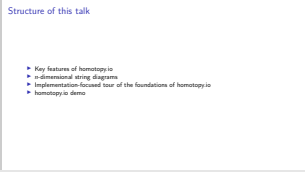


Thank you for inviting us to speak today. Many of you may have seen homotopy.io before in other contexts, but this talk is specifically about the implementation of our tool as opposed to its mathematical foundations. For this reason, we will not be discussing the mathematical details of higher categories, but rather the technical details of how we have implemented a proof assistant for working with them. We will give a split presentation - I'll present some slides for the first half of the talk, and then Calin will present a demo of the tool.

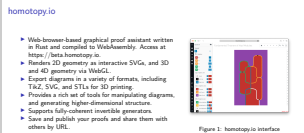


Figure 1: homotopy.io interface

homotopy.io is a graphical proof assistant for working with finitely-presented semistrict higher categories as n-dimensional string diagrams. It is written in Rust, and compiles to WebAssembly to run in the web browser. No installation is required, and it can be accessed currently at https://beta.homotopy.io. It can be used for simple cases like drawing TikZ of string diagrams, or to build up complex string-diagrammatic proofs, as the system checks that each input is admissible. Interaction all happens through a point-and-click interface, which triggers the recursive algorithms which manipulate the underlying combinatorial encodings - I won't have time to go into much detail about this, but please see our accompanying paper and also the previous bodies of work on homotopy.io for details. homotopy.io now has support for coherently-invertible generators, which simplifies working with equational theories of string diagrams. You can save your proofs and share them with others by URL, as well as publish them permanently on the homotopy.io website in an arXiv-like fashion so that it can be included as a reference in a paper.

n-dimensional string diagrams

Figure 2: 2D string diagram representing $f \oplus id(W) \cdot id(X) \oplus g$

String diagrams provide a graphical calculus for reasoning about morphisms in monoidal categories. In a certain vein, monoidal categories are a special case of higher categories; ergo, generalising away from this setting yields a generalised string diagram: one which is topologically n-dimensional.

homotopy.io is a proof assistant based on a model of higher category, Associative n-categories, which has this idea at its core. This model has strict associativity and unitality laws, meaning that string diagrams can be written down in the usual way with no ambiguity about bracketing of terms, but weak interchange. Weak interchange means that each admissible deformation of a n-dimensional string diagram is *proof relevant*; that is, it admits a $(n+1)$-cell from the source to the target of the deformation which when rendered in n-dimensional space looks like a homotopical deformation, hence the name homotopy.io. In other words, instead of having *equalities* between different string diagrams, we have directed *rewrites* between them, which themselves have data attached to them.

Consider the 2D string diagram shown here. An admissible deformation of this diagram is the planar isotopy given by interchanging the morphisms f and g.

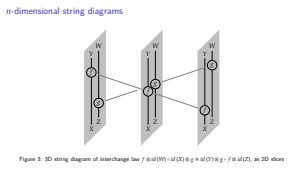


Figure 3: 3D string diagram of interchange law $f \otimes id(W) \cdot id(X) \otimes g \approx id(Y) \otimes g \cdot f \otimes id(Z)$, as 2D slices

Performing this move, we have the diagram from which we started on the left, and the resulting diagram on the right. This whole process can be thought of as a 3D string diagram, with the central slice representing the *singular* moment in which the morphisms f and g occupy the same height.

[next slide]

The primary interface of our tool is the display and manipulation of 2D slices of these n-dimensional string diagrams. To generate this homotopy move, the user would click-and-drag the morphism node labelled f until it is below g, and the tool responds to either tell the user the move is inadmissible, or generates a rewrite of diagrams which is then applied to show the new state. In this case, the move is admissible, so the tool would respond with the diagram contained in the slice in the middle. Another downward drag on f would then complete the interchange, resulting in the diagram on the right.

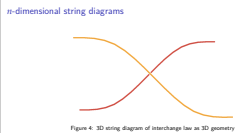
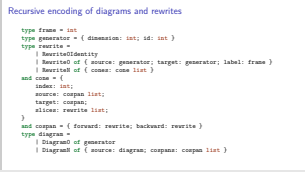


Figure 4: 3D string diagram of interchange law as 3D geometry

Our tool can render this as this 3D geometry, which is a braid. The user would be able to pan around this geometry in 3D space.
4D diagrams can also be rendered as an interactive animation of 3D geometry evolving over time.
[back slide]



Here is some pseudocode for the data structures which encode the diagrams and rewrites in homotopy.io. Diagrams and rewrites are defined recursively, and we use lightweight dependent typing to distinguish between the zero and higher-dimensional cases.

