School of Computer Science, University of Nottingham

Toward a Formalization of ’Virtual Sets’

Christina O’Donnell

September 2025

Acknowledgements

I would like to thank Thorsten Altenkirch for helping direct this project and for his invaluable feedback. I also must thank Mark Williams for sharing his idea with me that motivated dissertation.

Delcaration

Submitted 21st September 2025, in partial fulfilment of the condiations for the award of the degree of MSc (Computer Science).

Abstract

This dissertation investigates the categorical structure underlying the monoidal category of finite sets and injective functions, with the goal of understanding whether it admits a well-defined trace operator. A trace is a categorical generalization of feedback structures such as recursion, or feedback, introduced by Joyal et. al. (1996), traces can be applied to perform the role of recursion in categorical semantics of programming langauges, and the role of feedback in quantum computing. Mark Williams (2025) recently conjectured that the category of finite sets and injective functions may have a sufficient trace operator, however the construction has yet to be formally verified.

The notion of 'virtual set', introduced by Williams (2025) is a set that can contain 'negative elements', satisfying the condition for a compact closed category. This construction amounts to taking a quotient of Inj^2, and then proving coherence of the result, as spelled out in Joyal et. al. (1996). The difficulty is to prove all of the coherence lemmas required to show it has monoidal, and trace structure, which is independent of equivalent pairs of functions. Such a structure is of interest in simplicial type theory, where all inclusion map between simplicial complexes must necessarily be injective.

This dissertation proceeds by developing the category Inj of finite sets and injective functions, constructing a composition operator and proving associativity and unit laws. A tensor product is then defined as well as a partial proof of the monoidal laws. We then define a trace operator and prove some of the required properties about its behavior. Finally, the Int construction is discussed.

All definitions and proofs are implemented in cubical Agda, making this one of the first attempts to formalize the notion of traced monoidal categories. We discuss my two main approaches at formalizing the monoidal category of injective functions, and constructing the tensor operator, and trace, as well as proving some of the properties about it.

Keywords: Trace Monoidal Categories, Compact Closed Categories, Cubical Agda

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Scope

The motivation for this project is mainly exploritory. It was observed by a preprint by Mark Williams (2025) [24] that the category of finite sets and injective function (Inj) admits a trace operator, as introduced in Joyal-Steet-Verity (1996) [12] . A trace operator acts like a subtraction on morphisms, and in the case of finite sets, it literally removes elements from the map, decreasing the size of the domain and codomain by the same amount.

The intuition is then that if we have a tensor that acts as an addition, and a trace acting as a subtract operator, then can we construct an extension that is closed under traced using these operations? To put another way, can we construct a category in which every map has an inverse. It was proven by Joyal et. al. (1996) [12] that a category with a trace satisfying certain coherence conditions in section **??** can be used to construct a category closed under trace, called a compact closed category, in which the original category forms a sub-category. In other words, it forms an extension to the original category, that is closed under trace.

Although traced monoidal categories have been expressed in the Agda categories library [10] the Int-construction (as defined in Joyal et. al. 1996 [12]) has not been formalized to date in any major proof assistant (Agda [23], Coq [21], Isabell/HOL [18], LEAN Mathlib [17]). The original plan was to implement it here, however it was found after some time developing this, that the process was more complex that initially expected.

1.2 Informal Problem Statement

At a high level, this dissertation investigates whether an injective function on finite sets can be operated on by a certain 'trace' operator, satisfies the Joyal-Street-Verity axioms [12] for a traced monoidal category, and hence a compact closed category, using the Int-construction.

1.3 Research Objectives

The aim of this project is to build a up a definition of the category of 'virtual sets' by constructing a category of injective functions, defining a tensor and trace opoerator, and proving that the coherence conditions hold, and finally to use the Int-construction to build a compact closed category, which is a symmetric monoidal category in which every object has a dual.

