Imperial College London

Stringent

Authors:
Csongor Kiss
Toby Shaw
Ignacio Navarro
Daniel Slocombe
James Long
Thomas Grigg

Supervisors:
Prof. Susan Eisenbach
Dr. Tony Field

Contents

1 Introduction

Category theory is a growing branch of mathematics that deals with mathematical structure. It has vast applications due to its highly abstract nature - it is often of interest to computer scientists due to its applications in functional programming.

Proofs in this field are typically very synctatic, which can lead to bloated arguments that obscure the semantics. A modern approach to solving this problem is obtained through the use of diagrammatic proofs using string diagrams. String diagrams are a two-dimensional notation for expressions in category theory. They visually embed some of the laws of category theory, resulting in intuitive and elegant proofs.

Currently it is common practice to draw and manipulate string diagrams manually due to a lack of relevant tools. This heavily detracts from the elegance of using string diagrams, indicating a need for a tool capable of automating this process.

We present *Stringent*, an interactive proof assistant for generating equational proofs in category theory using string diagrams. The tool:

- Allows users to make steps in a diagrammatic proof, verifying these steps in real time and recommending subsequent steps.
- Accepts textual input from a well-defined language specification.
- Provides a clean and intuitive user interface.
- Allows string diagrams to be exported for use in academic papers.
- Is realised as a web-application for easy online access.

2 Brief Introduction to Category Theory

"Category theory starts with the observation that many properties of mathematical systems can be unified and simplified by a presentation with diagrams of arrows." - Saunders Mac Lane

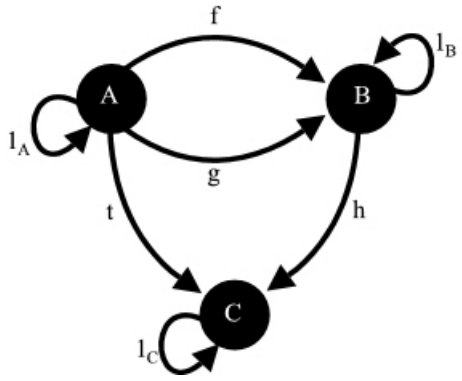
Often mathematical objects exist with the notion of a structure-preserving mapping defined upon them. A category is itself a mathematical object that aims to axiomatically capture the structure of such mathematical objects. Formally:

Definition 1. A **category** $\mathcal{C}$ is defined as a triple: $\mathcal{C} = (\text{ob}(\mathcal{C}), \text{hom}(\mathcal{C}), \circ)$ where

- $\text{ob}(\mathcal{C})$ is a class whose elements are called **objects**.
- $\text{hom}(\mathcal{C})$ is a class whose elements are called **morphisms** (or maps or arrows). An element m of $\text{hom}(\mathcal{C})$ must specify a single ordered pair (A, B), $A, B \in \text{ob}(\mathcal{C})$ where A is referred to as the **source object** and B as the **target object**. We often write $m \in \text{hom}(\mathcal{C})$ as $m : A \to B$ to make this clear, but this notation should not be confused with that of a function between two sets.
  Further, we write $\text{hom}(A, B)$ to denote the **hom-class** from A to B, i.e. the class of all morphisms with specified pair (A, B).
- $\circ$ is a binary operation $\text{hom}(B, C) \times \text{hom}(A, B) \to \text{hom}(A, C)$, called **composition of morphisms** which conforms to the following axioms:
  - (associativity) if $f : A \to B$, $g : B \to C$ and $h : C \to D$ then $h \circ (g \circ f) = (h \circ g) \circ f$
  - (identity) for every object $X \in \text{ob}(\mathcal{C})$, there exists a morphism denoted $1_X : X \to X$ and called the identity morphism for X, such that for every morphism $f : A \to X$ and every morphism $g : X \to B$, we have $1_X \circ f = f$ and $g \circ 1_X = g$.

  .

E.g.



Here, we must have $h \circ f = t = h \circ g$.

Our motivation for formally defining the notion of a category came from wanting to explore the structure of mathematical objects. Of course, a category is itself a mathematical object - so what would a structure-preserving mapping look like between two categories?

Definition 2. A **functor** is a structure-preserving mapping between categories. A functor F from category $\mathcal{A}$ to category $\mathcal{B}$:

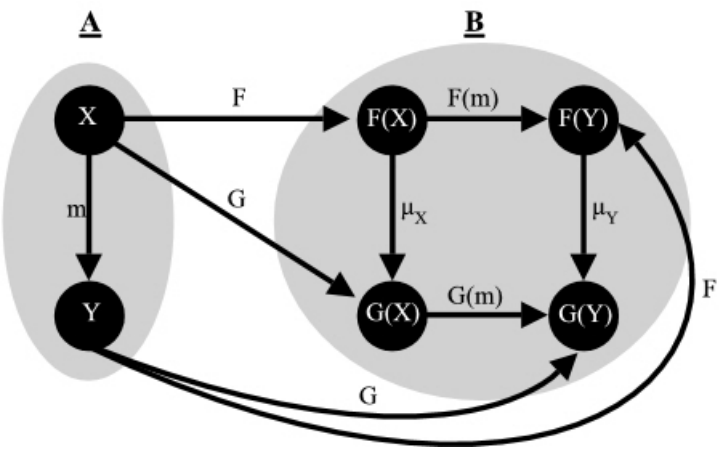
- Associates every object $X \in \text{ob}(\mathcal{A})$ an object $F(X) \in \text{ob}(\mathcal{B})$
- Associates every morphism $m : X \to Y \in \hom(\mathcal{A})$ a morphism $F(m) : F(X) \to F(Y) \in \hom(\mathcal{B})$, preserving identity morphisms and composition of morphisms. That is, explicitly:
  - $F(id_X) = id_{F(X)}$
  - $F(g \circ f) = F(g) \circ F(f) \quad \forall f, g \in \hom(\mathcal{A}) \;\; s.t. \;\; f : X \to Y$, $g : Y \to Z$

