A Large Scale Format Compliance Checker for TEX Font Metrics

Didier Verna

Abstract

We present `tfm-validate`, a TEX Font Metrics format checker. The library's core functionality is to inspect TFM files and report any discovered compliance issue. It can be run on individual files or complete directory trees. `tfm-validate` also provides a convenience function to (in)validate a local TEX Live installation. When run this way, the library processes every TFM file in the distribution and generates a website aggregating all the discovered non-compliance issues. One public instance of `tfm-validate` is now automatically triggered on a daily basis. The corresponding website is available at `texlive.info/tfm-validate/`.

1 Introduction

As part of ETAP[1], our experimental typesetting algorithms platform [8, 9], we have developed a parser for TFM (TEX Font Metrics) files, simply called `tfm`. To ensure robustness, a parser for an official data format must be prepared to handle all sorts of compliance problems, with varying degrees of seriousness ranging from simple warnings to non-recoverable errors. `tfm` not only provides a rich (hopefully exhaustive) ontology of errors, but also a powerful recovery mechanism, allowing to proceed for as long as possible with the parsing, for example by fixing errors on the fly or discarding problematic input.

A side-effect of `tfm`'s robustness is that it is possible to use it as a validation tool rather than for actually loading font information. Indeed, the `tfm` exception handler reifies the problematic situations into objects (in the "Object-Oriented" sense) which can be silently collected until the parsing is over or needs to be terminated prematurely. These objects can in turn be used to produce a full compliance report for the analyzed file. We have automated this process for the whole TEX Live distribution, resulting in the (in)validation of almost 80000 fonts, and the generation of a website providing direct access to the generated compliance reports.

This paper is organized as follows. Section 2 provides an overview of the `tfm` library and explains how it is made robust. Section 3 describes the very peculiar exception handling mechanism in use, and how it simplifies the design of `tfm-validate` considerably. Finally, Section 4 analyzes the results of the TFM validation process applied to the whole TEX Live distribution.

2 The `tfm` Library

The `tfm`[2] library was designed to bring TEX Font Metrics information to Common Lisp [1] applications. It essentially provides an entry point function called `load-font`, which takes a file name as argument and returns a data structure containing an abstract representation of the contents of the TFM file. A full description of the library is out of the scope of this paper. The interested reader will find a complete user manual in the distribution, as well as online. The important thing for this discussion is that `tfm` aims at being both robust and flexible.

2.1 Robustness

Robustness for a parser means that it should be prepared to handle *all* the possible problematic situations, for example in order to abort loading the culprit font file and exit gracefully, rather than just crashing or behaving erratically. During the development of `tfm`, we have identified twenty such situations, with varying degrees of severity.

Examples of critical situations include truncated files or invalid section pointers, making it impossible to know exactly where to find character, ligature, kerning information, *etc.* In those situations, there is nothing clever one can do to make the bogus font functional.

A less critical, yet problematic situation, would be the detection of a cycle in a ligature program, resulting in an infinite loop when attempting the ligature. In such a case, we can still hope to get a functional (although incomplete) font if we just forget about the ligature(s).

Non critical situations might be inconsistencies in parts of the TFM file which are purely informative (such as several places in the header) and not actually used to render the font. TEX itself simply ignores a number of such situations and proceeds normally.

Finally, note that the severity of a problem may depend on the context. One interesting such case is that of the font's design size. The TFM format requires it to be greater than 1. At the same time, TEX allows the design size to be overridden by the user (this is what happens when you say `\font\foo=cmr10 at 12pt` for example). An invalid design size is normally an error, but it doesn't really hurt when overridden by a correct one. Hence, the `tfm` library signals an error in the former case, but only a warning in the latter.

