FST Intersection: Ending Dictionary Redundancy in Apertium

Author1, Author2, Author3

Affiliation1, Affiliation2, Affiliation3
Address1, Address2, Address3
author1@xxx.yy, author2@zzz.edu, author3@hhh.com

Abstract

A Finite State Transducer (FST) used as an analyser, whose output is input to another FST, may have entries that don't pass through the second FST. We discuss certain problems that this creates in the Apertium machine translation platform, and describe the development of a tool to *trim* such entries. The tool is made part of Apertium's `lttoolbox` package.

1. Introduction and background

Apertium (Forcada et al., 2011) is a rule-based machine translation platform, where the data and tools are released under a Free and Open Source license (primarily GNU GPL). Apertium translators use Finite State Transducers (FST's) for morphological analysis, bilingual dictionary lookup and generation of surface forms; most language pairs[1] created with Apertium use the `lttoolbox` FST library for compiling XML dictionaries into binary FST's and for processing text with such FST's.

Below we give some background on how these FST's fit into Apertium as well as their capabilities; then we discuss the problems that have lead to redundant dictionary data, and introduce our solution.

1.1. FST's in the Apertium pipeline

Translation with Apertium works as a pipeline, where each *module* processes some text and feeds its output as input to the next module. First, a surface form like 'fishes' passes through the **analyser** FST module, giving a set of analyses like `fish<n><pl/fish<vblex><pres>`, or, if it is unknown, simply `*fishes`. Tokenisation is done during analysis, letting the FST decide in a left-right longest match fashion which words are tokens. The compiled analyser technically contains several FST's, each marked for whether they have entries which are tokenised in the regular way (like regular words), or entries that may separate other tokens, like punctuation. Anything that has an analysis is a token, and any other sequence consisting of letters of the `alphabet` of the analyser is an unknown word token. Anything else can separate tokens.

After analysis, one or more **disambiguation** modules select which of the analyses is the correct one. The **pretransfer** module does some minor formal changes to do with multiwords.

Then a disambiguated analysis like `fish<n><pl>` passes through the **bilingual** FST. Using English to Norwegian as an example, we would get `fisk<n><m><pl>` if the bilingual FST had a matching entry, or simply `@fish<n><pl>` if it was unknown in that dictionary. So a known entry may get changes to both lemma (`fish` to `fisk`) and tags (`<n><pl>` to `<n><m><pl>`) by the bilingual FST. When processing input to the bilingual FST, it is enough that the *prefix* of the tag sequence matches, so a bilingual dictionary writer can specify that `fish<n>` goes to `fisk<n><m>` and not bother with specifying all inflectional tags like number, definiteness, tense, and so on.

The output of the bilingual FST is then passed to the **structural transfer** module (which may change word order, ensure determiner agreement, etc.), and finally a **generator** FST which turns analyses like `fisk<n><m><pl>` into forms like 'fiskar'. Generation is the reverse of analysis; the dictionary which was compiled into a generator for Norwegian can also be used as an analyser for Norwegian, by switching the compilation direction.

A major feature of the `lttoolbox` FST package is the support for multiwords and compounds. A **lexical unit** may be

- a simple, non-multiword like the noun 'fish',
- a space-separated word like the noun 'hairy frogfish', which will be analysed as one token, but otherwise have no formal differences from other words,
- a multiword with inner inflection like 'takes out'; this is analysed as `take<vblex><pri><p3><sg># out` and then, after disambiguation, but before bilingual dictionary lookup, turned into `take# out<vblex><pri><p3><sg>` that is, the uninflected part (called the $lemq$) is moved onto the lemma,
- a token which is actually two words like 'they'll'; this is analysed as `prpers<prn><subj><p3><mf><pl> +will<vaux><inf>` and then split after disambiguation, but before bilingual dictionary lookup, into `prpers<prn><subj><p3><mf><pl>` and `will<vaux><inf>`,
- a combination of these three multiword types, like Catalan 'creure-ho que', analysed as `creure<vblex><inf> +ho<prn><enc><p3><nt># que` and then moved and split into `creure# que<vblex><inf>` and

[1] A *language pair* is a set of resources to translate between a certain set of languages in Apertium, e.g. Basque–Spanish.

