Extension language integration of LuaTeX and LilyPond

David Kastrup

Abstract

LuaTEX uses Lua as its extension language while the music typesetter LilyPond employs the Scheme dialect GUILE for that purpose.

It is interesting to see how those extension languages are integrated into the "core" language and what interfaces are used for passing information back and forth between user, principal language, and extension languagem and to what degree the languages interact to form a coherent experience or one more modelled along the line of "I'd rather like to discuss this with your brain surgeon".

> It will [hold]. I couldn't put a mere mending charm on the Ring of Erreth-Akbe, like a village witch mending a kettle. I had to use a Patterning, and make it whole. It is whole now as if it had never been broken.
> – Ursula K Le Guin, "The Tombs of Atuan"

As an initial remark: *using* LilyPond does not require dealing with the complex mechanisms shown here. Those mechanisms exist exactly to save the user from requiring arcane knowledge before being able to do serious work.

The music typesetting system LilyPond's low-level implementation language is C++. The Scheme interpreter GUILE, designated as the GNU project's extension language, was integrated into LilyPond at an early stage.

While LilyPond as a music description language and its extensibility through GUILE have moved forward together, the underlying internals and concepts, data structures and algorithms are still getting streamlined.

It remains a challenge to straighten out the system architecture and unify programming and user features to a degree matching even the most naïve expectations characteristic for beginners.

While web searches, mailing lists and other social media can help with independently encountered problems[1], one should not exhaust the goodwill of users with actual or perceived complexity and inconsistencies.

1 LilyPond and Scheme

1.1 A sketch of Scheme

Scheme, a language in the Lisp family, is marked by its absence of structuring interpunctation. With few exceptions, the only structuring feature is a pair of matching parentheses enclosing a list. Since every layer of nesting is a list of its own, keeping track of open parentheses and choosing an appropriate indentation is a task frequently delegated to a list-aware editor.

Executing a program involves the 'evaluation' of lists, and each list corresponds to a function (or macro) call, with the first list member specifying the function, and the other list members specifying function argument expressions.

Here are some example expressions and the corresponding results:

```
                (+ 2 3)   ↦   5
(if (< 2 3) (+ 5 7) 8)   ↦   12
```

In contexts where lists would be evaluated, the evaluation can be inhibited by using one of several quoting mechanims, the simplest being straight quotes:

```
'(+ 1 2) ↦ (+ 1 2)
```

Since lists are both the program representation of Scheme as well as its most important data structure, manipulating programs is easy, and Scheme has a powerful and expressive macro system. At the same time, this lack of typographical structure provided by other interpunctions than parentheses levels is usually considered the least endearing characteristic of languages in the LISP family. Indentation conventions help with finding one's way around other people's Scheme programs.

1.2 Transitioning between the worlds

LilyPond's input language focuses on musical constructs: note pitches, note durations, chords, articulations, melodies, and so on. Scheme as a programming language is more concerned with entities like functions, numbers, strings. In order to restrain the impetus for duplication of functionality over the respective main reigns of the languages, the transitioning between both should be as smooth as possible:

[1] "'Computers are useless. They can only provide answers."' – Pablo Picasso

not as much like switching between German and English in a poem, but rather like between verbs and nouns in a sentence.

The principal tools for transitioning in LilyPond are two constructs: `#` within LilyPond mode will use Scheme to read and evaluate the immediately following Scheme expression; `#{` inside of Scheme will switch into LilyPond mode until reaching the matching `#}` code. Either construct will *return* the resulting value for use in the respective other language.

Quite importantly, reading/scanning of embedded code is done completely under the control of the respective other language interpreter, so the foreign input is free from string quoting characters and similar contraptions.

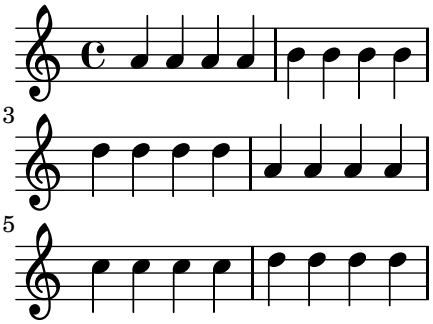
Now let us jump right into the deep end and delve into a brief example intertwining those constructs with Scheme macros.