A functor from one category to another thus describes a way to symbolically encode one category into another. In fact, a functor between two fixed categories is also a mathematical object on which we can define a sensible structure-preserving morphism.

Definition 3. A **natural transformation** is a mapping between functors such that if $F : \mathcal{A} \to \mathcal{B}$ and $G : \mathcal{A} \to \mathcal{B}$ are functors between the categories $\mathcal{A}$ and $\mathcal{B}$, then the natural transformation $\mu : F \to G$ takes the form of a collection of morphisms μ from $\hom(\mathcal{B})$ satisfying:

- Every object X in $\text{ob}(\mathcal{A})$ has a morphism μ_X between objects of B satisfying $\mu_X : F(X) \to G(X)$ where μ_X is called the **component** of the natural transformation μ at the object X.
- (naturality) Given a morphism $m : X \to Y$ in $\hom(\mathcal{A})$ then the components μ_X, μ_Y must be such that $\mu_Y \circ F(m) = G(m) \circ \mu_X$

Graphically, naturality looks like this:



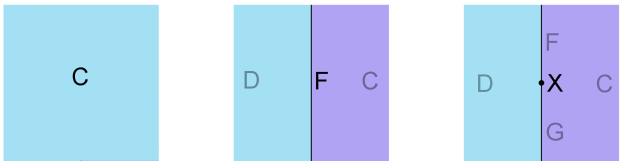
Naturality: the square formed in $\mathcal{B}$ commutes.

Functors are especially useful when we want a mathematical object with some structure whose elements we do not particularly care about (commonly known as a polymorphic object). We can define a functor F : `SET` $\to$ `SET` that maps an arbitrary collection of elements into a set representative of the structure we want to achieve. To give an example of this within computer science take the functor `LIST` : $A \to$ `NIL|A CONS (LIST A)`. This grammar defines a set, i.e. take $A = \{1, 2\}$ then the grammar defines {`Nil`, `1 Cons Nil`, `1 Cons (2 Cons Nil), ...`}.

Although typically in these cases `SET` would be replaced with the category of types in some given programming language. Natural transformations then easily take on the role of mappings between these structured data types, showing already the power of category theory and its applications in this area.

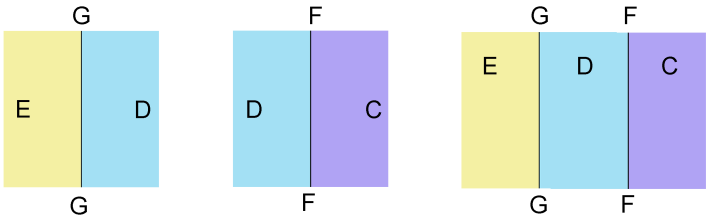
2.1 String Diagrams

As stated previously, proofs in category theory often become overly bloated and syntactical, frequently requiring a long chain of equivalent expressions. String diagrams aim to combat this by providing a clean and intuitive visual representation of categories, functors and natural transformations.
Based on applying the Poincaré Duality to the more traditional commutative diagram notation: string diagrams represent categories as planes, functors as lines (strings) and natural transformations as points.



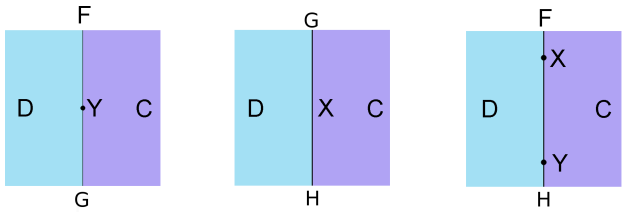
Left to right: a single category C, a functor $F : C \to D$, and a natural transformation X between two functors $F : C \to D$ and $G : C \to D$.

The diagrams neatly handle the composition of functors: we place the diagrams horizontally and read right to left.



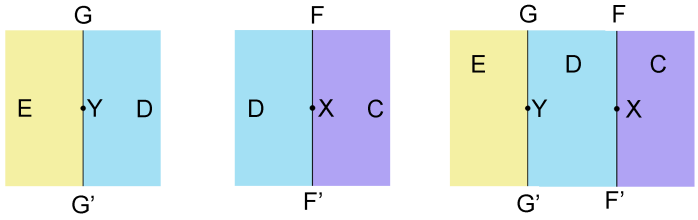
The diagram on the right shows the composition of the two on the left forming $G \circ F : C \to E$.

Composition of two natural transformations is done vertically which we read top to bottom.



Here we form $Y \cdot X : F \to H$.

