Thrax: An Open Source Grammar Compiler Built on OpenFst

Terry Tai[1], Richard Sproat[1], Wojciech Skut[2]

Google, Inc
[1]*New York, NY, USA*
[2]*Zurich, Switzerland*
{ttai,rws,wojciech}@google.com

***Abstract*—We present Thrax, an open source finite-state grammar compiler built on OpenFst. We describe the functionality of the language and the underlying algorithms, and briefly discuss some of its applications inside Google. Thrax is a work in progress, and it is hoped that a large user community will develop around the use of Thrax, and that this in turn will lead to improvements in the functionality and design.**

I. Introduction

The OpenGrm Thrax Grammar Compiler is a set of tools for compiling grammars expressed as regular expressions and context-dependent rewrite rules into weighted finite-state transducers, built on top of the OpenFst library [1]. It is named after Dionysius Thrax (170 BC–90 BC), the first Greek grammarian.

Let's say you need a text normalization system for a text-to-speech system and you need (among other things) to define how to read digit sequences as numbers: "113" is to be read as "one hundred thirteen". One might write special-purpose code (in C++, Java, etc.) to perform this transduction, but it is far easier and far more general to use finite-state methods. One could of course use a library, such as OpenFst, which provides core finite-state algorithms and build one's transducers directly. But this is very hard to do, and it is almost always desirable to have tools that can compile a high level description of rules into the FSTs that do the actual work.

There have been many previous toolkits for grammar compilation into FSTs including PC-KIMMO [2], lextools [3], XFST [4], the HFST http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/, among others. Why a new toolkit?

The initial reason lay in our need to have a toolkit for various Google-internal applications (Section V), based on the Google finite-state transducer (FST) library. The latter has been open-sourced as OpenFst and has been used in a wide variety of applications, and this led to our second motivation, namely to provide a grammar-development toolkit that is fully integrated with this popular open-source FST project. However, we also took this as an opportunity to develop a number of often-requested features to enchance the rule-writing experience.

In the remainder of this paper we start with an example grammar that illustrates the workings of the system. We then provide a detailed description of the functionality of the system, describe some applications of Thrax within Google, and end by discussing future work.

II. Grammar Fragment

The appendix contains a Thrax grammar fragment for a simple grammar that tokenizes input (splitting off punctuation, and so forth), expands some numbers into words, and transduces into "Earnest" the real names of characters who pass themselves off as Earnest in Oscar Wilde's play. The grammar is heavily commented and thus should be easy to follow. An example of the behavior of the grammar can be seen in the following examples using the tool `thraxrewrite-tester`:

```
$ thraxrewrite-tester --far=example.far --rule=TOKENIZER
Input string: Well, I can't eat muffins in an agitated manner.
Output string: Well , I ca n't eat muffins in an agitated manner .
Input string: Mr. Ernest Worthing, B. 4, The Albany.
Output string: Mr. Ernest Worthing , B. four , The Albany .
Input string: Lieutenant 1840,
Output string: Lieutenant eighteen forty ,
Input string: Uncle Jack!
Output string: Uncle Ernest !
```

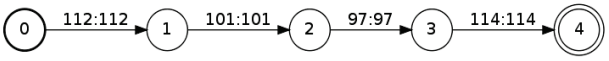
III. General Description

In this section we give a quick introduction to the language and its syntax. More in-depth tutorials can be found with the actual Thrax package.

Thrax revolves around rules which, typically, construct an FST based on a given input. In the simplest case, we can just provide a string that represents a (trivial) transducer and name it using the assignment operator:

```
pear = "pear";
```

In this example, we have an FST consisting of the characters "p", "e", "a", and "r" in a chain, assigned to the identifier `pear`:



This identifier can be used later in order to build further FSTs, using built-in operators or using custom functions:

```
kiwi = "kiwi";
fruits = pear | kiwi;  # Union of the two.
```

In Thrax, string FSTs are enclosed by double-quotes (`"`) whereas simple strings (often used as pathnames for functions) are enclosed in single-quotes (`'`).