We list seven objectives, each of which building of the previous objective:

Objective 1 (Define the 'precategory' Inj)**.** Build the precategory [16] Inj:

objects Define the type finite sets

morphism Formally define injective functions

identity Define an identity $id \ : \ A \to A$ morphism for every object.

composition Define a composition operator $\circ$ on morphisms:

$$g \circ f \text{ is an arrow from } A \text{ to } C \text{ where } f : A \to B \text{ and } g : B \to C$$

Objective 2 (Show that Inj forms a (full) category)**.** Show that the precategory defined in objective 1 forms a (full) category [3]. This requires the following:

associativity show that for suitable morphism types f, g, h that both ways of composing results in the same morphism:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

$$\text{where } f : A \to B \qquad g : B \to Ch : C \to D \qquad A, B, C, D \text{ objects}$$

unit

$$id \circ f = f \text{ and,}$$
$$f \circ id = f$$

$$\text{where } f : A \to B$$

Build the category Inj, with a suitable composition function $\circ$ and identity element id. Construct a proof of associativity of and left and right unitor laws (cf. common texts include MacLane 1998 and Awodey 2010 [3, 16]). Finally show that the type *Inj* is set-like in the homotopy sense (see definition 13).

Objective 3 (Define a tensor operator)**.** Define a tensor operator $\oplus$ and unit $\mathbb{0}$ on Inj. That is a *bifunctor* F from Inj^2 to Inj (see definition 2) and an object selected to act as the unit. It must be checked that the functor F preserve the following operations:

dom/cod Given a morphisms f, then,
the domain $\mathrm{dom}(Ff) = F\mathrm{domf}$, and
the codomain $\mathrm{cod}(Ff) = F\mathrm{codf}$.
(NB. This is handled by the type system in Agda.)

identity A functor must map identity arrows to identity arrows.

composition F applied to the composition $g \circ f$ is equal to $Fg \circ Ff$ for arrows f, g.

Objective 4 (Construct a (weak) Monoidal Category)**.** Show that the tensor product defined above forms a monoidal category. This required the following natural isomorphisms:

associator $\alpha_{A,B,C} : A \oplus (B \oplus C) \cong (A \oplus B) \oplus C$.

left-unitor $\eta_A : \mathbb{0} \oplus A = A$

left-unitor $\rho_A : A \oplus \mathbb{0} = A$

Then show that the following two coherence laws hold:

'triangle' law when there are two ways to use unitors to to cancel the unit, then the two resulting morphisms are equal.

'pentagon' law When there are two different ways to convert between $(A \oplus (B \oplus C)) \oplus$ and $(A \oplus B) \oplus (C \oplus D)$, using the associator α, then these arrows must be equivalent.

Aside: It turns out that these two laws are sufficient to say that if there is any way to convert from one object to another using just the unitors, and associators, then it must be unique. (Mac Lane 1998 [16] in Ch VII § 2.)

Objective 5 (Define a trace operator on `Inj`)**.** Define a function $Tr_X : ((A \oplus X) \rightarrowtail (B \oplus X)) \to (A \rightarrowtail B)$. See definition 7.

Objective 6 (Check trace coherence laws)**.** The following laws must hold:

tightening See axiom 1.

sliding See axiom 2.

vanishing See axiom 3.

superimposing See axiom 4.

Objective 7 (Implement Int-construction)**.** Implement the construcion of a compact closed category [8] from a traced monoidal category. This involves reproducing the results from Joyal et. al. 1996 [12].

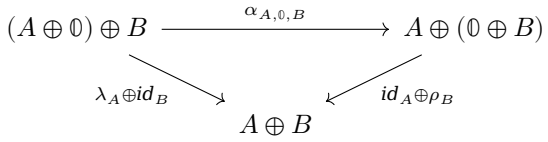


Figure 1.1: The triangle coherence law for a monoidal category $(\mathcal{C}, \oplus, \mathbb{0})$.