[1] `github.com/didierverna/etap`

[2] `github.com/didierverna/tfm`

```
CL-USER> (tfm:load-font "/tmp/cmr10.tfm")
While reading /tmp/cmr10.tfm,
while reading the character encoding scheme string,
padded string "TeX (ex)" is not in BCPL format.
See §10 of the TFtoPL documentation for more information.
   [Condition of type NET.DIDIERVERNA.TFM:INVALID-PADDED-STRING]

Restarts:
 0: [KEEP-STRING] Keep it anyway.
 1: [FIX-STRING] Fix it using /'s and ?'s.
 2: [DISCARD-STRING] Discard it.
 3: [CANCEL-LOADING] Cancel loading this font.
 --more--
```

Figure 1: Sample interactive recovery session

2.2 Flexibility

Flexibility for a parser means that *when possible*, it should provide less drastic ways to recover from problems than just giving up. `tfm` currently provides a dozen recovery options, the availability of which depends on the situation.

As mentioned previously, it is possible to discard a ligature or a kerning instruction rather than aborting the whole loading process if something is wrong (like an invalid character code). Another example is the requirement that the width, height, depth and italic corrections tables all start with a first value of 0. When appropriate, `tfm` offers to fix a bogus value (by setting it to 0) and proceed, rather than just aborting.

Note that the question of whether a font would really be functional after recovery is very pertinent. Discarding a single ligature because of an invalid character code may be safe. Resetting a non-zero first table entry may be safe as well, but it might also be the case that the entire table (or the whole font for all we know) is in fact completely corrupted. The point here is that it is not the job of the library to make a decision; only to offer options.

In fact, having options may come in handy for interactive use (Common Lisp applications can be run both interactively and as standalone executables). Figure 1 illustrates this. In this example, a fake `cmr10` font has been corrupted on purpose: the character encoding string present in the file's header has been modified to contain parenthesis, which is illegal. When loading the font interactively, the user ends up in the debugger and is presented with a number of "soft" recovery options (keeping the string as-is, fixing it, discarding it), in addition to plain cancellation.

A non interactive application, on the other hand, would have the ability to automatically select an option without requiring user intervention. In production, the most likely choice is `CANCEL-LOADING` (and then fall back to another font). Given the goal we are trying to achieve here however, we will rather select the recovery option that allows us to proceed with the parsing for as long as possible.

Figure 2 summarizes all the possible problems (rectangles) and the corresponding recovery options (ellipses) that `tfm` provides. The details are not important. The intent of this picture is to convey the feeling that even for a relatively simple file format, a complete error / recovery ontology can quickly become rather intricate.

3 The `tfm-validate` Library

While `tfm` was originally a requirement for ETAP, `tfm-validate`[3] is a typical case of a project that was born out of curiosity rather than necessity, and also because it was actually quite easy to do. The key ingredient in `tfm-validate`'s design simplicity is the very peculiar exception handling that Common Lisp provides, the so-called "condition system" [6, 4], which will now describe.

3.1 The Common Lisp Condition System

Most programming languages with explicit support for exception handling use some form of "`try/catch`" mechanism, as illustrated in the left part of Figure 3. A program may establish points at which exceptions (thrown elsewhere) are caught and handled. In the example, the program throws an exception while executing `func4`. The exception travels up the call stack until it reaches the handler in `func2`. If the exception is actually caught there, execution resumes at that point. Otherwise, the exception goes one more step up, to `func1`.