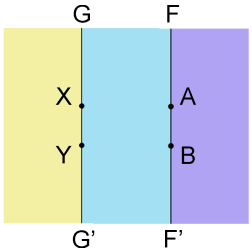
We can generalise the notion of horizontal composition to natural transformations.



We form $Y \circ X : G \circ F \to G' \circ F'$.

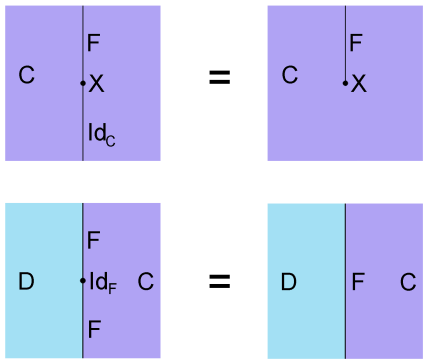
These two forms of composition are related by the Interchange law. This states that

$$(Y \cdot X) \circ (B \cdot A) \equiv (Y \circ B) \cdot (X \circ A)$$



This diagram embeds the interchange law.

The identity functor is never drawn, as it would only serve to clutter the diagram and would have no effect - you could arbitrarily draw an identity functor anywhere on the diagram. The same is true of the identity natural transformation.



Here Id_C is the identity functor in the category C and Id_F is the identity natural transformation on the functor F.

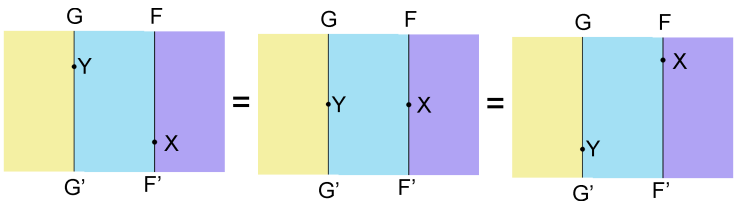
This means that the string diagrams silently bookkeep the axioms

$$id_F \circ F = F = F \circ id_F$$

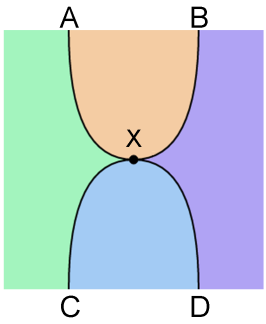
for any functor F and for any natural transformation η

$$id_\eta \cdot \eta = \eta = \eta \cdot id_\eta$$

We will now define the sliding equations which follow from the interchange law and the invisibility of identity natural transformations. These equations mean we can move any natural transformation vertically. By this point we often omit explicit labelling of categories to reduce clutter in the diagrams and rely primarily on colour to distinguish them.



Instead of drawing simple diagrams as before, the natural transformation $X : A \circ B \to C \circ D$ can be drawn as follows.



This allows access to each of the individual functors for further manipulations as well as a far more intuitive view of the types of the involved arrows that would normally be lost in the written notation [2].

2.2 Functional Programming and Haskell

More practically, category theory can be used to reason about various aspects of functional programing languages. In particular we discuss Haskell. We introduce the category `HASK`, the category of Haskell types. Here objects are Haskell types and morphisms are Haskell functions. We consider two morphisms to be equal if the functions are equal, that is if `f::A -> B` and `g::A -> B` and `f x = g x` for all `x` $\in$ `A`.

The identity morphism is the identity function `id`, and the composition of morphisms can be taken as the composition of functions `f.g` [8].

In practice, `HASK` is not a well formed category due to the existence of `undefined` but when reasoning about Haskell we generally ignore it [4].

Definition 4. An **endofunctor** is a functor whose source and target categories are equal.

From here on, we will predominantly be discussing endofunctors in the category `HASK`, and investigating the class of objects in haskell they represent. The typeclass `Functor` is available in the Haskell prelude, it defines one function `fmap :: (a -> b) -> f a -> f b` which should obey the following laws:

```
fmap id = id
```

```
fmap (p.q) = (fmap p) . (fmap q)
```

So `fmap` takes `id` to itself and is propagated through function composition [7].

We now show that if a datatype `F` instances `Functor` then the combination of `F` on types and `fmap` on functions satisfies the definition of an endofunctor in `HASK` by category theory.
We construct the endofunctor E from `F` and `fmap`. If `a` is a type then `F a` is the object mapped to by E. If f is a morphism, i.e. a function of type `(a -> b)` then `fmap f` generates a function of type `(F a -> F b)`. Finally we see that the laws above guaranteeing preservation of identity, and the preservation of morphism composition are exactly as stated in definition 2. □

We give the following example of a natural transformation in Haskell. Consider the definition

```
safeHead :: forall a. [a] -> Maybe a
safeHead (x : _) = Just x
safeHead []      = Nothing
```

(The type variable `a` is introduced with an explicit forall to emphasise the fact that the choice of `a` does not matter.)

