Toward a fully formalized definition of syllepsis in weak higher-categories

Thibaut BENJAMIN

The aim of this presentation is to illustrate the use of formal methods in order to reason in the theory of weak ω-categories. The formalism considered here is based on the type theory CaTT introduced by Finster and Mimram [1], extended with some metatheory on top of it, of which an implementation is freely available [2].

We first present how the system works and can be used, and then develop some "real-world" examples, such as the definition of the braidings in k-tuply monoidal ω-categories (following the terminology of Baez [3]), $k \geq 2$, with the aim of showing that we have a syllepsis in the case $k \geq 3$. These developments have motivated metatheoretical improvements to the proof assistant, which we also present and discuss here, in order to handle and partly automate large proofs.

Being able to prove the existence of such a morphism is a critical test for this theory, as it is one of the first of the expected results of higher category theory which has not been proved. It also constitutes a strong indicator of the correctness of the implementation.

1 The type theory of weak ω-categories

We begin by giving a quick and informal overview on the CaTT type theory [1], which can be thought of as the internal language for weak globular ω-categories, on which the proof assistant we are using here is based.

In this type theory, there are two type constructors, $\star$ of arity 0, and $\to$ of arity 3, along with the corresponding introduction rules

$$\frac{}{\Gamma \vdash \star} \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x \underset{A}{\to} y}$$

At this point, the models of the type theory are globular sets: $\star$ is the type of 0-cells (or points) of a globular set, and $x \underset{A}{\to} y$ is the globular set between two n-cells x and y, both being of the same type A, so both being parallel (or objects). Moreover, a context in this theory can be thought of as a description of a finite globular set.

In order to have categories as models of the type theory, we should also have ways to compose cells. Here, compositions are expressed in an *unbiased* fashion, meaning that we should have, as primitive operations, all the possible compositions of reasonable shape: we can characterize the globular sets that we want to be able to compose and call them *pasting schemes.* This is achieved by introducing a new judgment $\Gamma \vdash_{\mathsf{ps}}$, expressing that the context Γ is a pasting scheme (when considered as a finite globular set) as well as four associated inference rules. Moreover, any pasting scheme comes equipped with a *source* and a *target* pasting scheme, respectively denoted $\partial^{-}\Gamma$ and $\partial^{+}\Gamma$, which can be computed by induction on Γ. We can understand these notions by considering the type of the cell resulting from the composition of a pasting scheme Γ: it should be a cell, which goes from a composition of the source of the pasting scheme, to a composition of its target. We can finally complete the definition of the type theory, by adding a rule expressing the fact that every pasting scheme should have a composite, which we name using a new term constructor coh: given a pasting scheme Γ, and two terms t and u of the same type corresponding to ways to compose its source and target, it produces a term $\mathsf{coh}_{\Gamma,t\to u}$ of type $u \to t$. Formally, we have the rule

$$\frac{\Gamma \vdash_{\mathsf{ps}} \qquad \Gamma \vdash t \underset{A}{\to} u \qquad \partial^-\Gamma \vdash t : A \qquad \partial^+\Gamma \vdash u : A}{\Gamma \vdash \mathsf{coh}_{\Gamma,t \underset{A}{\to} u} t \underset{A}{\to} u}$$

(along with appropriate side conditions). This rule allows to generate all the operations in higher categories, but it remains to generate coherences, expressing for instance that two composite of a given pasting scheme are equivalent. This is achieved by the rule

$$\frac{\Gamma \vdash_{\mathsf{ps}} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{coh}_{\Gamma,A} : t \underset{A}{\to} u}$$

(along with appropriate side conditions) which, given a pasting scheme Γ and two terms t and u of the same type corresponding to two ways of composing Γ, produces a new term $\mathsf{coh}_{\Gamma,u\to t}$ of type $u \to t$.

It is important to notice here that pasting schemes intuitively correspond to a configurations of cells in a *generic position*: the cells are variables, there are no loops, etc. Deriving a new cell in such a context is often a primitive operation in this theory. However, whenever we want to derive a new cell in a specific context which is not generic, we will always start by showing what this new cell is in a generic position, and then apply the whole construction in our particular case.