[3, 15]

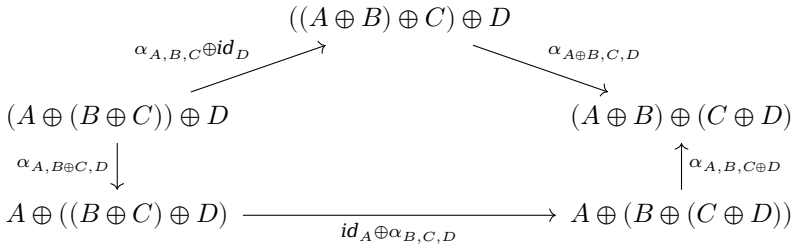


Figure 1.2: The pentagon coherence law for a monoidal category $(\mathcal{C}, \oplus, \mathbb{0})$. [3, 15]

1.4 Structure of the Dissertation

We begin this dissertation by defining some category-theoretic defintions we will be relying on in chapter **??**. In chapter **??**, we perfom a brief literature reiview of traced monoidal categories. In chapter **??**, we build up the formalization of injective functions up to defining a trace operator. Following on from there, in we construct the formalization of the structures in two streams, each starting from the defintion and properties on finite sets, building up to category theoretic definitions. The first stream builds injective functions as a dependent sum of a function and a proof that function is injective, and the second stream defines functions inductively, which are injective by construction. Finally, in chapter 4 we list the results, discuss further work, and limitations of this formalization effort.

1.5 Methodology

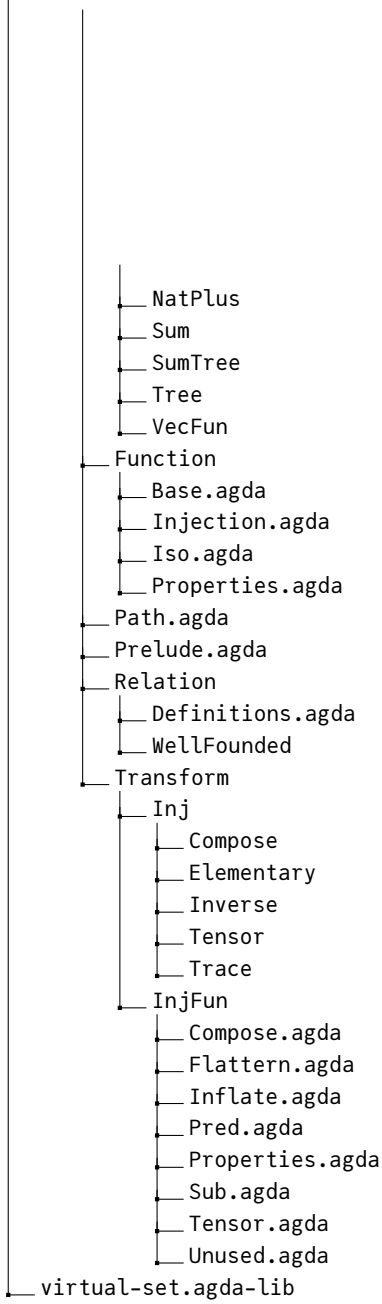
The majority of this work was carried out in a single Git [5] repository hosted on GitHub GitHub [19]. Is is implemented in cubical Agda [23] using the cubical Agda standard library [6]. Originally I started working in the Agda standard library [7] however it was suggested by Thorsten that I should switch to a cubical library, to avoid ”setoid hell” (cf. Altenkirch 2017 [2], more generally, Allais et. al 2025 [1]), suggesting 1Lab library [14]. I moved my definitions over to 1Lab but there were some akward definitions, which were slowing me down. Most notably the definition of finite sets (`Fin`), which used a quotient (truncation) type [11, 22]. I eventially settled on the standard cubical

library [6].

The work is fully reproducible for its Git repository [19], using Nix flakes [9], and include all of depdencies all of the lemmas I have completed. It even include the list of dependencies and python script for building the report. The code is structured modularly and designed intentionally for ease of further development.

The layout of the repository is as follows:

```
build.py
flake.lock
flake.nix
justfile
latex
    agda.sty
    chapters
        conclusion.tex
        formalization.tex
        introduction.tex
        lit-reivew.tex
        prelim.tex
    main.tex
    preamble.tex
    report.bib
libraries
nix
    args.nix
    packages.nix
    shells.nix
postprocess-latex.pl
src
    Compat
        1Lab
        1Lab.agda
    Dissertation
    DissertationTex
    Notes
    VSet
        All.agda
        Cat
            Inj.agda
            InjFun.agda
            Trace.agda
        Data
            Fin
            Fin.agda
            HITTree
            Inj
            InjFun
            Maybe.agda
            Nat
            Nat.agda
```



