summary

Reviewing these examples it is clear that the intially intended sorting mechanism in **xindy** is hardly capable of handling these rather complex sorting schemes using only string and regular expression substitutions. In fact it is possible, but we would need a lot of regular expression substitutions appending something to the end of strings (this would simulate several runs). Furthermore, this approach would be rather slow and hardly manageable from the user’s perspective.

2 The I18N solution and ISO 14651

The I18N (Internationalization) approach handles international sorting rules with its *Locale* descriptions. The current implementation allows the definition of letters (character classification) via its `LC_CTYPE` feature and the definition of a collation order by the declarations in the `LC_COLLATE` category. A good description of these topic can be found in [4]. The forthcoming ISO 14651 describes the principles of the `LC_COLLATE` approach and discusses various aspects of this solution.

A brief description of their solution can be given as follows:

Given is a set of strings to be ordered according to a set of some rule sets. A string is sorted according to the first rule set. For example, in the first rule set there is no difference in the case of letters and their accents (two words are then called *quasi-homograph*). The result of this first step is a partial order on the keywords. There may be left sets of keywords for which the relative order of the set members is not known, yet. This happens because these (initially different) keywords were mapped into some kind of equivalence class. For example, when ignoring case the words *foo*, *FÓO* and *foô* are all considered to be equal.

For each of these equivalence classes another set of rules is applied trying to further make the partial order become less partial. The second rule set should now further define the relative position into an equivalence class. This can be done by now taking the accents into consideration. After the second rule set is applied to the keywords there may be still equivalence classes left, for example the keyword differing only in the case of their letters (called *homograph*). A result of this step might be the keywords *foo, FOO, Foo*.

In the next run using another set of rules the amiguity can be eliminated and a total order can be obtained.

The ISO standard proposes at least four different runs to be sure to obtain a total order on Latin-based alphabets. This includes backward comparison, which is needed for implementing the French sorting rules.

3 Proposal

3.1 A possible solution

The `locale` approach can be used to describe the sorting schemes of may languages. Though, using it in conjunction with **xindy** has several disadvantages.

1. We could incorporate the GNU libc implementation of `locale` into **xindy**. But defining a new set of rules would have a significant overhead. A new table must first be created and compiled with `localedef` into a form readable by the library. This is in contrast to the dynamic paradigm of **xindy** and therefore in my opinion not feasible.
2. We could extract parts of the library and reuse parts of the implementation. Actually I have not yet looked into the source but I'm not sure if this helps to solve the problem.
3. A largely complete specification in `LC_COLLATE` is rather complex and in no way *declarative* which initially was a design goal of **xindy** Additionally, it solves the problem of pure alphabets but dealing with markup in the keyword is still an open issue. The markup can often not be pre-processed since the markup may only play a rule in one of the later sorting runs. It actually depends on the user's needs.

Based on these observations my current proposal is a mixture of the `locale` approach and the current implementation in **xindy**.

- My proposal is to extend the sorting rule mechanism to incorporate the idea of several successive runs obtaining partial orders until a total order is obtained.
- The specifiaction of the command `sort-rule` can be extended with an additional argument `:level` taking the number of the level into which the sort rule is to be put. Additionally there must be a specification on how each run is to be sorted (forward, backward). These rules may still contain regular expression substitutions which may come into consideration at any level as necessary.

I'll give an example of how powerful this approach can be:
Assuming we have the following keywords to sort:

```
\tt{ARM}
\it{arm}
Arm
arm
Armbrust
armselig
```

Taking into consideration that we want to sort case-independent at the first level of comparision this can be done with the following rule set:

```
A          -> a
\tt{(.*)} -> \1  :again
\it{(.*)} -> \1  :again
```

This obtains the following result:

1. equivalence class (arm): `Arm, arm, \it{arm}, \tt{ARM}`
2. equivalence class (armbrust): `Armbrust`
3. equivalence class (armselig): `armselig`

The intended sorting rule says that the keywords containing markup should come before the others. Thus we must define a rule set expressing this sort order:

```
\tt{(.*)} -> 0\1  :again
\it{(.*)} -> 1\1  :again
A          -> 2a
a          -> 2a
```

Now we have prefixed the letters to obtain a further relative sorting order:

1. equivalence class (02arm): `\tt{ARM}`
2. equivalence class (12arm): `\it{ARM}`
3. equivalence class (2arm): `Arm, arm`

The last step is now to obtain a total order. We do not specify any other rules. since we sort according to the position in the ISO Latin alphabet with A being before a obtaining

1. equivalence class (Arm): `Arm`
2. equivalence class (arm): `arm`

Thus, we have gradually refined the partial order into an total one. The advantage is that we are still able to use declarative descriptions such as `\tt{(.*)} -> \1` to match a many keywords at once.

3.2 Open issues

I have several questions about this scheme and I'm interested in your opinion.

1. Are there any deficiencies in this scheme. What are the cases it does not cover?
2. Is this understandable for Joe Anyuser?
3. Must it actually be completely understandable for Joe Anyuser? (see next question)
4. How could a simple user front-end look like that hides most of the details from the user? (We could use Lisp's macro-mechanism to preprocess some other form of description that expands to a bunch of these primitive mapping rules presented above.)
5. The principle of tokenising is not explicitly defined in this scheme. To make it real good we must operate entirely on tokens (see the `2arm` stuff for example which is actually only a work-around to re-incorporate tokens in a very uncleanly manner). But is it necessary to include tokens?

4 References

The following books and papers were referenced in this document.

1. Joachim Schrod, *An International Version of MakeIndex*, Cahiers GUTenberg, 10, p81-90, 1991

2. Joachim Schrod and Gabor Herr, *MakeIndex Version 3.0*, Technical University of Darmstadt, 1991

3. ISO 14651, *International sorting order*

4. Man-page of `locale(5)`, A good description is available on SUN Solaris Systems. Other information and examples of locale specifications for various alphabets are available in the GNU libc distribution (package `glibc-localedata`).