2 Formal proofs in 2-tuply monoidal categories

2.1 First examples using `catt`

We can now begin to show the use of the tool [2] resulting from the type theory described above. The syntax to introduce a term constructed by `coh` is the following

```
coh [name : string] [C : Context] : [A : Type].
```

where `C` is a constext describing a pasting scheme, and `A` is the type in the context `C` describing the new axiom we are defining.
We begin by simple examples coming from (1-) category theory.

- The identity morphisms, and composition are both primitive, as they are defined in generic positions:

  ```
  coh id (x : *) : x -> x.
  coh comp (x : *) (y : *) (f : x -> y) (z : *) (g : y -> z) : x -> z.
  ```

- The identity morphism is the right unit for the composition. Here, there are infinitly many morphisms to introduce in order to show this, all of them are primitive operations, and we only show how to introduce the first two, without going higher in the coinductive statement:

  ```
  coh unitr  (x : *) (y : *) (f : x -> y) : comp x y f y (id y) -> f.
  coh unitr- (x : *) (y : *) (f : x -> y) : f -> comp x y f y (id y).
  ```

- There is a square (or self-composition) operation on endomorphisms of an object. This is not a primitive operation because an arrow being an endomorphism is not a generic situation. This operation is however definable from primitive operations:

  ```
  let square (x : *) (f : x -> x) : x -> x = comp x x f x f.
  ```

As we can see in the previous example, there are two ways of introducing new operations. The first one is when the operation is primitive, and it is introduced with the keyword `coh`. This sort of introduction is axiomatic in the system, and thus is a valid defintion. The second way is for non-primitive operations, and is introduced by the keyword `let`. This method of introduction has to give a valid definition of the newly defined term, in terms of previously known terms, i.e., ultimately in terms of applied primitive operations.

When we apply an operation, we are supposed to specify all the arguments of this operations, that is why here we have compositions like `comp x y f y (id y)`. However, many of these arguments are in fact non relevant, because they can be recovered with only fewer arguments (e.g. `x` and `y` can be inferred from the type of `f`). In the system we are using, there is an algorithm to specify instantly which arguments have to be explicit, and recover all the data to build the complete terms. We do not detail this algorithm which is based on classical unification. However, from now on, we will write the formalizations as if this algorithm was known. The previous definitions thus become

```
coh unitr  (x : *) (y : *) (f : x -> y) : comp f (id y) -> f.
coh unitr- (x : *) (y : *) (f : x -> y) : f -> comp f (id y).
let square (x : *) (f : x -> x)         : x -> x = comp f f.
```

2.2 The braiding

Let us now give a full description of the formalisation of the *braidings*, in 2-tuply monoidal ω-categories. We will start with a quick reminder and few definitions, to explain precisely what we want to formalise. We freely use the formal definition given in section 2.1.

Definition 1. A *(1-tuply) monoidal ω-category* is a weak ω-category in which there is only one 0-cell.

In our formalism, there is no way of imposing that there is exactly one 0-cell, but we can consider the monoidal category whose cells have a given 0-cell `x` as iterated 0-source and 0-target. For instance, the definition of the tensor product of 1-cells is thus:

```
let tens (x : *) (a : x -> x) (b : x -> x) : x -> x = comp a b.
```

and similarly for higher-cells.

Definition 2. A *2-tuply monoidal ω-category* is a weak ω-category in which there is only one 0-cell x and one endomorphism of x, namely the identity on x.

The tensor product $\otimes$ can be defined as above and we would like to show the existence of a *braiding* for every pair of 2-cells a and b

$$\gamma \quad : \quad a \otimes b \quad \to \quad b \otimes a$$

The aim of this section is to formalise the existence of such a morphism. For this purpose, we can start thinking of the context we want to define the operation in. Two 2-cells in a 2-tuply monoidal category correspond to the following context of the underlying category:

```
(x : *) (a : id x -> id x) (b : id x -> id x)
```

This context does not describe at all a generic situation, so we have to define the braiding as a composition of more primitive operations. We start by defining the tensor product on the objects of a 2-tuply monoidal category. For that, we need to define the composition on 2-cells of a category in general, and apply it in our specific context. We have two choices on how to compose these 2-cells (horizontal or vertical composition). Here, we chose the vertical composition.

```
comp-two-cells
  (x : *) (y : *) (f : x -> y) (g : x -> y) (h : x -> y) (a : f -> g) (b : g -> h) :
  f -> h.
```