Diagrams are either 0-dimensional, consisting of a single generator (which is essentially a name), or higher-dimensional, consisting of a source diagram and a list of cospans which iteratively forwards and backwards rewrite that source to the target.

A rewrite between 0-dimensional diagrams is either the identity or a map between 0-diagrams which carries some framing data. This framing data is like a direction of the rewrite with respect to its coordinate system in n-dimensional space, for example something like north-to-south in 2D, and is used internally for generating the coherently-invertible structure and also plays a role in typechecking. A rewrite between $(n+1)$-dimensional diagrams is a list of cones, which sparsely encodes a collection of rewrites between n-dimensional diagrams in its `slices` field. In an n-dimensional string diagram, for any given point the surrounding local piece of diagram is likely to be the identity, and it is these pieces of topology which are elided by the cone encoding. This is analogous to how an adjacency list representation of a directed graph sparsely records the edge relation as opposed to an adjacency matrix.

Example 2D diagram encoding

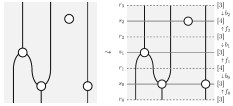


Figure 5: Encoding a 2D string diagram (reutterHighlevelMethodsHomotopy2019)

Let's move to a worked example.

This picture on the left is a 2D string diagram. Scanning horizontally up and down the page, each horizontal slice itself is a 1D string diagram. The very first slice r_0, from the bottom, intersects 3 of the vertical wires in the string diagram (hence the marking 3 on the right), and is indistinguishable from its neighbours until the slice marked s_0 is reached - this is where *something happens* in the diagram. The same is true above s_0 until s_1 is reached, and so on. We use this to stratify the diagram into singular levels, which have intervening regular levels and rewrites of 1D diagrams between them. Now, using our encoding, this 2D diagram is given by the 1D diagram determining its source on the bottom, along with this collection of rewrites which arrange into an iterated cospan.

[back slide]

Example 2D diagram encoding

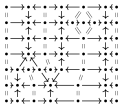


Figure 6: ?? wireframe, with explicit rewrites

This picture makes all the underlying 0-dimensional diagrams and rewrites which constitute the diagram. It's not shown in the picture, but typically one would want to work with diagrams with respect to some signature, i.e. one in which each of these points have some kind of type information attached to them. For instance, this central point on the monoid-shaped thing might have its generator associated to an algebraic symbol m, and the points along the wire may be associated to a symbol A, and in this sense this piece of diagram would encode a monoid m with carrier A. Framing data has also been omitted from the picture, but for example it allows for the rewrites constituting both legs of the monoid to be distinguished.

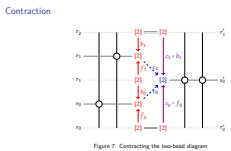


Figure 7: Contracting the two-bead diagram

I'd just like to give a bit of the flavour of the kind of algorithmic tools we have in homotopy.io. We call the move which is obtained from starting with the diagram on the left and either clicking-and-dragging the bottom-left bead upwards, or the top-right bead downwards. This is because it shrinks the height of this diagram — on the left, we have two singular levels, and on the right we have 1 singular level. What this algorithm does is it computes some kind of colimit of the iterated cospan on the left, which would be the only singular level of the diagram on the right, and then reconstructs the rest of the right-side diagram, and also the 2D rewrite which goes from left-to-right. The fact that it's a colimit means that this is the *canonical* way to shrink the diagram, and it doesn't always exist because some contraction moves won't be admissible.

A more detailed description of what this algorithm does is in the paper, but essentially it, like most of our important algorithms, is an algorithm which is recursive on diagram dimension — and this partly motivates the algebraic datatypes we chose on the previous slides to encode diagrams and rewrites. Handwaving a lot, there is some strongly-connected component calculation on some directed graph which underlies this diagram on the left, which generates the correct shape of the desired colimit, and then this would be correctly filled with typing information on each node and edge whenever the contraction move exists.