Thrax provides a set of built-in functions that aid in the construction of more complex expressions. We have already seen the disjunction "|" in the previous example. Other standard regular operations are *expr**, *expr+*, *expr?* and *expr*{*m,n*}*,

the latter repeating *expr* between *m* and *n* times, inclusive. Composition is notated with "@" so that *expr1 @ expr2* denotes the composition of *expr1* and *expr2*. Rewriting is denoted with ":" where *expr1 : expr2* rewrites strings that match *expr1* into *expr2*. Weights can be added to expressions using the notation "<>": thus, *expr<2.4>* adds weight *2.4* to *expr*.

Various operations on FSTs are also provided by built-in functions, including `Determinize`, `Minimize`, `Optimize` and `Invert`. Thus:

```
Invert[fst];
```

inverts the named fst. There are many other functions, including `LoadFst` (loads a named FST from a file), `StringFile` (provides efficient compilation of a list of strings stored in a file) and `SymbolTable` (loads a symbol table from a file).

IV. Key Features

A. Imports

The Thrax compiler compiles source files (with the extension `.grm`) into *FST archive* files (with the extension `.far`). FST archives are an OpenFst storage format for a series of one or more FSTs. The FST archive and the original source file then form a pair which can be imported into other source files, allowing a Python-esque include system that is hopefully familiar to many. Instead of working with a monolithic file, Thrax allows for modular construction of the final rule set as well as sharing of common elements across projects.

B. Functions

Thrax has extensive support for functions that can greatly augment the capabilities of the language. Functions in Thrax can be specified in two ways. The first is inline via the *func* keyword within `grm` files. These functions can take any number of input arguments and must return a single result (usually an FST) to the caller via the *return* keyword:

```
func DumbPluralize[fst] {
  # Concatenate with "s"...
  result = fst "s";
  # ...and then return to caller.
  return result;
}
```

Alternatively, functions can be written C++ and added to the language. This allows users to dramatically extend the Thrax language to perform custom tasks, ranging from relatively simple things like custom I/O to complicated procedures such as distributed datacenter lexicon lookups. This is accomplished by subclassing (in C++) the `thrax::function::Function` class and calling the proper registration macros.

Regardless of the function implementation method (inline in Thrax or subclassed in C++), functions are integrated into the Thrax environment and can be called directly by using the function name and providing the necessary arguments. Thus, assuming someone has written a function called `NetworkPluralize` that retrieves the plural of a word from some website, one could write a grammar fragment as follows:

```
apple = "apple";
plural_apple = DumbPluralize[apple];

plural_tomato = NetworkPluralize[
    "tomato",
    'http://server:port/...'];
```

C. FST String Encodings

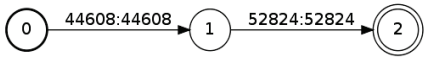
Thrax supports three methods of string FST input. The default is to create arcs and labels using normal ASCII bytes, where

```
pear = "pear".byte;
```

would generate the FST shown above. Thrax also works with UTF-8 encoded unicode. In that case one can specify that the string is to be parsed as a UTF-8 string:

```
kimchi = "김치".utf8;
```

The individual arc labels in that case will be the Unicode code points corresponding to the individual Unicode characters:



Finally, we found that we often wanted to represent whole words (or other non-character units) as individual arcs in our FSTs. To this end, Thrax can also load symbol tables, which provide mappings from symbols (any string) to labels (any integer). This allows for larger bodies of text to be compactly represented.

```
# Load up symbol table.
en_symtab = SymbolTable[
    '/path/to/symbol-table'];

# sentence will have 3 arcs with labels
# matching en_symtab's mapping for "figs",
# "are", and "yummy".
sentence = "figs are yummy".en_symtab;
```

D. Context-Dependent Rewrites

One very important function is the *context-dependent rewrite* function `CDRewrite`. Given a weighted transducer specifying a mapping between two regular languages, two unweighted context acceptors, and a machine describing the transitive closure of the alphabet (i.e. Σ^*), `CDRewrite` generates a new FST that performs a context dependent rewrite everywhere in the provided contexts. The context-dependent rewrite algorithm used is that of [5]. The fourth argument (`sigma_star`) must be a minimized machine:[1]

```
CDRewrite[tau, lambda, rho,
          sigma_star]
```

[1]An optional fifth argument selects the direction of rewrite; we can either rewrite left-to-right (default) or right-to-left or simultaneously. An optional sixth argument selects whether the rewrite is optional. It can either be obligatory (default) or optional

Thus for example:

```
CDRewrite["cheese" : "gouda", "Dutch ",
          "", sigma_star];
```

specifies a left-to-right rule that maps "cheese" into "gouda" obligatorily if the left context is "Dutch".