`ho<prn><enc><p3><nt>` after disambiguation, but before bilingual dictionary lookup.

In addition to the above multiwords, where the whole string is explicitly defined as a path in the analyser FST, we have dynamically analysed compounds which are not defined as single paths in the FST, but still get an analysis during lookup. To mark a word as being able to form a compound with words to the right, we give it the 'hidden' tag `<compound-only-L>`, while a word that is able to be a right-side of a compound (or a word on its own) gets the tag `<compound-R>`. These hidden tags are not shown in the analysis output, but used by the FST processor during analysis. If the noun form 'frog' is tagged `<compound-only-L>` and 'fishes' is tagged `<compound-R>`, the `lttoolbox` FST processor will analyse 'frogfishes' as a single compound token `frog<n><sg>+fish<n><pl>` (unless the string was already in the dictionary as an explicit token) by trying all possible ways to split the word. After disambiguation, but before bilingual dictionary lookup, this compound analysis is split into two tokens, so the full word does not need to be specified in either dictionary. This feature is very useful for e.g. Norwegian, which has very productive compounding.

1.2. The Problem: Redundant data

Ideally, when a monolingual dictionary for, say, English is created, that dictionary would be available for reuse unaltered (or with only bug fixes and additions) in all language pairs where one of the languages is English. Common data would be factored out of language pairs, avoiding redundancy, giving *data decomposition*. Unfortunately, that has not been the case in Apertium until recently.

If a word is in the analyser, but not in the bilingual translation dictionary, certain difficulties arise. As the example above showed, if 'fishes' were unknown to both dictionaries, the output would be `*fishes`, while if it were unknown to only the second, the output would be `@fish`. Given '*fishes', a post-editor who knows both languages can immediately see what the original was, while the half-translated `@fish` hides the number information in the source text. Removing features like number, definiteness or tense can skew meaning. But it gets worse: Some languages inflect verbs for *negation*, where the half-translated lemma would hide the fact that the meaning is negative.[2]

And, as mentioned above, a word not known to the bilingual FST may not have its tags translated (or translated correctly) either; when the transfer module tries to use the half-translated tags to determine agreement, the *context* of the half-translated word may have its meaning skewed as well.

Trying to write transfer rules to deal with half-translated tags also *increases the complexity of transfer rules*. For example, if any noun can be missing its gender, that's one more exception to all rules that apply gender agreement (as well as any feature that interacts with gender).

[2]For simple cases like this, a workaround is to carry surface form information throughout the pipeline, but this fails with multiwords (described below) and compounds, which are heavily used in many Apertium language pairs.

Finally, there are issues with tokenisation and multiwords. Multiwords in Apertium are entries in the dictionaries that may consist of what would otherwise be several tokens. As an example, say you have 'take' and 'out' listed in your English dictionary, and they translate fine in isolation. Now, for Catalan we want to translate the phrasal verb 'take out' into a single word 'treure', so we list it as a *multiword with inner inflection* in the English dictionary. This makes any occurrence of forms of 'take out' get a single-token multiword analysis, e.g. 'takes out' gets the analysis `take<vblex><pri><p3><sg># out`. But then the whole multiword *has* to be in the bilingual dictionary if it is to be translated. If another language pair using the same English dictionary has both 'take' and 'out' in its bilingual dictionary, but not the multiword, the individual words in isolation may be translated, but the whole string together will not be translated.

