Using HsLua

Examplified by Pandoc's filtering system

Albert Krewinkel

Berlin HUG, 2017-09-20

HsLua

What's HsLua?

HsLua

- bridges Haskell and Lua;
- allows to include a scripting language in any program;
- has a small footprint;
- is (reasonably) safe; *and*
- abstracts over multiple Lua versions.

Why would one use it?

Use-cases include:

- flexible, Turing-complete configuration language;
- making a program scriptable;
- exposure of internals to non-Haskell users.

Pandoc

The universal document converter

From Docbook, Docx, EPUB, Haddock, HTML, JSON, LaTeX, Markdown (flavors: CommonMark, GitHub, MultiMarkdown, PHP Markdown Extra, strict), MediaWiki, Muse, native, ODT, OPML, Org, RST, t2t, Textile, TikiWiki, TWiki, and VimWiki.

To Asciidoc, Beamer, CommonMark, Context, Docbook4, Docbook5, DOCX, DokuWiki, DZSlides, EPUB2, EPUB3, FB2, Haddock, HTML4, HTML5, ICML, JATS, JSON, LaTeX, groff (man and ms), Markdown (all flavors mentioned as above), MediaWiki, Muse, native, ODT, OPML, Org, plain, RevealJS, RST, RTF, S5, Slideous, slidy, TEI, Texinfo, Textile, and ZimWiki.

Via Internal document model.

Document AST: Inlines

```
data Inline
    = Str String
    | Space | SoftBreak | LineBreak
    | Emph [Inline]
    | Strong [Inline]
    | Cite [Citation]  [Inline]
    | Code Attr String
    | Math MathType String
    | RawInline Format String
    | Span Attr [Inline]
    …
    deriving (Show, Eq, Ord, Read, Typeable, Data, Generic)
```

Document AST: Blocks

```
data Block
    = Plain [Inline]
    | Para [Inline]
    | CodeBlock Attr String
    | RawBlock Format String
    | BlockQuote [Block]
    | OrderedList ListAttributes [[Block]]
    | BulletList [[Block]]
    | Header Int Attr [Inline]
    | Div Attr [Block]
    …
    deriving (Eq, Ord, Read, Show, Typeable, Data, Generic)
```

Custom writers

Write a custom output format

- Parse input into internal representation.
- Use Lua to convert elements to strings.
- Write resulting string.

```
-- Example functions handling inline elements.
-- Take and return a string.
function Strong(s)
  return "<strong>" .. s .. "</strong>"
end
function Str(s)
  return html_escape(s)
end
```

Custom writer implementation

```haskell
instance {-# OVERLAPS #-} ToLuaStack [Inline] where
  ils = push =<< inlineListToCustom ils

-- | Convert list of Pandoc inline elements to Custom.
inlineListToCustom :: [Inline] -> Lua String
inlineListToCustom lst = do
  xs <- mapM inlineToCustom lst
  return $ mconcat xs

-- | Convert Pandoc inline element to Custom.
inlineToCustom :: Inline -> Lua String
inlineToCustom (Strong lst) = callFunc "Strong" lst
inlineToCustom (Str str) = callFunc "Str" str
```

Example 1: Invoke dot on some code blocks

```
function CodeBlock(s, attr)
  -- If code block has class 'dot', pipe the contents
  -- through dot and base64, and include the
  -- base64-encoded png as a data: URL.
  if attr.class and
     string.match(' ' .. attr.class .. ' ',' dot ') then
    local png = pipe("base64", pipe("dot -Tpng", s))
    return '<img src="' .. png .. '"/>'
  -- otherwise treat as code
  else
    return "<pre><code" .. attributes(attr) .. ">"
           .. escape(s) .. "</code></pre>"
  end
end
```

Example 2: panlunatic

- Panlunatic is a custom writer which outputs JSON.
- Produced data can be read back into Pandoc.

```
pandoc -t custom.lua input.md | \
pandoc -f json output.epub
```

- Manipulations of the document AST are possible.
- Example: making image paths relative.

```
panlunatic = require("panlunatic")
setmetatable(_G, {__index = panlunatic})
function Image(s, src, tit, attr)
  local relSrc = src:gsub("^/", "")
  return panlunatic.Image(s, relSrc, tit, attr)
end
```

Pros and cons

Advantages:

- Portable
- Powerful

Disadvantages:

- Slow
- Awkward
- Requires extra software (dhjson, panlunatic)

Lua filters

Native lua filters

```
-- Parse raw blocks containing markdown into
-- a pandoc block element.
funciton RawBlock(elem)
  if elem.format == "markdown" then
    local pd = pandoc.read(elem.text, "markdown")
    return pd.blocks[1]
  end
end
```

Getting elements from Lua

```haskell
-- | Return the value at the given index as block
peekBlock :: StackIndex -> Lua Block
peekBlock idx = do
  tag <- getTag idx
  case tag of
      "BulletList"  -> BulletList <$> elementContent
      "Para"        -> Para <$> elementContent
      "OrderedList" -> uncurry OrderedList <$> elementContent
      "RawBlock"    -> uncurry RawBlock <$> elementContent
 where
   -- Get the contents of an AST element.
   elementContent :: FromLuaStack a => Lua a
   elementContent = getTable idx "c"
```

Pushing structured data to Lua

```
-- | Push a block element to the top of the lua stack.
pushBlock :: Block -> Lua ()
pushBlock (Para blcks) = pushViaCall "pandoc.Para" blcks
pushBlock (RawBlock f cs) = pushViaCall "pandoc.RawBlock" f cs
```

Lua:

```
pandoc.Para = function(content)
  return {c = content, t = "Para"}
end
pandoc.RawBlock = function(format, text)
  return {c = {format, text}, t = "RawBlock"}
end
```

Usefulness

Custom readers and filters make pandoc very versatile.

- Gieben, R. “Writing I-Ds and RFCs Using Pandoc and a Bit of XML.” (RFC 7328, 2014).
- Krewinkel A, Winkler R. (2017) Formatting Open Science: agilely creating multiple document formats for academic manuscripts with Pandoc Scholar. *PeerJ Computer Science* 3:e112

Wrapping up

Summary

- HsLua makes Lua usable with Haskell.
- Lua is great to make your program extensible.
- Use Pandoc for all your document conversion needs.

Further reading

- HsLua: https://github.com/osa1/hslua
- Pandoc: https://github.com/jgm/pandoc
- Pandoc types: https://github.com/jgm/pandoc-types

Appendix

pushViaCall

```haskell
pushViaCall :: PushViaCall a => String -> a
pushViaCall fn = pushViaCall' fn (return ()) 0

class PushViaCall a where
  pushViaCall' :: String -> Lua () -> NumArgs -> a
instance PushViaCall (Lua ()) where
  pushViaCall' fn pushArgs numArgs = do
    getglobal' fn
    pushArgs
    call numArgs 1
instance (ToLuaStack a, PushViaCall b) =>
         PushViaCall (a -> b) where
  pushViaCall' fn pushArgs num x =
    pushViaCall' fn (pushArgs *> push x) (num + 1)
```