E. Temporary or "Extended" Symbols

It is sometimes useful to be able to insert "bookmarks" into strings in order to keep track of key locations, and it can also be useful if those bookmarks are represented by symbols that are outside the normal alphabet of the grammar. One can define extended symbols in Thrax by enclosing them in square brackets. All of the text inside the brackets will be taken to be part of the symbol name, and future encounters of the same symbol name will map to the same label:

```
cross_pos = "cross" ("" : "[s_noun]");
pluralize_nouns = "[s_noun]" : "es";
```

V. Usage Within Google

Thrax is used extensively within Google on a number of speech-related projects, including Voice Search [6], Voicemail Transcription, and text-to-speech synthesis. In voice search, for example, one needs to compute how various search terms are likely to be read, including such things as URLs and numbers.

For numbers in particular, we have built infrastructure that allows us to rapidly develop number grammars that map from digit strings to number names: this "123" might map into "one hundred twenty three" in English. We have developed a set of grammars that map between digit strings and possible factorizations into sums of products of powers of tens, modeling the various ways in which different groups of languages factor these numbers. We have provided factorization schemes that work for Western languages and others where the powers of ten that have simple lexical expressions are 1, 2, 3, 6, 9, 12; for East Asian languages that use 1, 2, 3, 8, 12; and for Indian languages that use 1, 2, 3, 5, 7, 9 .... Following [3], these grammars are incorporated into language-specific grammars that specify the lexical items associated with each number or power of ten and also gives rules about their combination. Here, for example, is a fragment of the grammar for English, where expressions like `[E3]` denote the powers of ten:

```
cardinal_vocabulary =
    ("0" : "zero")
  | ("1" : "one")
  | ("2" : "two")
  | ("3" : "three")
...
  | ("1[E1]" : "ten")
  # Negative cost to favor "eleven" over "ten one"
  | ("1[E1]1" : "eleven" <-0.1>)
  | ("1[E1]2" : "twelve" <-0.1>)
...
  | ("2[E1]" : "twenty")
  | ("3[E1]" : "thirty")
...
  | ("[E2]" : "hundred")
  | ("[E3]" : "thousand")
...
;
```

Using this infrastructure we have developed number grammars for over 25 languages to date.

VI. Open Source Project

Thrax is available from OpenGrm.org. It is fully open-sourced, and released under the Apache 2.0 licence (http://www.apache.org/licenses/LICENSE-2.0). The distributed system includes documentation and example grammars. More material will be added in the future, and we will be releasing periodic updates to the system as there are further developments to the code base internally.

VII. Further Work

As noted in the abstract, Thrax is a work in progress. While the tools as they stand allow for one to develop highly complex grammars, there is much scope for improvement.

One area to improve would be to provide more functionality for handling complex linguistic systems. The lextools package [3] included mechanisms for handling linguistic features (e.g. notations such as `[masc,pl]`), as well as a tool for handling paradigms. In lextools one could define a paradigm in terms of, say, a set of endings; one could then define a second paradigm in terms of the first, where the properties of the first would be inherited by the second, except in a set of specified cases. No such functionality exists in Thrax at present, but this is something that is useful to add, since it makes the development of systems of complex inflectional morphology much easier.

Another area of improvement would be grammars to provide infrastructure for the processing of scripts from any section of the Unicode Basic Multilingual Plane. At present, some functionality for defining character classes is provided as part of the standard library, but this mostly applies to characters in the ASCII range.

Acknowledgment

We thank Cyril Allauzen for his implementation of the context-dependent rewrite algorithm from [5]. Martin Jansche wrote the efficient string compiler underlying the `StringFile` function.

References

[1] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri, "OpenFst: A general and efficient weighted finite-state transducer library," in *Proceedings of the Twelfth International Conference on Implementation and Application of Automata, (CIAA 2007)*, vol. 4783. Prague, Czech Republic: Springer, 2007, pp. 11–23.