Due to these issues, most language pairs in Apertium have a separate copy of each monolingual dictionary, manually *trimmed* to match the entries of the bilingual dictionary; so in the example above, if 'take out' did not make sense to have in the bilingual dictionary, it would be removed from the copy of the monolingual dictionary. This of course leads to a lot of redundancy and duplicated effort; as an example, there are currently (as of SVN revision 50180) twelve Spanish monolingual dictionaries in stable (SVN trunk) language pairs, with sizes varying from 36798 lines to 204447 lines.

The redundancy is not limited to Spanish; in SVN trunk we also find 10 English, 7 Catalan, and 4 French dictionaries. If we include unreleased pairs, these numbers turn to 19, 28, 8 and 16, respectively – in the worst case, if you add some words to an English dictionary, there are still 27 dictionaries which miss out on your work. The numbers get even worse if we look at potential new language pairs. Given 3 languages, you "only" need $3 * (3 - 1) = 6$ monolingual dictionaries for all possible pairs (remember that a dictionary provides both an analyser and a generator). But for 4 languages, you need $4 * (4 - 1) = 12$ dictionaries; if we were to create all possible translation pairs of the 34 languages appearing in currently released language pairs, we would need $34 * (34 - 1) = 1122$ monolingual dictionaries, where 34 ought to be enough.

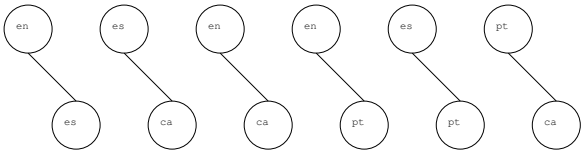

Figure 1: Current number of monodixes with pairs of four languages

The lack of shared monolingual dictionaries also means that other monolingual resources, like disambiguator data, is not shared, since the effort of copying files is less than the effort of letting one module depend on another for little gain. And it complicates the reuse of Apertium's extensive (Tyers et al., 2010) set of language resources for other systems: If you want to create a speller for some language supported by

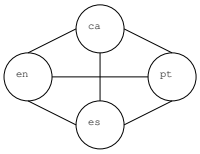

Figure 2: Ideal number of monodixes with four languages

Apertium, you either have to manually merge dictionaries in order to gain from all the work, or (more likely) pick the largest one and hope it's good enough.

1.3. A Solution: Intersection

However, there is a way around these troubles. Finite state machines can be intersected with one another to produce a new finite state machine. In the case of the Apertium transducers, what we want is to intersect the output (or **right**) side of the full analyser with the input (or **left**) side of the bilingual FST, producing a *trimmed* FST. We call this *trimming*.

Some recent language pairs in Apertium use the alternative, Free and Open Source FST framework HFST (Linden et al., 2011)[3]. Using HFST, one can create a "prefixed" version of the bilingual FST, this is is the concatenation of the bilingual FST and the regular expression `.*`, i.e. match any symbol zero or more times. Then the command `hfst-compose-intersect` on the analyser and the prefixed FST creates the FST where only those paths of the analyser remain where the right side of the analyser match the left side of the bilingual FST. The prefixing is necessary since, as mentioned above, the bilingual dictionary is underspecified for inflectional tags such as definiteness, and so on.

The HFST solution works, but is missing many of the Apertium-specific features such as different types of tokenisation FST's, and it does not handle the fact that multiwords may split or change format before bilingual dictionary lookup. Also, HFST represents compounds with an optional transition from the end of the noun to the beginning of the noun dictionary – so if `frog<n>` and `fish<n>` were in the analyser, but `fish<n>` were missing from the bilingual FST, `frog<n>+fish<n>` would remain in the trimmed FST since the prefix matches. In addition, using HFST in language pairs whose data are all in `lttoolbox` format would introduce a new dependency.

Thus we decided to create a new tool within `lttoolbox`, called `lt-trim`. This tool should trim an analyser using a bilingual FST, creating a trimmed analyser, and handle all the `lttoolbox` multiwords and compounds, as well as letting us retain the special tokenisation features of `lttoolbox`. The end result should be the same as perfect manual trimming. The next section details its implementation.[4]

[3]Partly due to available data in that formalism, partly due to features missing from `lttoolbox` like *flag diacritics*.