If we have two functors, Φ and Ψ between categories C and D, the first condition of a natural transformation η from Φ to Ψ is to associate each object in C a morphism in D. For our example we have two endofunctors in `HASK`, `List` and `Maybe`. This means that to associate every object with a morphism is to give every type a function.
The second condition requires naturality, i.e. the following diagram should commute, where f is a morphism in D.
So in our case, if we concrete our example to `safeHead Int` and consider a function on integers, say `f = isPrime`, the unary function that returns `True` iff its input is a prime number. Then f is represented by `fmap f`. Our diagram thus requires the follwing two expressions to be equal.

```
(fmap isPrime) . safeHead

safeHead . (fmap isPrime)
```

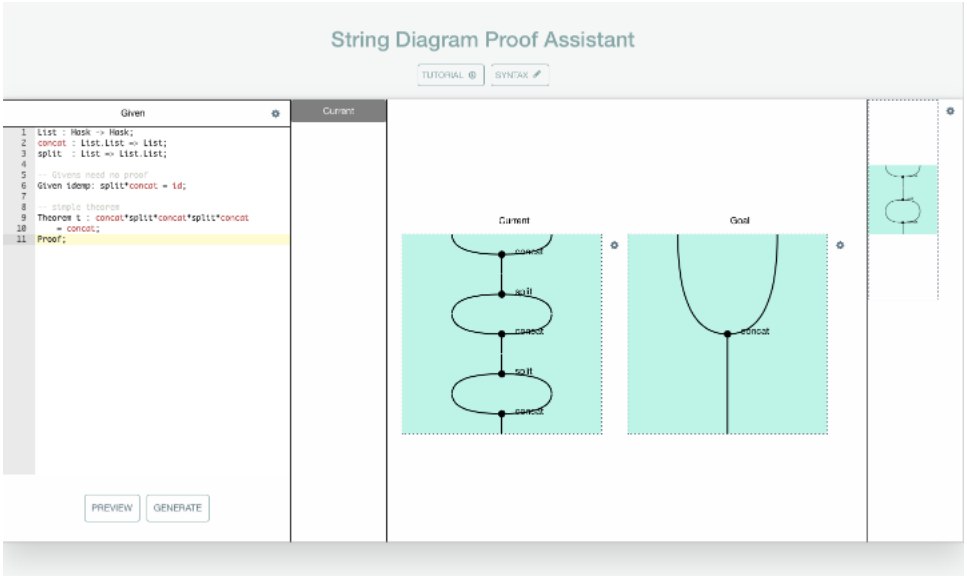
So the first ordering takes the head of a list into a `Maybe` and either applies the function to it or leaves it as `Nothing`. The second ordering applies the

function to every element of a list (possibly none) and then extracts the head into a `Maybe`. These give the same result. This means that `safeHead` is indeed a natural transformation. In the usual notation, it would be written as `[] => Maybe`, as it is a morphism between the List functor and the Maybe functor.

3 Usage

When using *Stringent*, the user must initially define their givens and their goal, as well as any custom constructs they may wish to utilize during their proof. In order to transfer these expressions unambiguously to *Stringent* they are required to adhere to the language specification - the user is provided with a reference guide for this specification through a button at the top of the screen. This quick-reference allows users to adapt to the language as quickly as possible.

Despite this, the new user will inevitably make mistakes, and *Stringent* is forgiving in this case: if the parser cannot process the input, a helpful error message is displayed detailing the mistake as well as its location in the input. Inputting text is made significantly easier due to the input field having the features of a modern text editor. This includes multi-line editing and syntax highlighting.



Once text is inputted and the parser has checked its validity, the user is then allowed to start their proof. As the user progresses by manipulating the displayed diagram, they may define sub-goals in order to segment the path to their objective.

When starting a sub-proof, the current diagram is generated by the left-hand side of the equation. The user aims to transform this diagram into the current target specified by the sub-goal (the right-hand side of the equation). Recommended steps for doing so are suggested by *Stringent* along the way. The user manipulates the diagram by selecting valid transformations via applications of rewrite rules which are themselves previously proven sub-goals or axioms.

Due to the nature of the tool, it is possible a first time user may be overwhelmed by the interface. To overcome this we have written a tutorial which carefully explains each of the components. Further, while several basic construct shapes are provided, it is possible for a more advanced user to upload their own custom shapes. This is provided via a menu option and so should not distract the new user from becoming familiar with the basics.

Proofs are often an exploratory process, the user will undoubtedly reach their goals through intuition and trial and error. This will lead to cases where the user has to backtrack to a previous point. This is facilitated via a history system where the user can access any step in the current sub-proof. If the current diagram exactly matches the inputted goal then *Stringent* will notify the user and the manipulation process will terminate. Once this has occurred the user can be certain that they have obtained a valid proof of the equality they were attempting to prove, by the soundness of string diagrams.

After a proof is completed the user will often want to include the proof in some written article or piece of work. To allow this we let the user download the images corresponding to any of the steps taken towards their goal in several widely supported formats such as PNG and SVG.

4 Design and Implementation

We aimed to design our application with three considerations in mind:

- Automation - *Stringent* should minimise the work involved in building proofs with string diagrams.
- Interactivity - *Stringent* should be accessible and easy to use.

- Correctness - *Stringent* should not be able to generate invalid proofs.

4.1 Technologies

We needed a target platform to write the user interface for *Stringent* and a web application seemed like the obvious choice. A webapp allows almost global compatibility with few issues as well as very easy distribution. Compared to a GTK binary, or the result of another graphical framework where these concerns would have been an issue, *Stringent* can be hosted on a web server and accessed by anyone.