[2] E. Antworth, *PC-KIMMO: A Two-Level Processor for Morphological Analysis*, ser. Occasional Publications in Academic Computing, 16. Dallas, TX: Summer Institute of Linguistics, 1990.

[3] R. Sproat, "Multilingual text analysis for text-to-speech synthesis," *Natural Language Engineering*, vol. 2, no. 4, pp. 369–380, 1997.

[4] K. Beesley and L. Karttunen, *Finite State Morphology*. Stanford, CA: CSLI Publications, 2003.

[5] M. Mohri and R. Sproat, "An efficient compiler for weighted rewrite rules," 1996, pp. 231–238.

[6] J. Schalkwyk, D. Beeferman, F. Beaufays, B. Byrne, C. Chelba, M. Cohen, M. Kamvar, and B. Strope, "Google search by voice: A case study," in *Visions of Speech: Exploring New Voice Apps in Mobile Environments, Call Centers and Clinics*, A. Neustein, Ed. Springer, 2010.

APPENDIX: GRAMMAR FRAGMENT

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# Copyright 2005-2011 Google, Inc.
# Author: rws@google.com (Richard Sproat)

import 'byte.grm' as bytelib;
import 'numbers.grm' as numbers;

# Import grammars from byte.grm and numbers.grm into their own named namespaces
# ("as <namespace>"). The way this is written it is assumed that byte.grm and
# numbers.grm exist in the directory in which this file exists, and that you are
# building in this directory.
#
# To compile this grammar, the easiest way is to run
#
# thraxmakedepend example.grm
#
# And then run "make". "thraxmakedepend" is a python script that reads in a
# top-level grammar, figures out the dependencies, and writes out a makefile (by
# default named "Makefile") that contains the relevant targets and build
# commands, consisting of calls to "thraxcompiler". For this grammar, the
# Makefile should look as follows:
#
# example.far: example.grm byte.far numbers.far
# thraxcompiler --input_grammar=$< --output_far=$@
#
# byte.far: byte.grm
# thraxcompiler --input_grammar=$< --output_far=$@
#
# numbers.far: numbers.grm byte.far
#  thraxcompiler --input_grammar=$< --output_far=$@
#
# clean:
#  rm -f byte.far numbers.far
#
# As indicated, the final target is the FST archive example.far, which will
# contain a single fst named TOKENIZER.fst (see below). After compiling the
# whole suite of grammars, it is safe to run "make clean", which will remove all
# but the top-level FST archive. (Of course if you do that and need the other
# archives later on you will have to rebuild them.)
#
# If one wishes to build the grammar in a different directory from the one where
# one or more of the grammars reside, then one must specify pathnames that are
# either relative to the build directory, or else full pathnames. For example:
#
# import '../byte.grm' as bytelib;
#
# In any case, one must have read and write permissions to the directories since
# thraxcompiler will by default build fars in the same directory as the grammar.
#
# To test this grammar one can use thraxrewrite-tester as follows:
#
# $ thraxrewrite-tester --far=example.far --rule=TOKENIZER
# Input string: Well, I can't eat muffins in an agitated manner.
# Output string: Well , I ca n't eat muffins in an agitated manner .

# Simple example of a mapping between strings: this transducer does nothing
# except insert a space:

insspace = "" : " ";

# Use the transducer defined by kSpace in byte.grm to map a sequence of one or
# more spaces to exactly one space:

reduce_spaces = bytelib.kSpace+ : " ";

# This deletes a sequence of zero or more spaces:

delspace = bytelib.kSpace* : "";

# For context-dependent rewrite rules, one must specify the alphabet over which
# the rule is to apply. This should be specified as the transitive closure of
# all characters that one might expect to see in the input. The simplest way to
# do this in general is to allow it to consist of any sequence of bytes as
# below. This is not necessarily the most efficient way to specify it. If you
# know that your input will be much more restricted, then you can specify a
# smaller alphabet, which in turn will yield gains in compilation efficiency:

sigma_star = bytelib.kBytes*;

# This rule illustrates the difference operator. One can specify the difference
# between any pair of regular expressions that specify *acceptors*. In this case
# we specify any punctuation symbol that is not a period:

punct_not_period = bytelib.kPunct - ".";

# Here are some examples of context-dependent-rewrite rules. The basic format
# for a context-dependent rewrite rule is:
#
# CDRewrite[change, left_context, right_context, sigma_star]
#
# where change is a transducer that specifies a mapping between input and
# output.
#
# In phonological rewrite rule notation this would be:
#
# input -> output / left_context _ right_context
#
# where "change" is a transducer, left_context and right_context acceptors, and
# sigma_star as described above. This specifies a left-to-right obligatory
# rewrite rule. See the documentation for other options.