[3] `github.com/didierverna/tfm-validate`

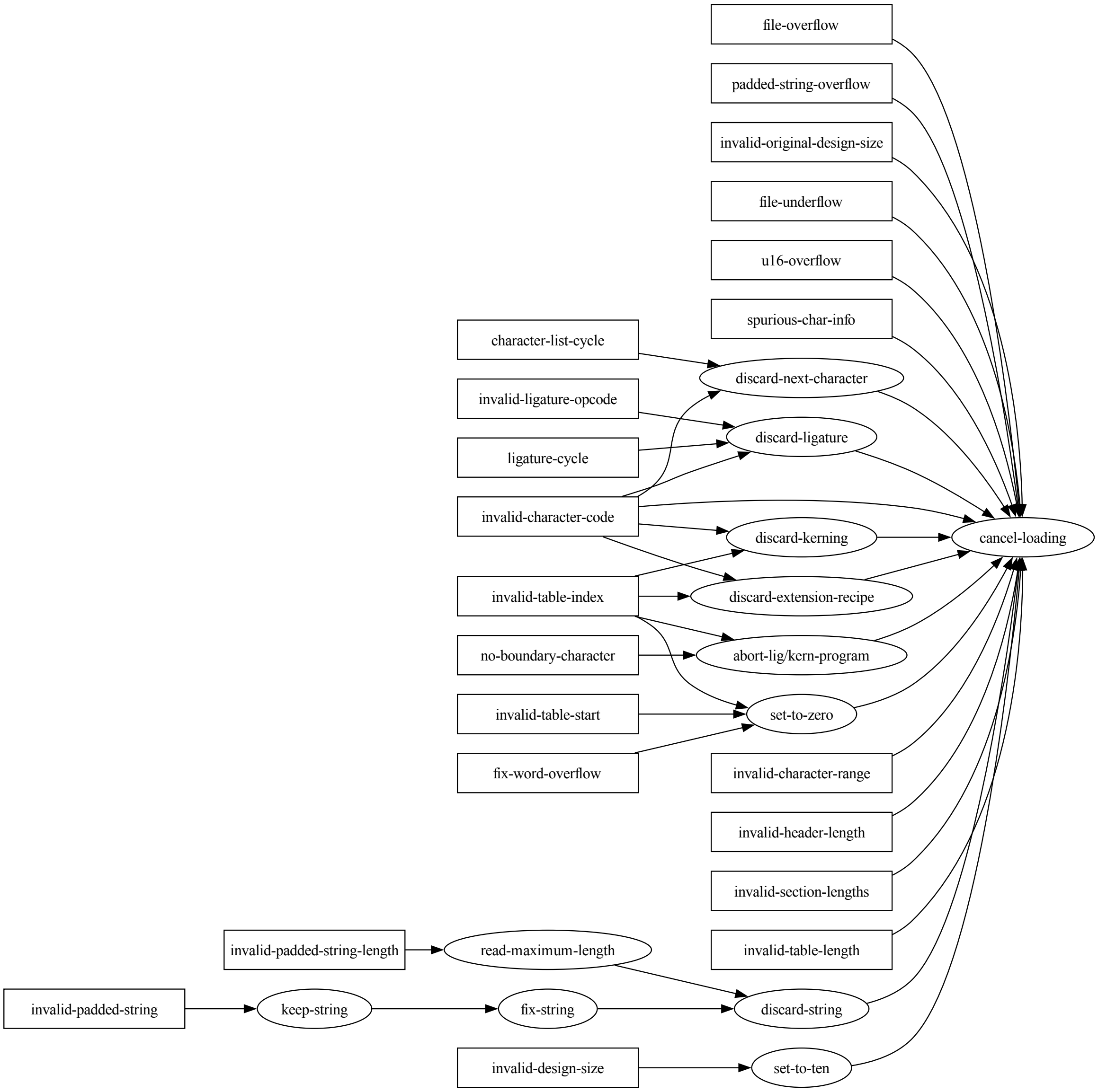

Figure 2: The `tfm` error ontology

Unfortunately, this mechanism suffers from an unnecessary limitation in expressiveness: the exception handler actually does two different things at the same time (and for no good reason). Indeed, a control point established by a handler serves not only to catch an exception, but also to resume execution. There is in fact no reason to limit ourselves to such a simple scheme, and the Common Lisp condition system adds one more degree of freedom to its exception handling infrastructure.