There are 3 main top-level directories for different purposes:

`./latex/` defines the parts of this dissertation that are written in *TeX* or an ajacent language, such as *BibTeX*. These are assembled by ‘./build.py‘ with the sections of the dissertation written in literate Agda (with markdown), to produce the final PDF file. Everything in `./src/Dissertation/` is converted into TeX with Pandoc, in order to process the embedded TeX commands, then Agda type checks the file, before adding syntax hylighting and references before inserting it into `./latex/generated/`, where it in turn is referred to by `./latex/chapters/formalization.tex`. This circuitous route was required as to date, Agda does not currently have a markdown-to-latex backed.

`./nix/` together with `./flake.nix` and `./flake.lock`.

Finally all of the Agda source is contained in `./src`. Compat contains functions that allow the use the functions of other libraries, such as 1Lab. These are not source I have developed myself, but they are functions copied into the repo to enable use of certain functions from other libraries. Normally libraries can’t be mixed, because it can result in mismatch of primitive definitions, meaning you

have to convert between these definitions manually every time you want to use a function from a foreign library. `./src/Dissertation/` and `./src/DissertationTex/` contins the files to generate the formalization chapter.

The bulk of the source is in `./src/VSet/`:

`./src/VSet/All.agda` is a list of all files to type check to ensure the repository is still valid. I run this regularly to ensure I haven't broken anything.

`./src/VSet/Cat/` : This contains category definitions and category constructions. Inj.agda and InjFun.agda are the two category constructions for the two ways to define injective maps explored in this work.

`./src/VSet/Data/` : This directory contains various definitions and properties of Data types. Most properties of data types are defined here, except the main development for Inj and InjFun is performed inside `./src/VSet/Transform`.

`agda`, and `./src/VSet/Path.agda` define basic properties of functions and paths, for use elsewhere.

`./src/VSet/Relation/` defines basic properties of functions and paths, for use elsewhere.

`./src/VSet/Transform/` defiens transformations of injective functions, such as inserting/removing links up to describing a trace function. This is where the majority of the development is situated.

Chapter 2

Preliminaries

prelim

2.1 Category Theory

2.1.1 Introduction

Category theory is the study of abstract structure generalizing concepts seen across many disciplines such as mathematics, computer science and physics. [3, 16]

Definition 1 (Category [3,20])**.** From the perspective of type theory, *category* consists of the following data:

- a type of objects $\mathsf{Obj} : \mathcal{U}$;
- for every $A, B : \mathsf{Obj}$, a type of morphisms $\mathsf{Hom}(A, B) : \mathcal{U}$;
- for every $A : \mathsf{Obj}$, an identity morphism $\mathsf{id}_A : \mathsf{Hom}(A, A)$;
- a composition operation

$$\circ : \prod_{A,B,C:\mathsf{Obj}} \mathsf{Hom}(B, C) \to \mathsf{Hom}(A, B) \to \mathsf{Hom}(A, C).$$

These are equipped with terms demonstrating that:

$$\mathsf{assoc}_{h,g,f} : (h \circ g) \circ f = h \circ (g \circ f), \quad \mathsf{idl}_f : \mathsf{id}_B \circ f = f, \quad \mathsf{idr}_f : f \circ \mathsf{id}_A = f,$$

for all $A, B, C, D : \mathsf{Obj}$ and $f : \mathsf{Hom}(A, B)$, $g : \mathsf{Hom}(B, C)$, $h : \mathsf{Hom}(C, D)$. Here $=$ denotes the identity type of the ambient type theory.

Classically objects are considered to be collection of objects with some structure, and morphisms considered structure preserving maps. However there is a wide array of interpretations are possible. To list a few:

Note 1. Categories are considered 'small' if their objects and morphisms can be represented as sets. This is done at the type level, ensuring that the category Cat is not contained in itself, to avoid contradictions.

Category	Objects	Morphisms
Set	Sets	Maps between sets
Ord	Total orders	Order preserving maps
Type [1]	Types	Computable functions between types
Grp	Groups	Group homomorphisms
Top	Topological spaces	Continuous maps
Cat	'small' categories	Functors
FinSet	Finite Sets	maps

Table 2.1: Examples of common categories. Standard definitions and names come from [3, 13, 16].