#
# In the first rule below the change is to insert a space, the left context
# anything at all (hence null) and the right context the acceptor specified
# above as punct_not_period:

separate_punct1 = CDRewrite["" : " ", "", punct_not_period, sigma_star];

# Similar to the above, except that we insert a space after the punctuation
# symbol:

separate_punct2 = CDRewrite["" : " ", punct_not_period, "", sigma_star];

# The following illustrates composition: we construct the rule separate_punct by
# composing separate_punct1 with separate_punct2:

separate_punct = separate_punct1 @ separate_punct2;

# This rule illustrates the use of the string delimiter tag "[EOS]". Assuming
# that any sentence internal period marks something like an abbreviation or
# maybe a decimal number, we only want to split off final periods. These may be
# followed by spaces or other puncutation symbols. So the following rule states
# that we insert a space when followed by a period that is itself followed by an
# optional string of spaces or punctuation symbols, followed by the end of the
# string. For the left context, one may specify the beginning of the string as
# "[BOS]".

separate_final_period = CDRewrite["" : " ",
                                  "",
                                  "."
                                  (bytelib.kPunct | bytelib.kSpace)* "[EOS]",
                                  sigma_star]
;

# The following rule composes two transducers, and optimizes the
# result. Optimize performs various optimizations on the transducer: removing
# epsilon arcs, summing arc weights, determinizing and minimizing. The resulting
# transducers is in general more compact and efficient. Especially in large
# grammars, it is a good idea to optimize at least some of the intermediate
# transducers. This can significantly speed up compilation.

first_phase = Optimize[separate_punct @ separate_final_period];

# This illustrates the use of the StringFile functionality. A stringfile is
# simply a list of literal strings:
#
# string1
# string2
# string3
# ...
#
# It is equivalent to the following disjunction:
#
# string1 | string2 | string3 | ...
#
# but for very large lists it is much more efficient.

ernest_equivalents = StringFile['ernest.txt'];

ernest = ernest_equivalents : "Ernest";

anyword = bytelib.kNotSpace+;
number = numbers.NUMBERS;

# We define a "word" as either "anyword", as above or a number, handled by the
# number grammar in numbers.grm, or the (silly) ernest rule. This rule
# illustrates the use of weights. We want to favor a number if it matches the
# input. That is we want "123" to map to "one hundred twenty three", not to
# "123". We can implement this using weights by disfavoring the "anyword"
# analysis. For any non-space sequence, the analyzer will allow both analyses,
# but the "anyword" analysis will always be disfavored. So when the TOKENIZER
# rule (below) is composed with a string, the best (shortest path) analysis will
# be the one with the number analysis, if that's available. Weights are
# interpreted according to the tropical (+, min) semiring (see
# http://www.openfst.org under "FST Weights"), so a weight of <1.0> as below
# will disfavor "anyword", which has an implicit weight of 0. Negative weights
# are also allowed, so this could have been written:
#
# word = anyword | (number <-1.0>) | ernest;
#
# Note that the grouping parentheses are *required* in this instance:

word = (anyword <1.0>) | number | ernest;

# A sentence consists of a sequence of words interspersed with spaces. Initial
# and final spaces get deleted, and interword spaces get reduced to exactly one
# space:

second_phase = delspace
               word
               (reduce_spaces word)*
       delspace
;

# Some sequences involving apostrophes are overzealously tokenized. These rules
# fix that and produces more reasonable tokenizations. Note that the grouping
# parentheses are required when one is construction a disjunction of
# transductions:

regroupings = ("n ' t" : " n't") | (" ' s" : " 's") ;

final_phase = CDRewrite[regroupings, "", (bytelib.kSpace | "[EOS]"),
                        sigma_star];

# The whole tokenizer is the composition of the three phases. We export this
# rule since we want it available in the final FST archive:

export TOKENIZER = Optimize[first_phase @ second_phase @ final_phase];
```