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

The Free and Open Source rule-based machine translation platform Apertium uses Finite State Transducers (FST's) for analysis, where the output of the analyser is input to a second, *bilingual* FST. The bilingual FST is used to translate analysed tokens (lemmas and tags) from one language to another. We discuss certain problems that arise if the analyser contains entries that do not pass through the bilingual FST. In particular, in trying to avoid "half-translated" tokens, and avoid issues with the interaction between multiwords and tokenisation, language pair developers have created redundant copies of monolingual dictionaries, manually customised to fit their language pair. This redundancy gets in the way of sharing of data and bug fixes to dictionaries between language pairs. It also makes it more complicated to reuse dictionaries outside Apertium (e.g. in spell checkers). We introduce a new tool to *trim* the bad entries from the analyser (using the bilingual FST), creating a new analyser. The tool is made part of Apertium's `lttoolbox` package.

Keywords: FST, intersection, RBMT, dictionary redundancy

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. This paper discusses the problem of redundancy in monolingual dictionaries in Apertium, and introduces a new tool to help solve it.

The following sections give some background on how FST's fit into Apertium, as well as the specific capabilities of `lttoolbox` FST's; then we delve into the problem of monolingual and bilingual dictionary mismatches that lead to redundant dictionary data, and present 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 tag suffix (here `<pl>`) will simply be carried over.

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, and the automatic tokenisation of all **lexical units**. A lexical unit may be

- a simple, non-multi-word like the noun 'fish',
- a space-separated word like the noun 'hairy frogfish', which will be analysed as one token even though it contains a space, 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

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

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 `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 files 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 of the analyser would be `@fish<n><pl>`, and of the complete translation just `@fish`. Given '*fishes', a post-editor who knows both languages can immediately see what the original was, while the half-translated `@fish` hides the inflection information in the source text. Just lemmatising the source text, which removes 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 might 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).

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 whenever the whole string together is seen, it will only be lemmatised, not 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 36,798 lines to 204,447 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

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

would need $34 * (34-1) = 1122$ monolingual dictionaries, where 34 ought to be enough.

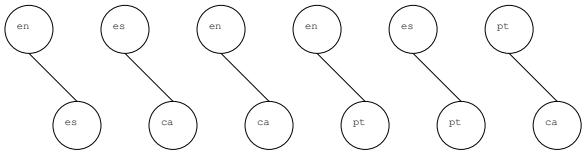

Figure 1: Current number of monodixes with pairs of four languages

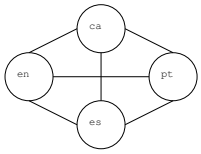

Figure 2: Ideal number of monodixes with four languages

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 so 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 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 process *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 tag suffixes (typically inflectional tags such as definiteness or tense, as opposed to lemma-identifying tags such as part of speech and noun gender).

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, unlike `lttoolbox`, `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 `frog<n>.*` matches. In addition, using `HFST` in language pairs whose data are all in `lttoolbox` format would introduce a new (and rather complex) dependency both for developers, packagers and users who compile from source.

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 the implementation of `lt-trim`.[4]

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 the input is already tokenised – 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 (empty/unlabelled) 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>`; we call the bilingual dictionary *prefixed* when it has the loopback transitions appended. The next section explains the implementation of prefixing.

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 part of the lemma corresponding to the inflected part, with the uninflected part of the multiword lemma coming last.[5] The bilingual dictionary has the uninflected part before the tags, since it has to allow tag prefixes instead of requiring full tag sequences. Section 2.3. details how we move the uninflected part after the (prefixed) tags in preprocessing the bilingual dictionary.

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

[4] Available in the SVN version of the `lttoolbox` package, see http://wiki.apertium.org/wiki/Installation.

[5] Having the tags "lined up with" or at least close to the inflection they represent eases dictionary writing.

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 states 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, 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 (in the example above, the state before the space), 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. In the example above, we would build up a $lemq$ transducer containing `out` (with an initial space). Upon seeing the first tag, we start adding transitions from the initial state of new,[6] and don't change the last-$lemq$-state of the $SearchState$.

When reaching a final tag state s, 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 (s, l) in f, creating a temporary copy of $lemq$ where the $lemq$-state l has been made the only final state, and adding a #-transition from each final tag state s in new into that copy. In the example, we would copy the $lemq$ into one where the state after the letter `t` were made final, and insert that copy after the final state s, the state after the `<vblex>.*`. If, from the original #, there were a $lemq$ path that didn't have the same last-$lemq$-state (e.g. `take# out<n>.*` or even `take# out.*`) it would end up in a state that were not final after s, and the path would not give any analyses (such paths are removed by FST *minimisation*). But if a $lemq$ path did have the same last state, we would want it included, e.g. `take# part<vblex>.*` to `take<vblex>.*# part`. 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.

Finally, we return new from $copyWithTagsFirst$ and look for the next #.

2.4. Intersection

The first method we tried of intersecting FST's consisted of multiplying them. This is a method that is simple to prove correct, however, it was extremely inefficient, requiring a massive amount of memory. First, the states of each transducer were multiplied. This meant that every possible state pair, consisting of a state from the monolingual dictionary and the bilingual dictionary, was assigned a state in the trimmed transducer. Next, each of the transitions were multiplied. As the intersection is only concerned with the output of the monolingual dictionary and the input of the bilingual dictionary, the respective symbols had to match; though very many of them did not, a significant number of matching symbols resulted in transitions to redundant and unreachable states. These were removed with minimisation.

The tool now implements the much more efficient method of intersection, depth-first traversal. Both the monolingual dictionary and the bilingual dictionary are traversed at the same time, in lockstep; only transitions present in both are further followed and added to the trimmed transducer. The process is managed through noting which states are to be processed, the pair of states currently being processed, the next pair of states, and the states which have already been seen. Besides having reachable counterparts in both transducers, a state pair will only be added to the queue if it has not been seen yet. However, to handle multiwords, a few other things are necessary.

If a + is encountered in the monolingual dictionary (indicating a +-type multiword) the traversal of the bilingual dictionary resumes from its beginning. In addition, in the event that a # is later encountered, the current position is recorded. The #-type multiwords alone can be easily handled if the bilingual dictionary is preprocessed to be in the same format as the monolingual dictionary; the tags must be moved before the #. However, a combination of both + and # requires the traversal of the bilingual dictionary return to the state at which the + was first encountered.

We also need some exceptions for epsilon transitions; the idea here is that if we see an epsilon in one transducer, we move across that epsilon without moving in the other transducer, and vice versa. Compound symbols in the analyser are treated similarly: we add the transitions to the trimmed analyser and move forward in the analyser without moving forward in the bilingual FST.

To the end user (i.e. language pair developer), the tool is a simple command line program with three arguments: the input analyser FST, the input bilingual FST and the output

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

trimmed analyser 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 scripts for our GNU Autotools-based build system that let a language pair have a formal dependency on one or more monolingual data packages[7]. There is now an SVN module `languages`[8] 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.

But 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[9] 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[10] programme.

[7] This both means that source and compiled monolingual files are made available to the make files of the language pair, and that the configure script warns if monolingual data packages are missing. Packagers should be able to use this so that 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, and use files from those packages.

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

[9] 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.

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

4. References

Mikel L. Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, 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. *HFST—Framework for Compiling and Applying Morphologies*, volume Vol. 100 of *Communications in Computer and Information Science*, pages 67–85.

Antonio Toral, Mireia Ginestí-Rosell, and Francis Tyers. 2011. An Italian to Catalan RBMT system reusing data from existing language pairs. In F. Sánchez-Martínez and J.A. Pérez-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 Sánchez-Martínez, 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.