These libraries work in the browser's native language, Javascript, but due to the nature of our project we had a back end that would be most naturally written in the purely functional Haskell using GHCJS to compile to Javascript. We could have chosen to write some parts in Haskell and others in Javascript but instead we chose to write as much as possible in Haskell leaving only a handlful of lines of bridging Javascript. This gave us a cohesive project to work on, as well as some unique benefits that Haskell brings over Javascript:

- Strong, static typing, eliminating the majority of runtime errors
- Referential transparency which makes programs far easier to reason about
- Expressiveness at both the type and language levels. For the language, high levels of abstraction mean that you can avoid repeating yourself, and for types you gain statically confirmed information, which enables compile time guarantees.

Unfortunately this choice did bring about some negatives: the GHCJS compile times were quite high and the resultant Javascript files were bloated from the GHC runtime. It also required the team to work in Haskell, which we were not all familiar with, leading to stunted development during the first few weeks.

Some alternatives in the same vein that we considered were Elm and Purescript. Elm is a pure language with strong static typing, making it an obvious candidate as the language of choice for our project. At first glance, it has many benefits:

- It is a simpler language, making it easier to teach to the less experienced members of the group.
- It natively compiles to Javascript, and is therefore designed to work in this ecosystem.
- We have experience using the language in prior work, leading to less time spent learning.

However, our experience with the language ultimately informed us against using it for one main reason; the type system lacks the sufficient abstractions necessary for modern development, leading to repetition and boilerplate code.

Purescript has many of the benefits of Haskell, with a strong and expressive type system as well as purity. It also shares a benefit with Elm: it is designed from the ground up for the web ecosystem. We decided against using it due to our lack of familiarity with the language, and the lack of tooling surrounding it compared to Haskell.

In order to layout our webapp we chose to use Flux and React. Flux neatly organises the flow of data to effectively encapsulate various parts of the project and is widely used, supported by bindings to GHCJS.

Finally, to render the diagrams themselves we chose the Scalable Vector Graphic (SVG) format. Vector images are the obvious choice for the diagrams as they are created by the composition of geometric objects and there is no reason to rasterize them. HTML5 has built in support for SVG elements which made it incredibly easy to display the results of our program. The format also supports paths which allow us to draw the characteristic bezier curves of the diagrams. SVG is also widely supported with image viewers, word processors and markdown languages. With only a few minor changes in how we generated the vector files it was possible to let users download them and view them locally. This allows users to easily include them in any publication they may be writing.

As an alternative we considered rendering directly to an HTML5 canvas. This had the advantages of speed and greater customizability but would have required writing a more extensive set of functions to deal with rendering. In

the end we did not find speed an issue outside abnormally large diagrams and did not have a need for the flexibility of writing individual pixels.

4.2 Parsing

Stringent begins with user input through text, as text is a universal data format. In order to gain structure and meaning from this, the text must first be parsed into an internal data structure. For common data structures, conventions exist for deciding the rules of the language (such as JSON or XML for attribute-value stores), however for such a customised data structure as string diagrams, no such conventions exist. Therefore we must first define a language for describing string diagrams which can be used for parsing.

We settled on the following language grammar specification for the core calculus:

`F : A -> B`	`F` is a functor from category `A` to category `B`.
`n : F => G`	`n` is a natural transformation from functor `F` to functor `G`.
`(n.m)`	The horizontal composition of natural transformations `n` and `m`
`(n*m)`	The vertical composition of natural transformations `n` and `m`.

For ease of use, we remove the need to place unnecessary brackets, since both composition operators are associative, we give vertical composition precedence over horizontal composition. The result is that `(a*b.c.d*e)` is equivalent to `(((a*b).c).(d*e))`.

To parse from this language specification, we use the approach of parser combinators[3]. The concept of parser combinators are functions which accept parsers as arguments and return more complex parsers as a result. For instance, the combinator 'many' will accept a single parser and return one which tries to parse using the original as many times as it can before failing. This could be used to transform the parser which accepts a single character into one which accepts a string. There are a large number of combinators, which express common patterns in languages, and these can be combined to form a complex parser.

An example parser from our code is:

```
functor = do
  c1 <- expr
  spaces
  string "->"
  spaces
  c2 <- expr
  return (Arrow c1 c2)
```

- `expr` is a parser which returns an expression.
- `spaces` is a parser which consumes spaces from the string. As shown above, `spaces` does not return anything which is used in the output, indicating that whitespace is ignored in our grammar.
- `string` is a function which takes a string and returns a parser which matches that string exactly.
- The resulting parser is one which matches the grammar: `A->B`, where A and B are expressions. This is exactly as we defined it in our language grammar earlier.
- The combinator style in Haskell makes use of monadic do-notation, allowing the code to be read in a straightforward manner. The implicit bind operations used in do-notation are combinators themselves, meaning that the whole thing ends up representing a single parser, despite reading like imperative code. Overall the resulting style of parsing is easy to read as well as easy to write, while being expressive enough to define many complex grammars.

We used the `parsec` library which provides many combinators for us, as it is the industry standard library for parsing in Haskell. Alternatively we could have written our own functionality or used a different library, but neither approach is justifiable given the breadth, reliability and speed of `parsec`.

