First steps with CaPriCon

Marc Coiffier

Contents

This page is intended as a tutorial on the use of a stack-based proof environment like the one provided on this site. Since we're going to need to print things out, and we don't yet have the knowledge to write such features ourselves, let's first import a few useful functions from a preexisting module.

```
'utils require import
```

- Required module: utils

Among other things, this module defines one function that will be of interest to us : `vis`. When called, this function simply prints out every hypothesis in context and every value currently on the stack. It's very useful as the final word in a sentence, to show the resulting context. It doesn't change anything, so feel free to sprinkle it at any point of your scripts for debugging purposes.

Now, in order to understand what a stack-based language is, we first have to understand the basic concept of a *stack*, and the role it plays during the execution of a script.

Stacks, and The Stack

A stack, in general, is a list of values, to which we arbitrarily assign two extremes, a *top* and a *bottom*. We can operate on either side, but as a general simplifying convention, all stack operations will take place at the top unless specified otherwise.

The most fundamental operations that can be carried out on a stack are *pushing* and *popping* values to and from it (at the top).

\

What does that have to do with CaPriCon ? Well, stack-based languages, as their name implies, implicitly operate on a stack, that serves as temporary storage for all the intermediate results of a computation. When we talk about *the stack*, without any more context, you can assume we're talking about this one.

In most stack-based languages, including CaPriCon, words designate *instructions* that modify the stack according to predefined rules, and complex scripts can be written by stringing words together in the right order, changing the stack in ever more interesting ways.

Your First Words

Let's start talking a little. As mentioned above, a sentence (or program) is comprised of several words separated by whitespace. Words can fall into one of three categories :

- *nouns* are constant words, that push a value onto the stack when run
- *verbs* are operational words, that can modify the stack or the environment when they are run
- *quotes* are sequences of *steps*, where a step can be either a word step, or a splice step (quotes and steps will be described in more detail below).

Nouns

The simplest kinds of words are *nouns*, more commonly known as atoms or symbols, and are written as a single quote (`'`), followed by some non-space characters.

```
'Bender
```

As you can see, each noun you write is pushed onto the stack, in the order in which they appear. Nothing mysterious here.

The CaPriCon interpreter also recognizes numbers[1], in the usual decimal format, as another kind of noun, which means they will mostly behave as expected.

```
1 2 100
```

You now know what happens when you write `'vis`, but how does `vis` alone get interpreted ?

Verbs and Vocabularies

In parallel to the stack, the interpreter also features its own *vocabulary*, which provides a correspondance between all the known nouns and their definitions.

Whenever a verb is run, the interpreter looks into its vocabulary for the meaning of that verb and executes it, with a different strategy depending on that meaning :

- if it is a quote, then the interpreter runs each step in the quote

[1] For now, 32-bit integers are the default, but if the need arises, I'll be glad to throw in some BigInt or floating-point support

- if it is a special builtin operation, then that operation is run according to its definition. The initial vocabulary provides a few builtins operations of that sort, which are listed here.
- otherwise, it is simply pushed onto the stack, as a constant

There are two main verbs to interact with the vocabulary : `def`, for adding new definitions, and overriding old ones; and `$`, for looking symbols up. They can be used as follows :

```
'x 3 def 'y 4 def
```

Of course, the most interesting verbs, and the ones I kept for last, are the ones referencing quotes, because they allow you to build upon simpler concepts to yield complex effects.

Quotes

Many stack-based language have features similar to our quotes. Let's start there.

The *raison d'être* of a quote is, simply put, to be able to write a program and keep it in stasis, until it can be run from a verb (or from the stack, using the builtin verb `exec`). In CaPriCon, this can be achieved by enclosing the sentence you want to "freeze" in brackets, like so :

```
pop pop pop 'is 'great
{ swap 2 shift "%s %s %s !" format }
```

That's all fairly straightforward, which is nice, but quotes of this form aren't very dynamic. They will always depend on, and possibly modify, their surrounding environment during evaluation (a feature commonly known as "dynamic binding"), which, while very flexible, doesn't provide a reliable way to write composable programs.

This may not seem like a problem right away, and indeed it isn't for the kinds of small examples we've been playing with, but for more reusable scripts, you should always be careful about the environment you leave behind when you're done with your work. Also, it's kind of a good feeling when you know your programs won't accidentally rewrite an index and start an infinite loop.

Splices and quotes : a case study

To better illustrate the need for a more powerful construct, let's imagine we want to be able to execute a quote in a local environment, so that all nouns defined during that quote's execution don't accidentally override the outside vocabulary.

The idea is to use the `vocabulary` verb to retrieve the vocabulary before executing the quote, save that vocabulary somewhere, then run the quote (which can perform arbitrary modifications to the stack and the vocabulary), and finally restore the old vocabulary afterwards using `set-vocabulary`.