2.1.2 Functors

Functors are mappings between categories that preserve the categorical structure, allowing us to relate different categorical contexts. A functor F must map identity morphisms to identity morphisms, map morphisms from A to B to morphisms between FA and FB, as well as respecting composition. [3]

Definition 2 (Functor)**.** A functor F between categories $\mathcal{C}$ and $\mathcal{D}$ is made up of two components: object-part: Function between $obj(C)$ and $obj(D)$ arrow-part: Function between $arr(C)$ and $arr(D)$ with the property that compositionality is preserved. If $f : x \to y$ and $g : y \to z$ are arrows in C Then $F(f) \circ F(g) = F(f \circ g)$ [3]

Definition 3 (Product Category)**.** Given a category C and a category D, the product category, written C x D is constructed of objects made of pairs of $c \in C, d \in D$, and morphisms made from pairs of morphisms $f : c1 \to c2; g : d1 \to d2$ then (f , g) is a morphism from (c1 , d1) to (c2 , d2). It can be seen that these satisfy the property of being associative, and having identities. [13] n

The product operator on categories is not strictly associative in terms of equality, but it is common in category theory to only be interested in isomorphism of objects and in this case isomorphism of categories. We will see later the construction of a monoidal product and

Definition 4 (Multifunctor)**.**

- A *bifunctor* F from $CxC \to D$ is simply a functor from the product category C x C to D.
- A *trifunctor* F from $CxCxC \to D$ is likewise a functor from the product category C x C x C to D.

[3]

2.1.3 Natural Transformations

[16] Natural transformations provide a way to compare functors and encapsulate the idea of systematic correspondence between categorical maps.

Definition 5 (Natural Transformation)**.** Given a pair of categories C, D and a pair of functor $F, G : C \to D$. A natural transformation is a family of maps $\eta_x : Fx \to Gx$ for all $x : C$. This map must have the following property, called naturality property for all $x, y : C; f : x \to y$ arrow in C: $\eta_y \circ Ff = Gf \circ \eta_x$

2.2 Monoidal Categories

Monoidal categories are an extension of the notion of a *monoid*, a category with only one object, but instead of being composed by the category composition operator $\circ$ and identity arrows id, it works on the object level, having a separate tensor product $\otimes$ and identity object I. These must satisfy the monoid laws, but only up to natural isomorphisms. For example the associator given by $\alpha_{A,B,C}$,

$$\alpha_{A,B,C} : A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C$$

is an natural isomorphism and doesn't need to be a strict equality. However it does need to be subject to 'coherence laws' these are rules that state that certain diagrams commute (which means that when you have multiple paths to get between certain objects, then, if the transformations are used then the result will be an identical morphism. In other words, there's at most one canonical way to convert between two objects forllowing the transformations.

We now formally state the definition of monoidal category.

Definition 6 (Monoidal Category [15])**.** Given a category $\mathcal{C}$, a monoidal category is made up of the following data:

- $I : \mathcal{C}$: Identiy object
- $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$: A product bifunctor.
- $\alpha_{A,B,C} : A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C$
- $\rho_A : A \otimes I \equiv A$
- $\lambda_A : I \otimes A \equiv A$ Such that certain coherence conditions hold, spelled out CITE

2.3 Traced Monoidal Categories

traced-monoidal-categories

Definition 7 (Right Trace)**.** In a traced monoidal category $\mathcal{C}$, a *right trace* is a natural transformation

$$\mathrm{tr}_R : \prod_{X:\mathcal{C}} \mathcal{C}[A \otimes X,\ B \otimes X] \ \Rightarrow\ \mathcal{C}[A, B] \qquad \forall A, B : \mathcal{C}.$$

Equivalently, written with the parameter X as a superscript:

$$\mathrm{tr}_R^X : \mathcal{C}[A \otimes X,\ B \otimes X] \ \Rightarrow\ \mathcal{C}[A, B] \qquad \forall A, B, X : \mathcal{C}.$$

Intuitively, the trace can be understood as a feedback loop: part of the output of a morphism is "fed back" into ts input.