4.3 DSL

At the heart of our proof assistant lies a domain specific language (DSL) that can be used to describe expressions and proofs thereof in our categorical

setting. In order to be able to render the expressions as string diagrams, our core language closely models the domain, while it is expressive enough so that arbitrary propositions can be proven inside it.

In our language, we distinguish between:

- **Categories**
  `Category`
  is the type of categories

- **Functors**
  $(a : \texttt{Category}) \to (b : \texttt{Category}) : \texttt{Functor}$
  is a functor from category a to category b.

- **Natural transformations**
  $\Sigma(x : \texttt{Functor}).(a : x) \Rightarrow (b : x) : \texttt{NaturalTransformation}$
  is a natural transformation from functor a to functor b, which are themselves functors of type x.

  $\Sigma x : A.B$ is a called a dependent sum: it's the type of pairs with first element of type A an the second element of type B, which may depend on the value x, as seen in the case of *natural transformations*: the underlying functor type of a natural transformation is important when composing natural transformations.

  As we can see, terms can appear in types, and vice versa, which means that in fact types and terms are the same. Because of this, the type system is said to be dependent.

- **Constructors**
  Sometimes it can be useful to talk about constructions based on other, existing ones. For instance, given an endofunctor f, we can express the notion of the free monad generated by f. Constructors are special (injective) functions that can be used to create and reason about user-defined constructions. For instance, given an endofunctor F on some category, one can talk about the free monad generated by F, `Free`F. We can then write and prove theorems about `Free`F.

```
Constructor Free : forall (f : a -> a), Free f : a -> a
```

- **Propositions**
  `Prop` is the type of propositions, i.e. properties that we aim to prove. If $q : p : \texttt{Prop}$, then p is called a proposition, and q a proof of the proposition, or a proof object. $(A : t) = (B : t) : \texttt{Prop}$ says that A and B are equal.

- **Proofs**
  Given an equality proposition $A = B$, we can rewrite an expression containing A by replacing an occurrence of A by B. Equality is symmetric, so $B = A \to A = B$. When proving equalities, the goal is to end up at a proposition (through rewriting) that reads $A = A$, which can be proven trivially, since equality is reflexive.

- **Type classes**
  For many theorems, the concrete structure we are proving them for does not matter, only certain properties of the structure, meaning that we can prove the theorem for all structures that admit these properties. This notion of ad-hoc polymorphism has been widely used in Haskell, via typeclasses [5]. Adopting this construction allows us, for example, to prove theorems about all monads m and n with a distributive law $(n.m \to m.n)$, instead of having to prove the theorem individually for all pairs of distributable monads of our interest.

  The following monad typeclass describes the natural transformations `return` and `join` that define a monad, along with the coherence conditions that need to be satisfied by all monads.

```
class Monad (M : forall a : Category, a -> a):
  return  : Id => M;
  join    : M . M => M;
  monad_1 : join * (join . M) = join * (M . join);
  monad_2 : join * (M . return)  =  join * (return . M);
  monad_3 : join * (M . return)  =  Id
end.
```

  The laws can be used to rewrite expressions in proofs. It is also possible to prove for a specific data type T that the triple (T, μ, η) forms a monad by showing that the coherence conditions hold (`join` $= \mu$ and `return` $= \eta$).

4.4 Type system

One great advantage of using Haskell as the implementation language is that we were able to benefit from its very strong static type system. Our design philosophy has been making illegal states unrepresentable. In many cases, we took advantage of modern GHC type-system extensions.

For example, in order to represent the expression language, one might naively write:

```
data Expr
  | Name String
  | Funct Expr Expr
  | NatTrans Expr Expr Expr
  | Vertical Expr Expr
  | Horizontal Expr Expr
```

For example, the expression $G.F \Rightarrow F.G : X \to Y$ is represented as

```
nt1
  = NatTrans
      (Funct (Name "X") (Name "Y"))
      (Horizontal (Name "G") (Name "F"))
      (Horizontal (Name "F") (Name "G"))
```

But bogus expressions are now also representable:

```
nt2
  = NatTrans
      (Funct (Name "X") (Name "Y"))
      (NatTrans (Name "Z") (Name "A") (Name "B"))
```

which does not make sense.

By lifting our domain-specific knowledge to the type-level, we can rule out invalid constructions, as such:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeInType #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE ExplicitForAll #-}

import GHC.TypeLits
import Data.Proxy
```

```
data T = Type | Term

data Prop = Prop

data Cat where
  Cat :: Symbol -> Cat

data Arr where
  NatT :: Symbol -> Symbol -> Arr
  Fun  :: Symbol -> Symbol -> Arr

data Expr (t :: T) o (k :: o) where
  Name
    :: forall o (t :: o) (s :: Symbol).
    Proxy s
    -> Expr 'Term o t
  Funct
    :: Expr 'Type Arr ('Fun c1 c2)
  NatTrans
    :: Expr 'Term Arr ('Fun c1 c2)
    -> Expr 'Term Arr ('Fun c1 c2)
    -> Expr 'Type Arr ('NatT c1 c2)
  Equals
    :: Expr 'Term o t
    -> Expr 'Term o t
    -> Expr Type Prop 'Prop
  Horizontal
    :: Expr 'Term Arr a
    -> Expr 'Term Arr a
    -> Expr 'Term Arr a
```

Haskell's type-level is dependently typed (`TypeInType` [6]), which makes it a very good platform for embedding a dependently typed language in. One minor difficulty is that there is a clear phase distinction between type-level computation and value-level computation, and extra effort is required to make use of this type-level hackery on dynamic input.

The singletons package[1] can be used to produce value-level witnesses of the type-level information, so even the user-input can be verified this way, the details of which we gloss over in this report.

Some examples that do compile:

```
good1 :: Expr 'Type Arr ('NatT "X" "Y")
```

```
good1
  = NatTrans
      (Horizontal (Name (Proxy @"G")) (Name (Proxy @"F")))
      (Horizontal (Name (Proxy @"F")) (Name (Proxy @"G")))
```

```
good2 :: Expr 'Type Arr ('Fun "X" "Y")
good2
  = Funct @"X" @"Y"
```

The following does not compile, because the term's codomain does not match that of its type.

```
bad1 :: Expr 'Type Arr ('Fun "X" "Y")
bad1
  = Funct @"X" @"Z"
```

Unfortunately the above formulation can still represent invalid states. If we keep pushing more information to the type-level, we can reach a surprisingly compact representation, which keeps track of the composition rules and the names of the functors as well:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeInType #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE ExplicitForAll #-}
{-# LANGUAGE TypeOperators #-}

import GHC.TypeLits
import Data.Proxy

data Cat where
  Cat :: Symbol -> Cat

-- Representing functors
data a ~> b where
  -- a named functor from a to b. It is only to be used at
     the type-level
  -- through DataKinds, as Symbol has no value-level
     inhabitants
  F
    :: Symbol
    -> a ~> b
  -- composition of functors
  (:.)
```

```
    :: (a ~> b)
    -> (b ~> c)
    -> (a ~> c)

-- Natural transformations, indexed by the underlying functor
   's type, k.
-- This is made possible by TypeInType
data N k (f :: k) (g :: k) where
  -- this implicitly ensures that both ‘f‘ and ‘g‘ are of
     type ‘a ~> b‘.
  N
    :: String
    -> N (a ~> b) f g
  -- vertical composition keeps the underlying functor
  (:*)
    :: N k f g
    -> N k g h
    -> N k f h
  -- horizontal composition composes the domain and codomain
     functors
  -- notice how types are kept track of at construction
  (:..)
    :: N (a ~> b) f g
    -> N (b ~> c) h j
    -> N (a ~> c) (f :. h) (g :. j)

-- f : F => G.H : A -> B
n1 :: N ("A" ~> "B") (F "F") (F "G" :. F "H")
n1 = N "f"

-- g : J => K.L : B -> C
n2 :: N ("B" ~> "C") (F "J") (F "K" :. F "L")
n2 = N "g"

-- f.g : F.J => G.H.K.L
-- n3 :: N ("A" ~> "C") (F "F" :. F "J") ((F "G" :. F "H") :.
    (F "K" :. F "L"))
n3 = n1 :.. n2
```

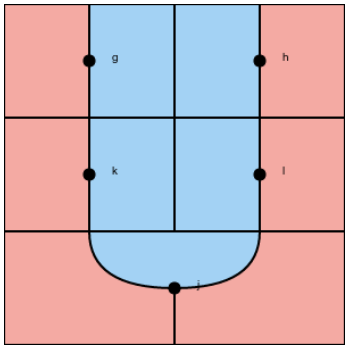
Interestingly, with the latest version of GHC (as of writing) `n3` only compiles with its type signature omitted, even if we copy paste back the inferred type from GHCi. This, and another bug we found in GHC stopped us from using this formulation. The advantage of this embedding is that existing type-checking facilities can be leveraged.

4.5 Rewriting

Rewriting can be implemented in a fairly straightforward way: given an equality proposition $\forall x, e1 = e2$, we check if any of the sub-expressions can be unified with $e1$ (or similarly, $e2$). If $e1$ is monomorphic, i.e. x does not appear free in $e1$, then the unification algorithm will look for exact matches, otherwise it will try to instantiate $e1$, and use the resulting substitution to instantiate $e2$, then replace the instance of $e1$ with that of $e2$.

4.6 Rendering

In order to more easily draw our expressions, we first create an intermediate format (`Diagram`) which can be drawn more easily. This type should contain exactly the information required to render it correctly, stripping irrelevant parts. Whereas the original type was homogeneous in how it treats functors, natural transformations and categories, `Diagram` makes a special case for unit natural transformations as building blocks for the diagram. We refer to these unit natural transformations as "nodes".



A diagram broken into its constituent nodes

```
data Diagram
    = Unit Node
    | HorizontalComposition Diagram Diagram
    | VerticalComposition Diagram Diagram
```

Here `Node` is defined separately and annotation type arguments have been omitted for brevity.

The first step in drawing the nodes to the screen is assigning coordinate parameters to each node. By doing this we reduce the problem of drawing the

entire diagram to the problem of drawing a single node. Once this has been done a node can be exported to SVG and this can be repeated across the entire diagram.

The process of assigning coordinates is done recursively down the structure of `Diagram`. We decide arbitrarily that the nodes align into rows (as opposed to columns). When converting to this format we ensure this by inserting dummy nodes which have no effect on the semantics of the diagram. Once this has been established the problem of assigning y-coordinates to each node is trivial (simply divide the space into equal ranges). Now the problem remains of assigning x-coordinates to each node, which we do via a stateful traversal in a "divide and conquer" manner.