1.3 Exposition

The following example provides and demonstrates some useful facility, making use of several interactions between LilyPond/GUILE.

```
#(define-macro (pattern args result)
 `(define-music-function
           (parser location ,@args)
    ,(make-list (length args) 'ly:music?)
    #{ $@(list ,@result) #}))

$(pattern (A B C D) (A B D A C D))
  { a' a' a' a' }
  { b' b' b' b' }
  { c'' c'' c'' c'' }
  { d'' d'' d'' d'' }
```



Now apart from having a reasonably nice effect (`pattern` takes two lists of symbols, the first specifying the input order of its arguments, and the second specifying the desired output order), the code is both short as well as sophisticated. We will now first analyze the logic and elements of the code, and then take a look at how LilyPond manages to make it work.

1.4 The storyline

Scheme macros are a variation of Scheme functions that changes execution order. When a Scheme function is called, first its arguments get evaluated, and then the function is executed with the evaluated arguments, and its return value is the result of the function call.

A macro changes the order: the macro is called *first* on the unevaluated arguments (in the case of the call of `pattern`, the first argument is a list with the symbols `A`, `B`, `C` and `D`). The result of the macro call is *then* evaluated in the Scheme evaluator. The result of this evaluation *after* executing the macro becomes the result of the macro call.

Since unevaluated Scheme code usually consists of nested lists, it is often convenient to specify the macro body in the form of *almost* entirely quoted lists (quotes keep the list from premature evaluation), but with a few variable elements spliced in.

Quasiquotes are the tool of choice for working with mostly constant lists with a few variable elements, and are seen in connection with most macros. The whole quasiquote is introduced with a backward quote character "`‘`" instead of the normal one "`’`". They are called "quasiquotes" because inside of such a quoted expression, you can 'unquote' parts by preceding them with a comma "`,`", and you can 'unquote-splice' a list expression into a surrounding list by preceding them with comma-at "`,@`".

So, for example,

```
`((+ 1 2) ,(+ 1 2) ,@'(+ 1 2))
```

evaluates to the list

```
((+ 1 2) 3 + 1 2)
```

The starting list `(+ 1 2)` containing the symbol "`+`" and the numbers `1` and `2` is retained unchanged. The same construct with an unquote before it is evaluated to `3` before being put in the list, and the quoted list `'(+ 1 2)` evaluates by removing the quote, to `(+ 1 2)` which is then spliced into the surrounding list, effectively removing one level of parentheses.

The call to `pattern` Now the call to pattern looks like

```
(pattern (A B C D) (A B D A C D))
```

and, considering that macro arguments are not evaluated, the result of substituting the arguments is simply

```
`(define-music-function
        (parser location ,@'(A B C D))
   ,(make-list (length '(A B C D)) 'ly:music?)
   #{ $@(list ,@'(A B D A C D)) #})
```

(we have inserted quotes appropriately to indicate that the lists are not to be executed, so that one can feed this into a Scheme sandbox inside of LilyPond) and if we now execute the quasiquote, we are

left with the following expression when finishing executing the macro:

```
(define-music-function (parser location A B C D)
   (ly:music? ly:music? ly:music? ly:music?)
   #{ $@(list A B D A C D) #})
```

This is the definition of a music function doing the required transformation of four music arguments to the requested order.

LilyPond's "$@" operator is a variation on the Scheme splicing operator and has been introduced in LilyPond 2.15.41: it can be used for splicing a list of expressions into the surrounding #{...#} construct, and so

```
#{ $@(list A B D A C D) #}
```

is essentially the same as

```
#{ $A $B $D $A $C $D #}
```

To achieve the same effect in older LilyPond versions, we would have had to write

```
(make-sequential-music
  (map ly:music-deep-copy (list ,@result)))
```

instead of

```
#{ $@(list ,@result) #}
```

since one effect of "$" is to create a copy. Whenever a music expression may be used more than once, we need to copy it since many functions processing music change their input while processing it. So the new splicing operator saves us from knowing `make-sequential-music` and some other things.

1.5 Behind the scenes

Why does this even work? This question is indeed puzzling since macro expansion is a complex beast, and #{...#}, embedded LilyPond, is actually a piece of LilyPond code that is stored away in a string and later executed in a copy of the LilyPond parser. So how is it able to access the original variable `result`, a parameter of the function call? The LilyPond manual states that Scheme expressions inside of embedded LilyPond are executed in "lexical closure". However, macro expansion happens at an earlier point of time, before closures are even being formed. So how can this work out?