3.1.1 Signal / Handle / Restart

The equivalent of "throwing an exception" is called "signalling a condition" in Lisp, and the concept is equivalent. There is, however, no such thing as single catch/resume points in the Lisp condition system. Instead, a program establishes points where it is possible to resume execution (called "restarts"), and point where conditions are caught (called "handlers"). This is illustrated in the right part of Figure 3. Given the same scenario as before, `func4` signals a condition. The condition goes up the call stack and finds a handler in `func2`. If this handler is interested, it

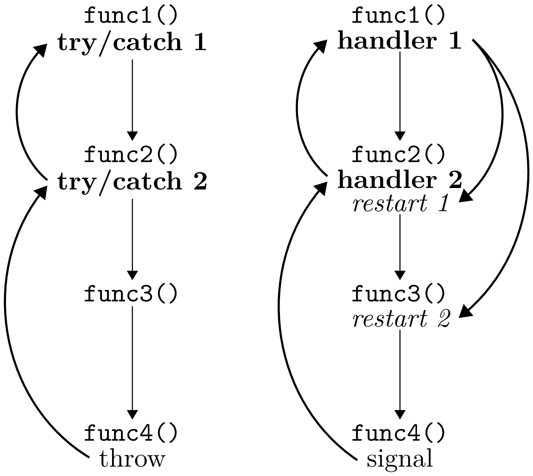

Figure 3: `try/catch` *vs.* `handle/restart`

now has two options: resume execution right here (with *restart 1*), or in `func3` with *restart 2*. Otherwise, the condition goes one more step up and the handler in `func1` is given the same two choices, since no additional restart is installed.

3.1.2 First-Class Conditions

A second important aspect of the Common Lisp condition system (not specific to Lisp this time) is that it is grounded in CLOS [5], the object-oriented layer of the language. It means that creating an ontology of errors boils down to designing a hierarchy of conditions *classes*, and the signalled conditions are reified as *objects*, that is, instances of the corresponding classes. In other words, conditions are "first-class" citizens in the language [2, 7].

3.2 The Design Simplicity of `tfm-validate`

Why is all this relevant to the design simplicity of `tfm-validate`? As mentioned before (Section 2.2), it is not the job of `tfm` to handle errors; only to detect them and offer as many soft recovery options as possible, for flexibility. In the technical terms of the Common Lisp condition system, we now understand that `tfm` *signals* conditions and provides a variety of *restarts*, but does *not* establish any handlers.

Short of handling conditions, a `tfm` user ultimately ends up in the debugger if something goes wrong (again, as demonstrated in Figure 1). But the key point is that since restarts and handlers are different concepts, it is possible to decide what to do programmatically rather than interactively, by establishing handlers *outside* `tfm`, or more specifically *around* calls to it.

We now understand why `tfm-validate` was in fact quite easy to write. The main entry point is a function called `invalidate-font`, which calls `tfm`'s `load-font` function. But before doing so, `invalidate-font` establishes a (rather large) handler for all the conditions that `tfm` may signal, and for every one of them, selects the "softest" restart available, allowing to proceed with the parsing for as long as possible. Note again that because handlers and restarts are not required to be located at the same places in the code, no modifications to the original `tfm` library are required to make it work like a compliance checker rather than for actually using fonts.

But `invalidate-font` doesn't stop there. Every time a condition is caught, the function collects it before restarting (remember that conditions are actual objects). The return value of `invalidate-font` is thus the list (possibly empty) of all the signalled conditions. In fact, `invalidate-font` doesn't do any printing by itself. After execution, the user gets the list of signalled conditions, and is then free to do whatever they wish with it, such as inspecting, printing in one form or another, or even generating a website…

4 TEX Live Validation

… which is the point we are getting to. The function `invalidate-font` which, again, is essentially a wrapper around `load-font`, collecting the signalled conditions, is 68 lines long. With 10 more lines, we offer a function checking the compliance of a whole directory tree rather than of a single font file. This function is unsurprisingly called `invalidate-directory`.

At that point, we were curious about the state of the TEX Live distribution, since it is a rather large repository of TFM files, all located under a single directory tree. As it turns out, running `invalidate-directory` on it revealed a quite large number of non-compliance issues, which was an incentive to put all that information into a human-readable shape.

4.1 Non-Compliance Reports

The `tfm-validate` library provides yet another entry point called `invalidate-texlive`. It generates a website aggregating non-compliance reports (one HTML page per culprit TFM file) plus a couple of indexes. With the help of Norbert Preining, the system is now run on a daily basis and the corresponding website is made available at `texlive.info/tfm-validate/`.