[4]Available from http://example.com/anonymized-until-peer-review.

2. Implementation of `lt-trim`

The implementation consists of two main parts: preprocessing the bilingual dictionary, and intersecting it with the analyser.

2.1. Preprocessing the bilingual dictionary

Like monolingual dictionaries, bilingual ones can actually define several FST's, but in this case there is no tokenisation – the distinction is only useful for organising the source, and has no effect on processing. So the first preprocessing step is to take the union of these FST's. This is as simple as creating a new FST f, with epsilon transitions from f's initial state, to each initial state in the union, and from each of their final states to f's final state.

Next, we append loopback transitions to the final state. Like mentioned in section 1.1. above, the bilingual dictionary is underspecified for tags. We want an analyser entry ending in `<n><pl>` to match the bilingual entry ending in `<n>`. Appending loopback transitions to the final state, ie. `<n>.*`, means the intersection will end up containing `<n><pl>`. The next section explains the implementation of loopbacks.

The final preprocessing step is to give multiwords with inner inflection the same format as in the analyser. As mentioned in section 1.1., the analyser puts tags after the inflected part, and then the uninflected part of the multiword lemma. The bilingual dictionary (because it has to allow tag prefixes instead of requiring full tag sequences) has the uninflected part before the tags. Section 2.3. details how we move the uninflected part after the tags in preprocessing the bilingual dictionary.

2.2. Prefixing the bilingual dictionary

Apertium alphabets consist of symbol pairs, each with a left (serves as the input in a transducer) and right (output) symbol. Both the pairs and the symbols themselves, which can be either letters or tags, are identified by an integer. First, the identifiers of identical left-right pairs of the desired symbols are determined. In the case of intersecting transducers using depth-first traversal, the method implemented in Apertium, only the tags are desired, though the option to include letter pairs as well still exists due to the deprecated multiplicative method. The side from which the symbols are obtained is also able to be specified, though in the case of prefixing a bilingual dictionary, only the right (output) symbols are used. All of the symbol-pairs of the given alphabet are looped-through, and depending on which side was specified, the respective symbols are analysed. The method differs between letters and tags. As the identifiers of letters, which are actually the letters themselves cast to integers, are consistent throughout all alphabets, the identifiers of letter pairs can be directly determined. The identifiers of tags, however, can differ, and so their individual identifiers must first be determined before that of their respective pairs. After the identifiers of the desired symbol pairs are determined, they are used to create loopbacks on the bilingual dictionary using the function $appendDotStar$. Transitions are created from the final state of the bilingual transducer that loop directly back with each of the identifiers.

2.3. Moving uninflected lemma parts

To turn `take# out<vblex>` into `take<vblex># out` and so on (and for longer tag sequences too), we do a depth-first traversal looking for an occurrence of the # symbol. Then we replace the #-transition t with one into the new transducer returned by $copyWithTagsFirst(t)$.

This function traverses the FST from the target of t, building up two new transducers, new and $lemq$. We keep a $SearchState$ during traversal, which is a pair of the current source state in the original FST and the last added state in $lemq$. Until we see the first tag, we add all transitions found to $lemq$, and record in the $SearchState$ which was the last $lemq$ state we saw. Upon seeing the first tag, we start adding transitions from the initial state of new,[5] and don't change the $lemq$-part of the $SearchState$.

When reaching a final tag state t, we add it and the last seen $lemq$ state l to a list of pairs f. After the traversal is done, we loop through each (t, l) in f, creating a temporary copy of $lemq$ where the $lemq$-state l has been made the only final state, and adding an epsilon-transition from each final tag state t in new into that copy. Finally, we return new from $copyWithTagsFirst$. Thus several $lemq$ paths may lead from the # to the various first tag states, but we only connect those paths which were connected in the original bilingual dictionary.