Now, we can define the tensor product on the 2cells of a 2-tuply monoidal category, by using this composition:

```
let tensor-braid
  (x : *) (a : id (id x) -> id (id x)) (b : id (id x) -> id (id x)) :
  id (id x) -> id (id x) = comp-two-cells a b.
```

Remark 1. In the definition of an operation introduced with `let`, precising the type of the term we are defining is not useful for the system, it is more of a sanity check for the user. We will thus omit this type unless it helps understanding. For instance, we could have written the previous introduction:

```
let tensor-braid (x : *) (a : id (id x) -> id (id x)) (b : id (id x) -> id (id x))
  = comp-two-cells a b.
```

Let us now define the braiding. A geometric visualisation of the situation is as follows:

$$\begin{tikzcd}
x \arrow[r, "\mathrm{id}_x" description] \arrow[r, bend left=70, "\mathrm{id}_x"] \arrow[r, bend right=70, "\mathrm{id}_x"'] \arrow[r, phantom, shift left=2ex, "\Downarrow a"] \arrow[r, phantom, shift right=2ex, "\Downarrow b"] & x & \rightsquigarrow & x \arrow[r, "\mathrm{id}_x" description] \arrow[r, bend left=70, "\mathrm{id}_x"] \arrow[r, phantom, shift left=2ex, "\Downarrow a"] & x \arrow[r, "\mathrm{id}_x" description] \arrow[r, bend right=70, "\mathrm{id}_x"'] \arrow[r, phantom, shift right=2ex, "\Downarrow b"] & x & \rightsquigarrow & x \arrow[r, "\mathrm{id}_x" description] \arrow[r, bend right=70, "\mathrm{id}_x"'] \arrow[r, phantom, shift right=2ex, "\Downarrow a"] & x \arrow[r, "\mathrm{id}_x" description] \arrow[r, bend left=70, "\mathrm{id}_x"] \arrow[r, phantom, shift left=2ex, "\Downarrow b"] & x & \rightsquigarrow & x \arrow[r, "\mathrm{id}_x" description] \arrow[r, bend left=70, "\mathrm{id}_x"] \arrow[r, bend right=70, "\mathrm{id}_x"'] \arrow[r, phantom, shift left=2ex, "\Downarrow b"] \arrow[r, phantom, shift right=2ex, "\Downarrow a"] & x
\end{tikzcd}$$

We first notice that the operation in the middle is an exchange law, and it is a primitive operation in the categories. to define it, we need to define the left and right whiskering

```
coh whiskl (x : *) (y : *) (f : x -> y) (z : *) (g : y -> z) (h : y -> z) (a : g ->h) :
comp f g -> comp f h.

coh whiskr (x : *) (y : *) (f : x -> y) (g : x -> y) (a : f -> g) (z : *) (h : y -> z) :
comp f h -> comp g h.

coh exch (x : *) (y : *) (f : x -> y) (g : x -> y) (a : f -> g)
                 (z : *) (h : y -> z) (k : y -> z) (b : h -> k):
comp (whiskr a h) (whiskl g b) -> comp (whiskl f b) (whiskr a k).
```

Now, applying the exchange rule in our special case where we have 2-cells of a 2-tuply monoidal category, we get the following

```
let exch-braid (x : *) (a : id x -> id x) (b : id x -> id x) = exch a b.
```

This is the core of the braiding, but it is not sufficient to define the entire morphism, it is only an intermediate rewriting step. We now have to understand how to get into a situation where we can use the exchange law, from the plain tensor product of a and b. Unfortunately, we cannot, as we would like to, produce an arrow in the generic situation

$$\begin{tikzcd}
x \arrow[r, "g" description] \arrow[r, bend left=70, "f"] \arrow[r, phantom, shift left=2ex, "\Downarrow a"] & y & \rightsquigarrow & x \arrow[r, "g" description] \arrow[r, bend left=70, "f"] \arrow[r, phantom, shift left=2ex, "\Downarrow a"] & y \arrow[r, "\mathrm{id}_y" description] & y
\end{tikzcd}$$