At the first layer, let us ask the Scheme sandbox (slightly reformatted):

```
lilypond scheme-sandbox
GNU LilyPond 2.15.42
Processing `[...]/scheme-sandbox.ly'
Parsing...
guile> (define-macro (pattern args result)
  `(define-music-function (parser location ,@args)
     ,(make-list (length args) 'ly:music?)
     #{ $@(list ,@result) #}))
guile> (macroexpand-1 '(pattern (A B C D)
                                (A B D A C D)))
(define-music-function (parser location A B C D)
    (ly:music? ly:music? ly:music? ly:music?)
  (#<procedure embedded-lilypond
     (parser lily-string filename line closures)>
    parser
    " $@(list ,@result) "
    #f 3
    (list (cons 2 (lambda () (list A B D A C D))))))
guile>
```

After expanding the macro, we arrive at a call to the internal function `embedded-lilypond` which gets, as expected, the contents of #{...#} as a string. But it also gets "`closures`" a list containing pairs, with the first element being an offset into the string, and the second being an anonymous "`lambda`" function for evaluating the embedded Scheme expression at that point in the string.

LilyPond's parser uses this anonymous function that has been created in the lexical environment (namely its placement in Scheme source code) of the #{...#} expression instead of actually interpreting the Scheme expression from the text at the time it passes the whole embedded LilyPond string to the parser. And it turns out that this `lambda` function magically already can deliver the right expression since it has been formed within the context of the music function and retained as a closure. So even while the full LilyPond expression is only stored as a string and interpreted later, the scraps of Scheme code inside have been macro-expanded and saved in a function.

The code reponsible for that is buried in the innards of `scm/parser-ly-from-scheme.scm` in the LilyPond source, turning #{...#} into the above expression right inside the Scheme reader: the original construct enters Scheme only in this preprocessed version.

Let us dig deeper. Let us take a look at the embedded Scheme expression itself, outside of the macro and quasiquote and unevaluated:

```
guile> '#{ $@(list ,@result) #}
(#<procedure embedded-lilypond
  (parser lily-string filename line closures)>
  parser
  " $@(list ,@result) "
  #f 6
  (list (cons 2
        (lambda ()
          (list (unquote-splicing result))))))
```

Now "`(unquote-splicing result)`" is just a different representation for "`,@result`", like we are now seeing a different representation for the whole #{...#} construct. As part of a larger quasiquote construct, the "`unquote-splicing`" operator will kick in and substitute the value of "`result`" into the list, and that is just what we are seeing here.

And the reason that `unquote-splicing` is actually found by the quasiquote operator is that in this stage of evaluation (or rather non-evaluation), the anonymous `lambda` function is not yet a function, but just a list where the first element is the symbol "`lambda`". So the macro can do its work on the raw form of the expression, and after the `unquote-splicing` has done its work, normal evaluation happens and turns the result into an anonymous function that is later called from the LilyPond parser when it encounters `$@` and skips the following Scheme expression, taking its value from the call of the `lambda` function instead.

Deep breath At the end of this journey, we have seen how a reasonably concocted broth made from shoestring and putty is able to intertwine the LilyPond language parser with an integrated Scheme layer in a manner that provides lexical closure, nesting, back-and-forth passing of values, macros and functions working across the language boundaries.

It is worth noting that input location is propagated carefully across those layers so that error messages for erroneous input point to the actual place of the problem, again making for a unified end user experience even in the event of user error.

2 Lua as LuaTEX's extension language

LuaTEX's way of integrating Lua has to be described in a more technical manner since the available low-level mechanisms are not sufficient for providing the kind of refined and natural interface that LilyPond offers for GUILE.

TEX actually has its own macro processing system called "mouth". For use inside of the mouth, LuaTEX provides the `\directlua` command. The contents of this command are passed as text into the Lua interpreter. There is no parameter passing mechanism short of splicing additional text into the interpreter. There is no lexical closure mechanism provided. Lua has access to internal TEX variables, but TEX's mouth can't set those variables. `\directlua` may print input into TEX's mouth via `tex.print`, however.

TEX's main processing layer is called its "stomach". There is no actual interface from LuaTEX's stomach to Lua. One can, however, use `\directlua` inside of a `\protected\def` to trigger expansion at the latest possible point of time, synchronized to the stomach's operation. Printing back into TEX's mouth via `tex.print` is still possible in this mode of operation. Setting variables inside of TEX at their current nesting level is possible, but there are no programmatic ways of making decisions based on such variables.

At a post-stomach level (the TEX taxonomy prudently does not assign a name) there is a primitive called `\latelua` that is triggered inside of the execution of the `\shipout` primitive. Printing back into TEX's mouth at this stage for any purpose other than immediate shipment is not commendable.

LuaTEX and Lua don't speak a common language, and their communication is complicated by each language having its own interpretation layers that cannot easily or naturally be switched off. A simple LuaTEX example would be

```
\directlua{tex.print("\noexpand\\message{Hi}")}
```

The use of `\noexpand` may seem artificial: however, in 15 minutes of experiments I have not been able to find a combination with `\unexpanded` that would work on the whole string without expanding `\\`, but probably a solution with `\detokenize` could be made to work even though its exact operation depends on the current catcode regime.