2.4. Intersection

The intersection implemented as a depth-first traversal of the analyser and the bilingual FST in lockstep; that is, we only follow transitions which are possible in both FST's. For the possible paths, we create new transitions in the new, trimmed FST.

3. Ending Dictionary Redundancy

As mentioned in section 1.3., there are already language pairs in Apertium that have moved to a decomposed data model, using the HFST trimming method. At first, the HFST language pairs would also copy dictionaries, even if they were automatically trimmed, just to make them available for the language pair. But over the last year, we have created GNU Autotools scripts that let a language pair have a formal dependency on one more monolingual data packages[6]. There is now an SVN module `languages`[7] where such monolingual data packages reside, and all of the new HFST-based languages pairs now use such dependencies, which are trimmed automatically, instead of making redundant dictionary copies. Disambiguation data is also fetched from the dependency instead of being redundantly copied.

Most of the released and "stable" Apertium language pairs use `lttoolbox` and still have dictionary redundancy. With the new `lt-trim` tool, it is finally possible to end the redundancy[8] for the pairs which use `lttoolbox`, with its tokenisation, multiword and compounding features, and without having to make those pairs dependent on a whole other FST framework simply for compilation.

The tool has only recently been released, and there is still much work to do in converting existing language pairs to a decomposed data model. Monolingual dictionaries have to be merged, and the various language pairs may have altered the tag sets in more or less subtle ways that can affect disambiguation, transfer and other parts of the pipeline. However, this is a one-time job, and any new languages added to Apertium can immediately reap the benefits.

Acknowledgements

Part of the development was funded by the Google Code-In[9] programme.

4. References

Mikel L. Forcada, Mireia Ginest-Rosell, Jacob Nordfalk, Jim ORegan, Sergio Ortiz-Rojas, Juan Antonio Prez-Ortiz, Felipe Snchez-Martnez, Gema Ramrez-Snchez, and Francis M. Tyers. 2011. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, 25(2):127–144.

Krister Linden, Miikka Silfverberg, Erik Axelson, Sam Hardwick, and Tommi Pirinen, 2011. *HFSTFramework for Compiling and Applying Morphologies*, volume Vol. 100 of *Communications in Computer and Information Science*, pages 67–85.

Antonio Toral, Mireia Ginest-Rosell, and Francis Tyers. 2011. An Italian to Catalan RBMT system reusing data from existing language pairs. In F. Snchez-Martnez and J.A. Prez-Ortiz, editors, *Proceedings of the Second International Workshop on Free/Open-Source Rule-Based Machine Translation*, pages 77–81, Barcelona, Spain, January.

Francis M. Tyers, Felipe Snchez-Martnez, Sergio Ortiz-Rojas, and Mikel L. Forcada. 2010. Free/Open-Source Resources in the Apertium Platform for Machine Translation Research and Development. *Prague Bull. Math. Linguistics*, 93:67–76.

[5] We keep state-mapping tables so we can refer to the new state corresponding to each original FST state.

[6] So if a user asks their package manager, e.g. `apt-get`, to install the language pair `apertium-foo-bar`, it would automatically install dependencies `apertium-foo` and `apertium-bar` first.

[7] http://wiki.apertium.org/wiki/Languages

[8] One could argue that there is still *cross-lingual* redundancy in the bilingual dictionaries – Apertium by design does not use an interlingua. Instead, the Apertium dictionary crossing tool `crossdics` (Toral et al., 2011) provides ways to extract new translations during development: Given bilingual dictionaries between languages A-B and B-C, it creates a new bilingual dictionary between languages A-C. One argument for not using an interlingua during the translation process is that the dictionary resulting from automatic crossing needs a lot of manual cleaning to root out false friends, unidiomatic translations and other errors – thus an interlingua would have to contain a lot more information than our current bilingual dictionaries in order to automatically disambiguate such issues. It would also require more linguistics knowledge of developers and heighten the entry barrier for new contributors.

[9] https://code.google.com/gci/