WiQEE : A Wiki for mathematical proofs

Marc Coiffier

Contents

Literate Programming in Wiki Form 1
A Concatenative Scripting Language 2
Proofs With Prismatic Constructions 3
Contributing to WiQEE, or creating your own 3
Example pages 4

The Quod Erat Edificandum Wiki – WiQEE for short – is a mix of two ideas. First of all, it is a documentation platform, whose main content is the articles written by its contributors. Secondly, it aims to become a formally-verified library of mathematical proofs and programs, that can be extracted to executable form, or composed together to prove more complicated theories.

Every page on this wiki is generated from a source file, written in a language called CaPriCon. The source of a page can be downloaded from its menu. If you want to know how the sausage gets made, I invite you to take a look at this index's source for an example of such source files.

If you do so, you may notice that the source doesn't look that different from the article itself. That is because CaPriCon is an exclusively literate programming language, in which most text is simply text, and the code is specially identified through dedicated syntaxes. The comment formatting language itself is Markdown in this case, although it would be trivial to adapt it to other textual markup languages such as TeX.

Literate Programming in Wiki Form

Everything you've read until now is a comment of the "index.md" program. If you want to write programs, though, you will probably need to write some *code*, which in this case consists in a sequence of *steps* that are executed in the order in which they appear. The first sort of code that you can execute is a code paragraph, and is indicated in the source by starting a line with `>`.

```
"<p>Such code is executed, and its output is added to the document.</p>" printf
```

Such code is executed, and its output is added to the document.

You can also write some inline code between "mustaches" (a mustache is a pair of nested `{}` inside the source), which will be executed as part of a paragraph, and whose output *will look like this* . If all works well, you should see a small image of steps next to the generated output. Clicking on those steps will, as in the previous case and without surprises, show you the steps that were taken to produce this output.

A Concatenative Scripting Language

If you peeked at the examples given before, you may have noticed something strange. In order to produce some output, you'd expect a sentence to look like `print "..."`, but instead the program was written in the reverse order, as `"..." print`.

Indeed, in the tradition of PostScript, CaPriCon is a concatenative, stack-based language. In such a language, programs are represented as sequences of symbols, that are executed from first to last, and that may keep intermediate results on a shared stack. All the power of the language comes from the words that are made available, rather than from the particular syntax of those words.

Concatenative languages have a few interesting properties that make them perfect candidates for integration into a literate environment like the one CaPriCon provides :

- *simplicity :* every program is only a sequence of words that are evaluated in order, nothing more. You don't have to learn any fancy syntax, only the words and what they mean.

  The CaPriCon language only has ~50 basic words in its vocabulary, which are referenced in this lexicon. Beyond that, you can define your own words by composing the existing ones into so-called "quotes", written `{ word1..wordn }`.

- *exploration :* concatenative languages are very well-suited to implement incremental development environments, in which you can devise programs (and proofs) by walking through their execution step by step, with complete control and visibility over the state of the environment at every stage.

  To experience this process of exploration, you can install the interactive CaPriCon interpreter (for Linux or OSX), that is used to power this site. This interpreter can be run as-is, but it is usually preferrable to load a *prelude* before any useful interaction can occur.

  A prelude is simply a file containing the basic definitions needed to write simple programs. Without one, you're left with the basic functions, which require you to be much more verbose. The default CaPriCon prelude can be found in the above packages, under the name "prelude".

If you still aren't quite sure how to go about writing your own scripts, this tutorial may help walk you through the first hurdles.

Proofs With Prismatic Constructions

While CaPriCon can be a decent addition to a documentation format, it was first designed as an interactive interface for the construction of proof terms, described in the Calculus of Prismatic Constructions (hence the name CaPriCon), which is an extension of the basic CoC with a lifting operator for naturally inductive values.

That operator is called μ, and it works by projecting inductive values onto a larger version of themselves, universe-wise. For example, given the "inductive" type $Id := \forall T : Set_n, T \to T$, and a hypothesis $id : Id$, $\mu(id)$ represents a computationally sound proof of $\forall(T^* : Id \to Set_{n+1}), T^*\ (\lambda(T : Set_n)(t : T), t) \to T^*\ id$. Each branch of the base inductive type is "projected" through μ to a parallel branch in a larger universe, with additional "rays" to provide reflection into that universe. μ thus acts like a sort of *prism* (in fact, an *endoprism*) that continuously projects inductive values onto their induction principle, giving rise to the eponymous *prismatic constructions*.

This generalization can be applied to model many common cases which previously required the introduction of abstract-only inductive types, including but not limited to : general dependent inductive types, complete with recursive and inductive-inductive types; co-inductive types and their associated semantics; and even some kinds of quotient types, though the full scope of those quotient types is still to be discovered.

Talk is cheap, show me the code From the previous overview, μ may seem quite complex, but it is actually implemented as an almost linear operation on De Bruijn indices. You can look at CaPriCon's Haskell implementation of this model to convince yourself of that. All in all, the whole interpreter/typechecker/pattern-matcher boils down to ~300 lines of index-shifting code.

If you feel hesitant about trusting a bunch of Haskell functions written by a stranger on the Internet, don't hesitate to start hijacking bits of it to write CaPriCon interepreters of your own, in your language of choice. Mine just happens to be Haskell, but maybe yours will bring something new and interesting to the table as well.

Contributing to WiQEE, or creating your own

It goes without saying that I encourage you to contribute to WiQEE in any way you'd like. You may find the project template for this site in this Git repository, and the source for the actual pages in this other Git repository.

If you want to contribute to this specific site, you can use the power of Pull Requests on the WiQEE-pages repository (the second one). I can't promise an immediate reply, but I'm working on automating the builds to allow every contributor to participate, without needing my validation at every turn.

If like me you lack that sort of patience, you can also test your pages by running your own version of this site (how exciting !). To do so, you have to clone the first repository, apply your modifications in the `src/pages` subdirectory, and run `make` over the whole thing. You'll need Pandoc and Sass to be installed, for Markdown and SCSS processing, as well as CaPriCon for obvious reasons. Once `make` has run its course, you should find a workable HTML version of your WiQEE in the `public` subdirectory.

Example pages

The basics :

- Booleans
- Equalities
- Natural numbers
- Lists
- Relations