At the time of this writing, the results of the validation process are as follows. 79016 fonts are inspected. 2983 fonts are skipped because `tfm` doesn't support OFM or JFM yet. 770 fonts are found to be non-compliant, which may seem quite a lot. On the other hand, there are only 4 kinds of problems, 3 of which are considered as warnings, and a single one a truly unrecoverable error.

4.2 File Overflow

By far, the most common issue that `tfm-validate` finds is file overflows, affecting 628 fonts. The TFM standard mandates that the first two bytes of a TFM file encode the file's length. A "file overflow" warning is signalled if the actual file's length is greater than expected. Note that `tfm` knows about the special values 0, 9, and 11, denoting extended TFM files (OFM or JFM), which are not supported yet.

Of course, when the declared file size disagrees with the actual, there is no way to tell for sure which (if any) is correct. However, absent any other problem during parsing, the file containing a tail of junk is much more likely than the first two bytes (only) being corrupted, hence a warning.

A quick test on a couple of such files seems to confirm that hypothesis. We compiled a sample document with them, and it appears that not only TeX has no problem loading the fonts, the outputs look normal as well. On top of that, let us mention that `tftopl` adopts the same posture: it signals the problem but otherwise just discards the junk (Section 20 of the documentation).

Further investigation on the tails was inconclusive. In particular we couldn't figure out whether some tails contain meaningful information rather than just junk (a possible cause for file overflows could be padding to storage blocks). As a consequence, the signalled warnings do not include the tails content.

4.3 String Overflow

The situation is slightly different with the next kind of problem we encountered, namely, padded string overflows, currently affecting 74 fonts.

A TFM file may contain two optional strings in its header. The first one, 40 bytes long, identifies the character coding scheme. The second one, 20 bytes long, is the font identifier (font family name). These strings are supposed to be in BCPL format. In particular, the first byte must contain the actual length of the string.

`tfm` signals a "padded string overflow" warning when a BCPL string is not padded with zeros. Doug McKenna suggested[4] that padding a BCPL string with zeros may not have always been a requirement, as it was only added to `pltotf` in April 1983, for version 1.3, that is, two years after its initial release (Section 87 of the documentation). On the other hand, David Fuchs mentioned padding with zeros as early as in February 1981 [3].

Anyway, the decision as to whether a padded string overflow should be a warning or an error is even simpler to make than in the case of a file overflow. Those strings are purely informative, they have no impact on the font's usability, so it does not hurt to continue loading the font.

Besides, the padding area seems to have been intentionally abused in the majority of the cases: a lot of fonts contain `"Y&Y Inc"` in there, making their origin quite clear. Because of that (and contrary to file overflows), the content of the padding area is included in the warnings.

4.4 Spurious Char Info

The next problem we encountered (also a warning, affecting 66 files) is a more obscure matter. TFM files have a so-called "char info table" providing the actual character metrics of the font. The table contains 4-bytes entries for the full range of characters from the minimum character code (*bc*) to the maximum one (*ec*). However, a font may also have "holes" in this range, that is, undefined characters for some codes between *bc* and *ec*.

Undefined characters must have a width of 0, materialized by a widths table index of 0 as well. The spurious char info warning indicates that an entry for an non-existent character is not completely zero'ed out. In the problematic char info entries that we found, the third byte usually has a value of 1 (indicating an index into a ligature or kerning program), and sometimes a non-zero fourth byte (the actual index).

A possible explanation would have been the existence of a so-called "boundary character" (also an obscure matter in TFM) which is not required to exist for real in the font, but upon inspection of several problematic ones, this appears not to be the case.

Note that `tftopl` completely ignores characters with a width index of 0 (Section 78 of the documentation), and `pltotf` zero'es out non-existent characters (Section 74 of the documentation). All the more reasons to not consider this problem a showstopper.