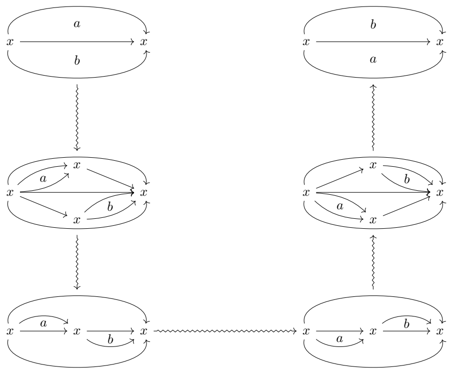
because these two 2-cells are not parallel. Indeed, the first one is from `f` to `g` whereas the second one is from `comp f (id y)` to `comp g (id y)`. However, we do have cancelation witnesses for the right composition with the identitiy, and we can compose the diagram in the following way

$$\begin{tikzcd}
x \arrow[r, "g" description] \arrow[r, bend left=70, "f" description] \arrow[r, phantom, shift left=2ex, "\Downarrow a"] \arrow[rr, bend left=60, "f"] \arrow[rr, bend right=40, "g"'] & y \arrow[r, "\mathrm{id}_y" description] & y
\end{tikzcd}$$

In order to describe this composition, we need the ternary composition on 3-cells, that we will just assume we have, since it is very similar to the binary one we have already explained. We can then derive as a primitive operation, the arrow we wanted:

```
coh whiskr-by-id-intro (x : *) (y : *) (f : x -> y) (g : x -> y) (a : f -> g) :
  a -> comp-2-cells-tern (unitr f) (whiskr a (id y)) (unitr- g).
```

Similarly, we can write the introduction of the whiskering by identity on the left, and we can also derive an elimination for these two operations, by simply reversing the order of the arrows. We now have to figure out how to use them. If we take our diagramatic proof back, and insert these precisions, we get a more richer diagram (we omit the names of the arrows, for readability):



Remark 2. The upside of this theory is that one can define general lemmas before applying, and thus factor the difficult definitions into applications of general steps. For instance, we can define a witness for the cancellation of a composition with its inverse.

```
let cancelW-comp (x : *) (y : *) (z : *)
                 (f : x -> y)  (f- : y -> x) (g : y -> z) (g- : z -> y)
                 (ff- : comp f f- -> id x) (gg- : comp g g- -> id y)
: comp (comp f g) (comp g- f-) -> id x
= [a term definable with primitive operations]
```

Once this term is defined, using the implicit arguments, we need to specify only two arguments to apply it in a specific case. Thus this term can be thought of as a tactic similar to what we can do in Coq, as it is a strategy to perform complicated operations. This is the correct way of tackling the formalisation problem.

Once all the appropriate lemmas are defined, and the global structure of the definition is has been factored in tactics, defining the braiding is rather straightforward, and takes 10 lines of formalisation. The uspide of factoring all this through lemma, is that we can now define the inverse braiding, which follows the same structure, and is thus also definable in 10 lines. More importantly, we can also easily define a witness for the cancellation of the braiding with its inverse, by factoring the definintion the same way, and defining appropriate cancellation witnesses directly for the tactics, which is much easier. Once this is done, and some appropriate easy additional lemmas are proved, defining the cancellation witness can also be done in 10 lines.

However it takes a fair amount of work to formalise all the machinery to reach that point, as about 830 lines are necessary before being able to start defining the braiding. Among these, around 300 lines are dedicated to defining the very standard operations of compositions, associativties and unit laws, in different arities and dimensions, then about 200 lines are used to define the some coherences of these operations, allowing various rewriting and cancelling steps in the aforementioned compositions. We then need around 300 lines to define various tactics, combining the previous coherences into bigger terms that factor the important steps of many proofs, and finally about 30 lines to define some *ad hoc* operations appearing in 2-tuply monoidal ω-categories.