The question is : where do we save the old vocabulary, so that executing our argument won't accidentally override the place we chose ? Given what we know about the stack and the environment, nowhere is safe. A value on the stack can always be `pop`ped or `clear`ed, and a definition in the vocabulary can always be overridden.

Answer : we save it in a quote. Without further ado, here is the solution that CaPriCon proposes :

```
clear 'local-exec {
  { exec ,{ vocabulary } set-vocabulary }
  exec } def
```

Let's break this down : `local-exec` is defined as the quote that, first, creates a new quote by splicing a constant – derived from running `vocabulary` – between executing the top of the stack (our only argument of interest) and resetting the vocabulary to whatever the constant was at the time of creation.

Then, `local-exec` simply calls `exec` to run the newly-created quote that already remembers the `vocabulary` from before. Our argument gets executed, then the old vocabulary that was captured is pushed on the stack, only to be immediately restored to its rightful place by `set-vocabulary`. We now have the newly calculated stack, in an environment where our vocabulary is unchanged.

\

This concludes the tour of all the basic CaPriCon language features. Once you've mastered those three concepts (nouns, verbs and quotes), and learned about enough builtin operations, all the programs, proofs and examples presented from this point on will be within your reach.

There's just one last detail we haven't gotten around to : building actual proofs. Until now, we've been playing around with names and definitions, but we haven't yet proven anything of interest. That's with good reason, because we don't yet know how to build proofs.

Please bear with me for this last section, as I try to explain how to build mathematical proofs out of stacks, quotes and a bit of magic.

Proof assembly

The easiest way to get comfortable manipulating mathematical proofs and theorems is to treat them like regular objects. In CaPriCon, theorems and proofs – which will hereafter be referred to as *terms* – are like numbers and symbols, that can be pushed onto the stack, or saved in the vocabulary.

Types and Universes

The most common kind of basic term you'll encounter are universes, noted Set_n, where $n \in \mathbb{N}$ is the *level* of that universe. You can introduce them with the `universe` builtin, that takes in a level and pushes a universe of that level on the stack :

```
0 universe
```

We just proved something ! Granted, we only proved that some universes exist, by giving an example of one, but still. Using that universe as a starting point, we can explore a bit further.

The first useful thing we can do given a term is ask CaPriCon to give us its type, unsurprisingly by using the `type` builtin.

```
dup type
```

We can see that Set_0 has type Set_1. In general, when a term has type Set_n for some n, we can treat that term as a *type*, that may or may not contain *objects*. Every well-formed term has a type, that can be computed with `type` as we observed, but not every well-formed term *is* a type.

The proof context

Many mathematical proofs begin by assuming the existence of a few objects, before studying those objects in more detail ("let n, m be two natural numbers, ...", "let f be a function from A to B, ..."). Once those objects are *introduced* to the reader, the rest of the proof can refer to them as though they already had a proper value of the given type.

Introducing new hypotheses

CaPriCon works in a similar way. If we have a type on top of the stack, like we do now, we can `intro`duce a variable (or hypothesis) of that type. Introducing

a new hypothesis from a type is equivalent to assuming that at least one term of that type exists, without caring about that term's specific shape.

If our type is a universe, like Set_0, we'll call such a hypothesis a *property* of its type, as a convention. Otherwise, we'll usually call it a *witness* of some property.

```
pop 'Prop intro
```

Using hypotheses

We now have a fresh but unknown property of Set_0, called `Prop`, in the context. We can retrieve that property by using its name, using the `variable` builtin, and check that it is indeed an element of the universe Set_0.

```
'Prop variable
```

We can go further, though. Our property `Prop` is still a type (because it has a type of shape Set_n, remember ?), so we can introduce a witness of it if we want. Let's call that witness `p` :

```
'p intro
```

Clearing hypotheses

Once a variable has been introduced, and used to construct some terms, it can be *extroduced* from the context, which has the effect of closing those terms under binders. The builtin verbs `extro-lambda` and `extro-forall` have the function of extroducing the last hypothesis that was introduced, respectively using lambda abstractions and products ("forall").

In our running example, we can for example create the term $\lambda(\mathrm{p} : \mathrm{Prop}).\, \mathrm{p}$ by extroducing a lambda abstraction after creating the term p .

```
'p variable 'Prop variable extro-lambda
```

Notice how the `p` hypothesis disappeared from the context, only to be found "transferred" to the terms on the stack that reference it. The `Prop` hypothesis was not affected because it couldn't possibly refer to `p`, being defined before it.

\

That's about all there is to hypotheses : use `intro` to create new ones; create some terms using `variable` et al.; and finally, clear them from the environment using some kind of extroduction.

Functions, Products and Applications