4.5 Fix Word Overflow

Finally, this one is the only true error we encountered, and it only affects two fonts: `ArevSans-Bold`, and `ArevSans-BoldOblique`. TFM has a notion of "fix word" numerical values which (with two exceptions) must remain within `]-16,+16[`. In particular, the actual font metrics (width, height, depth, and italic correction) are expressed in fix words.

[4] reference lost; could have been in a thread on `texhax`…

In the two aforementioned files, exactly 124 such values are off the charts. Again, for the sake of flexibility, `tfm` offers a soft recovery option for this problem (see Figure 2): setting the culprit value to 0, which would most likely result in an unreadable document. TEX refuses to load these fonts, which confirms the severity of the problem; hence an error.

5 Related Work

Manuel Pégourié-Gonnard wrote a Perl script[5] for checking the validity of a variety of files using external programs (typically, `tftopl` for TFM files). It is our understanding that this script produces a somewhat terse output: it prints a list of "bad" files without collecting more specific information, let alone presenting it in a human readable form.

According to a comment by Karl Berry, the script took a long time to run and the maintenance was tedious, so using it was abandoned in August 2019.

6 Conclusion and perspectives

As mentioned before, this project was born out of curiosity rather than necessity, and because it was easy to develop. Whether it is actually useful remains to be seen. Perhaps having compliance problems publicly advertised on a website will be a new kind of incentive for authors to update their files, and perhaps this project will be more helpful to watch over new additions rather than blame older content.

One merit of this project is to provide an insight into the global status of TFM compliance over a large set of fonts. In particular, we can see that the surprising number of non-compliant files is mitigated by the fact that most issues are in fact benign (only two fonts were found to be truly unusable).

In the future, we plan on adding new font formats to the system. Provided that we can find the appropriate documentation, OFM and JFM are likely to be straightforward additions and as a matter of fact, the `tfm` library is already prepared for it. We have also started to work on an OTF parser, designed along the same lines (that is, built around the Common Lisp condition system) but this will take slightly longer to complete.

Finally, the current layout of the website still has a lot of room for improvements. It currently provides two indexes, but the general question boils down to offering different forms of access to cross-referenced information. Karl Berry has already suggested a couple of possible ways to do so, which we will definitely take into account in the future.

[5] `tug.org/svn/texlive/trunk/Master/tlpkg/bin/tl-check-files-by-format`

Acknowledgements

The author wishes to thank Norbert Preining, Karl Berry, and Doug McKenna for fruitful exchanges during the development of both `tfm` and `tfm-validate`.

References

[1] ANSI. American National Standard: Programming Language — Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[2] R. Burstall. Christopher Strachey — Understanding programming languages. *Higher Order Symbolic Computation*, 13(1–2):51–55, 2000.

[3] D. Fuchs. TEX font metric files. *TUGboat*, 2(1):12–16, Feb. 1981.

[4] M. Herda. *The Common Lisp Condition System*. Apress, 2020. `doi.org/10.1007/978-1-4842-6134-7`

[5] S.E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[6] P. Seibel. *Practical Common Lisp*. Apress, Berkeley, CA, USA, 2005. Online version at `gigamonkeys.com/book/`.

[7] J. Stoy, C. Strachey. OS6 — An experimental operating system for a small computer. Part 2: Input/output and filing system. *The Computer Journal*, 15(3):195–203, 1972.

[8] D. Verna. ETAP: Experimental typesetting algorithms platform. In *15th European Lisp Symposium*, pp. 48–52, Porto, Portugal, Mar. 2022. `doi.org/10.5281/zenodo.6334248`

[9] D. Verna. Interactive and real-time typesetting for demonstration and experimentation: ETAP. In *TUGboat*, B. Beeton, K. Berry, eds., vol. 44, pp. 242–248. TEX Users Group, TEX Users Group, Sept. 2023. `doi.org/10.47397/tb/44-2/tb137verna-realtime`

◇ Didier Verna
EPITA Research Lab
14–16, rue Voltaire
94270 Le Kremlin-Bicêtre
France
`didier (at) lrde.epita.fr`
`https://www.lrde.epita.fr/~didier/`
ORCID 0000-0002-6315-052X