2.3 Implementation challenges

When the proofs become conger and longer, it becomes more and more challenging for the software to handle type checkings. As a result, the initial implementation we had was not sufficiently efficient, and every run of the programm was taking too much time, prohibiting any further development of the proof. This mostly comes from the fact the type theory [1] only cares about the primitive operations. As we have no mathematical proof yet that adding non primitive operations do not change the models of the theory, we chose to stick strictly to primitive operations, and thus to unfold all the non primitive ones, everytime they appear. That explains the growth of the term size, and the time spent checking these terms.
To solve this issue, we have implemented some ways to speed up the typechecking process. The first idea is to be able to carry already checked subterm inside an unchecked term. This makes the previous idea of factoring terms into tactics particularly relevant, as it also contributes to a faster checking. This alone was not sufficient, so we also used a hashtable to avoid checking the same terms too many times.

3 Formalisation in 3-tuply monoidal ω-categories

We will now discuss a little about the 3-tuply monoidal ω-categories.

Definition 3. A 3-*tuply monoidal ω-category* is a weak ω-category in which there is only one object (again denoted x), the identity on x is the only endomorphomism of x, and the identity 2-cell is the only 2-cell.

As before, in a 3-tuply monoidal category, we can define a tensor product, along with a braiding. But now the braiding is involutive. More precisely, there is an invertible 2-cell:

$$\gamma(a,b) \circ \gamma(b,a) \quad \to \quad \mathrm{id}_{a\otimes b}$$

Remark 3. In the previous section, we have defined the braiding for 2-tuply monoidal categories, and here we are again talking about braiding, but in 3-tuply monoidal categories. These two morphisms are defined the exact same way, the only difference is that objects are now arrows. To avoid redefining the same operation in all the dimensions, we have an algorithm that can generate the operation in the new dimension when needed. So we can simply write

```
let (x : *) (a : id (id x) -> id (id x)) (b : id (id x) -> id (id x)) = γ a b.
```

This will be interpreted as generating a new appropriate operation, and applying it

We have formalised some results in 3-tuply monoidal, ω-categories, towards a definition of the syllepsis. The main result that we have yet is that there are in fact two different braidings. This is in fact much more technical than the previous braiding, as it emphasises the importance of dependencies, and it takes 500 more lines of formalisation to define. Nevertheless, the ideas are interesting and presented graphically in the appendix.

To finish the definition of the syllepsis, we still have to prove that both of the braidings are equivalent, and that will conclude.

4 Further developments

Formalising these results have made clearer various ways of improving our system. The first idea is to be able to define families of operations and axioms indexed by integer. That would let us avoid defining the composition in every arities for example, and allow us to define them all at once. Another powerful idea is to handle automatically the inverses, avoiding us having to prove cancellation lemmas that are straghtforward and could have been derived algorithmically

References

[1] E. Finster and S. Mimram, ‘A type-theoretical definition of weak ω-categories’, in *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2017, pp. 1–12. DOI: `10.1109/LICS.2017.8005124`. arXiv: `1706.02866`.

[2] (2018). Github catt, [Online]. Available: `https://github.com/ThiBen/catt`.

[3] J. Baez, ‘Lectures on n-categories and cohomology’, Notes by M. Shulman.

[4] (2018). Nlab - periodic table, [Online]. Available: `https://ncatlab.org/nlab/show/periodic+table`.

Appendix

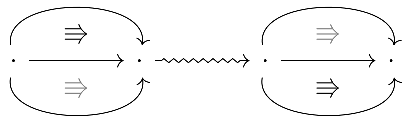
A Formal definition of the braiding

We present here the formal definition of braiding in 2-tuply monoidal ω-categories, as proved in catt. For the sake of the explaination, we have used in this presentation slightly different naming conventions, however, the global structure of the proof id the same.

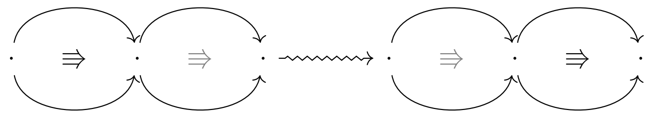
```
let eh (x : *) (a : id x -> id x) (b : id x  -> id x) =
let u  = unitr  (id x) in
let u- = unitr- (id x) in
let v  = unitl  (id x) in
let v- = unitl- (id x) in
    comp5 (hcomp (whiskrScan- a) (whisklScan- b))
          (red3F u- (whiskr a (id x)) (hinvrl x) (whiskl  (id x) b) v)
          (rew3A (equivrl- x) (exch a b) (equivlr x))
          (exp3F v- (whiskl (id x) b) (hinvlr- x) (whiskr a (id x)) u)
          (hcomp (whisklScan b) (whiskrScan a)).
```