A right trace must satisfy the following axioms. Let $A, B, C, D, X, Y : \mathcal{C}$. (Originally introduced in Joyal et. al. 1996 [12].)

Axiom 1 (Tightening)**.** *For morphisms*

$$h : A \to B, \quad f : B \otimes X \to C \otimes X, \quad g : C \to D,$$

we have

$$\mathrm{tr}_R^X((g \otimes 1_X) \circ f \circ (h \otimes 1_X)) = g \circ \mathrm{tr}_R^X(f) \circ h.$$

Axiom 2 (Sliding)**.** *For morphisms*

$$f : A \otimes X \to B \otimes Y, \quad g : Y \to X,$$

we have

$$\mathrm{tr}_R^X((1_B \otimes g) \circ f) = \mathrm{tr}_R^Y(f \circ (1_A \otimes g)).$$

Axiom 3 (Vanishing)**.** *1. For all $f : A \otimes I \to B \otimes I$,*

$$\mathrm{tr}_R^I(f) = f,$$

where I is the monoidal unit.

2. For all $f : A \otimes (X \otimes Y) \to B \otimes (X \otimes Y)$,

$$\mathrm{tr}_R^{X \otimes Y}(f) = \mathrm{tr}_R^X(\mathrm{tr}_R^Y(f)).$$

Axiom 4 (Superposing)**.** *For morphisms*

$$f : A \otimes X \to B \otimes X, \quad g : C \to D,$$

we have

$$\mathrm{tr}_R^X(f) \otimes g = \mathrm{tr}_R^X(f \otimes g)$$

Definition 8 (Left trace operator)**.** A left trace is a trace operator on the opposite functor $\otimes^-$, obtained by swapping the order of the arguments, so $A \otimes^- B = B \otimes A$. [12]

Definition 9 (Planar Category)**.** A planar category is a category with a left and right trace. [12]

Definition 10 (Symmetric Trace Category)**.** A planar category is symmetric monoidal. if the cross transformation [12]

Definition 11 (Spherical Category)**.** A spherical category is category in which left and right traces agree. [4]

Definition 12 (Compact Closed Category)**.** A compact closed category is a traced monoidal category in which every object has a dual, which is roughly its inverse with respect to the monoidal product. Joyal-Stree-Verity introduce the **Int**-construction. [12]

Chapter 3

Formalization

formalization

3.1 Overview of Formalization Chapter

The formalization was performed in Cubical Agda. This chapter documents the development of the formalization, building up from basic lemmas to more complex results. It is generated from type-checked literate Agda files, allowing formal verifications of the definitions included.

There are three main approaches I tried:

- `SumTree`: based representations of finite sets.
- `InjFun`: Dependent sum representation of injective functions.
- `Inj`: Inductive representation of finite sets.

The tree-based representation comes from the intuition that we want to have tensor product, which must have an *associator*, which is a natural isomorphism with certain coherence conditions. These coherence conditions can be thought about as rebalancing of a tree and would become obvious if we had, say, that the semantic object of any `SumTree` is isomorphic to that of any other `SumTree`.

It became apparent early in the project that the tree-based representations weren't leading anywhere, so it was abandoned. You can find it in the source code (CITE repo). It is omitted from this report because it wasn't used to develop any significant result.

The `InjFun` approach uses the standard definition of injective function: a function paired with a proof of injectivity. This approach made associativity relatively simple, but caused problems with formally defining a trace operator. This is because there's no natural way to talk about the individual edges of the function's graph. Additionally, my approach of chaining injective functions preserved injectivity but lost information about the order of the elements, which is crucial to define the trace operator.

Finally, my last representation, `Inj`, uses an inductive definition of injective function, which has the drawback of obscuring somewhat

the function, but has the benefit of keeping the order of elements fixed between operations, and has a relatively intuitive representation of the trace operator.

3.2 Foundational Definitions

3.2.1 Operations on Paths

We will now begin the construction, starting with functions related to paths. In homotopy type theory CITE, a path is essentially a proof of equality. Here `ℓ` is a *level* of Grothenieck universe CITE. It can safely be ignored on first reading. `Type` is a generalization of `Set`. For our purposes the distinction doesn't mattter.