In a nutshell, the default primitives of LuaTEX do not constitute a useful or friendly programming interface. While the same can be said for some standard TEX primitives, at least their situation is ameliorated by the presence of the somewhat ubiquitous thin wrapper of plain TEX forming the basis for the "TEXbook", the principal documentation of TEX. The raw iniTEX has no user-level documentation, and it was probably a good choice to let the "TEXbook" provide a view how the available primitives can (and should) be tied together in a coherent user experience.

Crossing between TEX and Lua requires a lot of knowledge of unrelated material and is, particularly in the Lua to TEX direction, hardly better than using pipes.

Now while differentiating between "primitive" and "definition" in LilyPond does not make all that much sense, one can at least look at the core C++ functions that the user interface is built on. In this case, we have

ly:parse-string-expression parser-smob ly-code filename line

> Parse the string *ly-code* with *parser-smob*. Return the contained music expression. *filename* and *line* are optional source indicators.

ly:parser-clone parser-smob closures

> Return a clone of *parser-smob*. An association list of port positions to closures can be specified in *closures* in order to have '`$`' and '`#`' interpreted in their original lexical environment.

Glossing over the techno-babble, it is nevertheless clear that even the 'primitives' are conveying the information needed for providing a smooth integration and unified experience of the different language layers in a natural manner as parameters and return values.

One can call LilyPond from Scheme and vice versa, in a functional and straightforward manner.

3 Summary

LuaTEX is not integrated into TEX in a manner that would make for a unified user experience, with common information flow and concepts. While some of this situation is a consequence of its history, LilyPond and its extension language GUILE are separate projects as well.

Like LuaTEX, LilyPond has to deal with the issue of independently executing input language interpreters. The solutions, while not unbreakable, unify the language layers to a degree where delayed evaluation, macro processing, value passing and even lexical closures operate in the expected manner. In that manner, the increasing exposure of advanced users to Scheme is rarely accompanied by jarring experiences of exploding internals like when dealing with LuaTEX.

While the typical user experience of LuaTEX can likely be improved quite a bit even based on the existing primitives and might possibly already *be* so in the Context format heavily relying on LuaTEX, this information does not escape into the wild.

If LuaTEX development focused more on the means of creating and promoting a unified user experience, its perception as an optional tinker box requiring considerate knowledge and thus being suited at best for package writers rather than end users might change.

In particular, it would provide a compelling argument for preferring LuaTEX over other engines that can, with considerable effort, be made to solve pretty much any computing task with the help of non-user accessible programming.

If the integration of Lua would provide actual end user features, end users *will* care about the engine they use. At the current point of time, LuaTEX mostly pitches itself to users already proficient in TEX programming and able to deal with the complications arising when including Lua fragments. The basic question users ask themselves nowadays is "Do I need LuaTEX?" when the real question should be "Do I want LuaTEX?". To make this case, it is more important for the end user to see what *he* can do better rather than what highly proficient users can do better using LuaTEX.

4 PostScriptum

Travelling back from the conference made me think of some nicer ways of working with Lua inside of TEX than currently possible. Let's try defining the TEX interface of the primitive `\parshape` in Lua.

```
\protected\luadef parshape (tex)
  n = tex:get_number ()
  arr = {}
  for i=1,n do
    arr[i] = {}
    arr[i][1] = tex:get_dimen ()
    arr[i][2] = tex:get_dimen ()
  end
  tex.parshape = arr
end

\luadef parshape.number (tex)
  return #tex.parshape
end
```

This is just a very rough sketch. `\parshape` in the stomach (`\protected` keeps it from expanding in the mouth) reads a number followed by dimension pairs and stores all that in the parshape. The function parameter `tex` is not the global `tex` array but rather a local version derived from it via metatables. It likely captures the current grouping as well as the parse state and has additional function calls available.

`\parshape` can be used in number contexts, returning the current parshape size.

`\luadef` should switch off TEX's reader, reading lines into the Lua interpreter until a function definition is completed. Since it should change interpretation of input characters (like `%` does) even inside of macros, a special character/catcode might be used. Note that after `\let\hash=#`, this character can be used with its special meaning even when reading in macros, so having both or just `\luadef` seems feasible.

Implementation details may turn out quite different than sketched here, but the key point is that Lua right now does not integrate into TEX's user interfaces to any reasonable degree, and there is a lot of potential for it doing better than that.