B The second braiding in 3-tuply monoidal categories

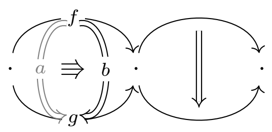
The idea behind the second braiding is to go through a second sort of exchange law. The morphism we have already defined uses the following exchange rule :



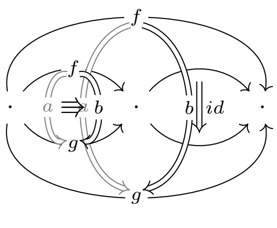
but now we are going to define a new braiding using the following exchange rule :



So now the question that is posed to us is how to reach a state where this step applies. This is not as easy as it was before although it follows the same general structure. The idea is to introduce first a equivalent of the whiskering as follows (we could call this operation a (3,0,2)-composition) :



Then, we want to define a cell witnessing that composing a 3-cell this way with the identity 2-cell does not change the 3-cell. We cannot immediately have such a witness because of a parallelism problem, so we have to first understand how to compose the (3,0,2)-composition with indentity, to bring it back on the same source and target as the orgiginal 3-cell. This can be done in the following way :



As the diagramms become difficult to visualise in three dimensions, we will now reduce the dimension by one, and draw all cells one dimension lower. The previous diagramm then becomes (where the blue arrows correspond to the different cancelletions of identity on the right):

$$\begin{tikzcd}
 & & f \arrow[d, blue] \arrow[dll, bend right, no head] \arrow[drr, bend left, no head] & & \\
 & & f \circ id \arrow[dl, no head] \arrow[dr, no head] & & \\
a \arrow[r, Rightarrow, blue] \arrow[ddrr, bend right] & a\star id \arrow[rr, Rightarrow] \arrow[dr] & & b\star id \arrow[r, Rightarrow, blue] \arrow[dl] & b \arrow[ddll, bend left] \\
 & & g \circ id \arrow[d, blue] & & \\
 & & g & &
\end{tikzcd}$$

We do the same thing when (3,0,2)-composing with the 2-cell identity on the left, and we get the following diagramm (where the red arrows correspond to the different cancelletions of identity on the left)

$$\begin{tikzcd}
 & & f \arrow[d, red] \arrow[dll, bend right, no head] \arrow[drr, bend left, no head] & & \\
 & & id \circ f \arrow[dl, no head] \arrow[dr, no head] & & \\
a \arrow[r, Rightarrow, red] \arrow[ddrr, bend right] & id\star a \arrow[rr, Rightarrow] \arrow[dr] & & id\star b \arrow[r, Rightarrow, red] \arrow[dl] & b \arrow[ddll, bend left] \\
 & & id \circ g \arrow[d, red] & & \\
 & & g & &
\end{tikzcd}$$

If we now apply these diagramms to objects in a 3-tuply monoidal ω-category, and try to compose them, as we would want to do after using them to rewrite the original 3-cells, we get the following :

$$\begin{tikzcd}
 & & id \arrow[d, blue] \arrow[dll, bend right, no head] \arrow[drr, bend left, no head] & & \\
 & & id \circ id \arrow[dl, no head] \arrow[dr, no head] & & \\
id \arrow[r, Rightarrow, blue] \arrow[ddrr, bend right] & id\star id \arrow[rr, Rightarrow] \arrow[dr] & & id\star id \arrow[r, Rightarrow, blue] \arrow[dl] & id \arrow[ddll, bend left] \\
 & & id \circ id \arrow[d, blue] & & \\
 & & id & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & id \arrow[d, red] \arrow[dll, bend right, no head] \arrow[drr, bend left, no head] & & \\
 & & id \circ id \arrow[dl, no head] \arrow[dr, no head] & & \\
id \arrow[r, Rightarrow, red] \arrow[ddrr, bend right] & id\star a \arrow[rr, Rightarrow] \arrow[dr] & & id\star b \arrow[r, Rightarrow, red] \arrow[dl] & id \arrow[ddll, bend left] \\
 & & id \circ id \arrow[d, red] & & \\
 & & id & &
\end{tikzcd}$$