There are three SVG components when drawing a node, the categories in the background, the strings, and the details of the node on top. The first thing we calculate are the paths of the strings which can determined by the input and output functors of the node. These paths are used to define the outline of the category regions. The categories are assigned a colour and drawn first at the bottom with the strings drawn on top as black lines. Finally the detail information such as the name of the node is drawn last.

The above algorithm is enough to gain a rough approximation for drawing digrams, however there are special cases to consider. Firstly we do not want to draw the identity functors and natural transformations. This can be achieved by a simple modification to the node-drawing code. Additionally the diagram can contain user defined constructs, visually manifesting as shapes on top of the diagram. This cannot be achieved at the individual node level, and must be calculated during the traversal.

5 Evaluation

It makes sense to evaluate our final result with considerations to our initial goals.

Automation

A goal of our project was to reduce the manual work surrounding string diagrams. We completely automated the process of drawing a string diagram, requiring only minimal textual input and clicking to create all the diagrams for a proof. The system is able to handle complex diagrams with custom notation, enabling our tool to be more than simply a toy.

We also automated the process of creating a proof, by suggesting potential transformations and handling sub-goals and history. This massively reduces the mental overhead when creating a proof, as we handle the inherent problem of bookkeeping.

Overall *Stringent* succeeded as a means to automate the process of proof creation, making string diagrams a more viable notation for proving theorems in category theory.

Interactivity

Stringent presents an intuitive user interface, and has shown to be easy to use among our target audience. One might argue that a user will be turned away from the application when greeted with a text input box. However for our target users, this input method is not only commonplace, but essential given the need for precision and reproducibility.

However, string diagrams are a visual notation. Continuing the dependency on text would mean our application fails in terms of its goal of interactivity. After the initial input stage, manipulation is done via clicking on large and clearly labelled buttons.

Early plans mentioned fluid visual manipulation of diagrams, helping for stylisation when exporting. While this is still a valid goal of the project, when weighed against the other goals of the project it could not be prioritised.

Overall *Stringent* is an interactive and accessible proof assistant, with room for improvements in fluidity.

Correctness

Throughout the development process, careful attention was given to maximising our confidence in the correctness of the implementation. To this end, we made use of Haskell's expressive type system to embed information we wanted statically confirmed.

We also made use of an extensive test suite, giving guarantees about properties of our codebase. This is explained in more depth in the next section.

Another aspect of our design which gave confidence was sufficient decoupling between sections, effectively isolating the critical code from less tested parts.

We are therefore highly confident in *Stringent*, as a tool for generating correct string diagram proofs, as each step of our development process was directed with this in mind.

5.1 Testing

User interface testing tends to be very brittle and complicated, and due to our fast iteration cycles on our UI, we felt that continually rewriting tests would create more work than the benefits formal testing would bring. Therefore, the UI remained largely formally untested during development, and testing was mainly done through informal user testing.

Because of the fast moving development, the user interface remained largely untested. Our mindset being that continually rewriting tests as the structure changed would create more work than the tests themselves would save.

The typechecker however was rigorously tested with the `HSpec` and `QuickCheck` Haskell testing frameworks, which proved to be extremely useful, given that the typechecking algorithm was really frequently tweaked, and subtle bugs would often emerge that were caught by the test suite in time. `QuickCheck` was used for testing the unification algorithm: it is a property based testing framework, which differs from traditional unit testing frameworks in that instead of requiring manually defined instances to check against we specify a property and generate random values to test against. This mechanism requires a way to generate random examples of a given type, and a property

expressed as a function that returns a boolean. Our unification algorithm was tested as follows:

1. Generate two arbitrary types, $t1$ and $t2$. Care was taken that these are likely to be somewhat similar, so that there is a realistic chance that the two generated instances can be unified.

2. Unify them

3. If the unification fails, we move on, this is not the interesting case.
   In case of a successful unification, we obtain a substitution s. The property we are testing is that the generated substitution is valid, i.e. that

   ```
   substitute s t1 == substitute s t2
   ```

This is done 100 times on each test cycle.

The most important systems to test were those in the core of the proof assistant. It is important that the parsing and typing systems are robust, since they directly model the theory behind the diagrams. Therefore testing was prioritised onto these components.

References

[1] Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.

[2] Ralf Hinze and Dan Marsden. Equational reasoning with lollipops, forks, cups, caps, snakes, and speedometers. *Journal of Logical and Algebraic Methods in Programming*, 85(5, Part 2):931 – 951, 2016.

[3] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, University of Nottingham, 1996.

[4] Patrik Jansson Nils Anders Danielsson, John Hughes. Fast and loose reasoning is morally correct. *ACM SIGPLAN Notices*, 41:206–217, 2006.

[5] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[6] Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System fc with explicit kind equality. In *ACM SIGPLAN Notices*, volume 48, pages 275–286. ACM, 2013.

[7] Haskell Wiki. Functor. `https://wiki.haskell.org/Category_theory/Functor`. Accessed: 2017-01-01.

[8] Haskell Wiki. Hask category. `https://wiki.haskell.org/Hask`. Accessed: 2017-01-01.