Unfortunately, if we do this it is impossible to cancel the blue and red 2-cells, because they are not parallel. So we have to find a new trick to be able to cancel them. To deal with this problem, we can use the witness that left and right cancellation are equivalent, so we can insert the following cells (drawn as 2-cells, but which are in fact 3-cells) :

$$\begin{tikzcd}
id \arrow[d, bend right=60, blue, ""{name=L}] \arrow[d, red, ""{name=M}] \arrow[d, bend left=60, blue, ""{name=R}] \arrow[from=L, to=M, Rightarrow, shorten=2pt] \arrow[from=M, to=R, Rightarrow, shorten=2pt] \\
id \circ id
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
id \circ id \arrow[d, bend right=60, red, ""{name=L}] \arrow[d, blue, ""{name=M}] \arrow[d, bend left=60, red, ""{name=R}] \arrow[from=L, to=M, Rightarrow, shorten=2pt] \arrow[from=M, to=R, Rightarrow, shorten=2pt] \\
id
\end{tikzcd}$$

We insert these cells in the previous diagramm ad follows :

$$\begin{tikzcd}
& & id \arrow[d, bend right=60, blue, ""{name=L}] \arrow[d, red, ""{name=M}] \arrow[d, bend left=60, blue, ""{name=R}] \arrow[from=L, to=M, Rightarrow, shorten=2pt] \arrow[from=M, to=R, Rightarrow, shorten=2pt] \arrow[dll, dash, bend right] \arrow[drr, dash, bend left] & & \\
& & id \circ id \arrow[dl, dash] \arrow[dr, dash] & & \\
id \arrow[r, Rightarrow] \arrow[drr, bend right] & id \star id \arrow[dr] \arrow[rr, Rightarrow] & & id \star id \arrow[dl] \arrow[r, Rightarrow] & id \arrow[dll, bend left] \\
& & id \circ id \arrow[d, blue] & & \\
& & id & &
\end{tikzcd}
\quad
\begin{tikzcd}
& & id \arrow[d, red] \arrow[dll, dash, bend right] \arrow[drr, dash, bend left] & & \\
& & id \circ id \arrow[dl, dash] \arrow[dr, dash] & & \\
id \arrow[r, Rightarrow] \arrow[drr, bend right] & id \star a \arrow[dr] \arrow[rr, Rightarrow] & & id \star b \arrow[dl] \arrow[r, Rightarrow] & id \arrow[dll, bend left] \\
& & id \circ id \arrow[d, bend right=60, red, ""{name=L}] \arrow[d, blue, ""{name=M}] \arrow[d, bend left=60, red, ""{name=R}] \arrow[from=L, to=M, Rightarrow, shorten=2pt] \arrow[from=M, to=R, Rightarrow, shorten=2pt] & & \\
& & id & &
\end{tikzcd}$$

Now, we can cancel the middle part of the diagramm by reassociating, and cancelling the following diagram, as a primitive operation:

$$\begin{tikzcd}
& & id \arrow[dll, red, bend right, ""{name=A}] \arrow[dll, blue, bend left, ""{name=B}] \arrow[from=A, to=B, Rightarrow, shorten=2pt] \arrow[drr, red, bend left] \arrow[dd, dash] & & \\
id \circ id \arrow[d, dash] & & & & id \circ id \arrow[d, dash] \\
id \star id \arrow[d] \arrow[rr, Rightarrow] & & id \arrow[dd] \arrow[rr, Rightarrow] & & id \star id \arrow[d] \\
id \circ id \arrow[drr, blue, bend right] & & & & id \circ id \arrow[dll, red, bend right, ""{name=C}] \arrow[dll, blue, bend left, ""{name=D}] \arrow[from=C, to=D, Rightarrow, shorten=2pt] \\
& & id & &
\end{tikzcd}$$

This allows us to reach a situation where the wanted exchange law can apply, and then replying the inverse of all the machinery we just describes brings us back to a tensor product. So combining all these steps allows us to define a new braiding, that exists in 3-tuply monoidal categories, but not in 2-tuply monoidal categories