We define the negation of a path in the standard way, as a proof that the existence of the path leads to absurdity (that the empty type is inhabited).

```
_≠_ : ∀ {ℓ} {A : Type ℓ} → (x y : A) → Type ℓ
x ≠ y = x ≡ y → ⊥
```

We then define two convenience functions on `≠`, the symmetric form, and conjugation, which uses `cong` to show that `≠` is injective.

```
≠sym : {A : Type ℓ} {x y : A} → x ≠ y → y ≠ x
≠sym x≠y y≡x = x≠y (sym y≡x)

≠cong : ∀ {A B : Type} {x y : A} (f : A → B) → f x ≠ f y → x ≠ y
≠cong {A} {B} {x} {y} f fx≠fy x≡y = fx≠fy (cong f x≡y)
```

Next we introduce some foundational functions that are present in the Agda Cubical library CITE. Transport is the operation used to map along paths. Is says that if you have some type, then you can explicitly map every element of `A` to some equivalent element in `B`. (CITE: Type below copied from Agda Cubical library Cublical/Foundations/Prelude).

```
transport : {A B : Type ℓ} → A ≡ B → A → B
transport = _
```

Transport filler is a dependent path between a certain `x` in `A` and the explicit transport of `x` along a path. We will use this to construct a paths between objects and transports of those objects. (CITE: Type below copied from Agda Cubical library Cublical/Foundations/Prelude). It amounts to constructing a triangle where the bottom left and right vertices are `x`, the top right vertex is `transport p x`. The bottom edge is `refl` on `x`, the right edge is `p` and the contents of the triangle [1] the 'fill' form the dependent path. DIAGRAM

```
transport-filler : ∀ {ℓ} {A B : Type ℓ} (p : A ≡ B) (x : A)
                 → PathP (λ i → p i) x (transport p x)
transport-filler = _
```

[1] Strictly everying in cubical type theory but it is often more convenient to ignore that fact and think about aribitrary homotopy spaces

Returning to my definitions, we introduce a simple lemma that states that `subst` cancels a with `subst` on the inverse path.

```
subst-inv : ∀ {A : Type} {x y : A} (B : A → Type) (p : x ≡ y) → (pa : B y)
          → subst B p (subst B (sym p) pa) ≡ pa
subst-inv {A} {x} {y} B p a =
  subst B p (subst B (sym p) a) ≡⟨ refl ⟩
  subst B p (transport (λ i → B (p (~ i))) a)
    ≡⟨ transportTransport⁻ (λ i → B (p i)) a ⟩
  a ∎
```

Then we define a Singleton type. A singleton is HoTT [2] is an element `x` in a type `A` together with a proof that all other elements in the type are equal to that element. Note that this is exactly the same as saying that `A` is contractable with an explicit witness `x`. (CITE: Definition copied from 1Lab).

```
Singleton : ∀ {ℓ} {A : Type ℓ} → A → Type _
Singleton x = Σ[ y ∈ _ ] x ≡ y
```

We can make use of this to create a more convenient `inspect` function. `inspect'` takes a value and gives another member of that type together with a proof that they are both the same. Using this in a `with` clause lets us pattern match on `x` and have a proof that `x` is equal to it's epanded form, which is helpful for proving identities.

```
inspect' : ∀ {ℓ} {A : Type ℓ} (x : A) → Singleton x
inspect' x = x , refl
```

Next we will introduce a few complex looking helper functions. These all work using the same core idea. We want to show and identity which involves a transport operation, but to prove transport operations are equal we have to use `transport-filler` (discussed earlier in this section). As motivation, we are going to want to be able to 'swap' the order of application of a `subst` and some constructor, or function, say `fsuc`, the finite set successor (introduced later). Because both `fsuc` and `subst` have a different type to the input, you can't just swap their application order directly, you need to adjust the types on both sides. Consider the following lemma,

```
subst-fsuc-reorder
  : ∀ {x y : ℕ} → (p : x ≡ y) → (a : Fin x)
  → subst (Fin ∘ suc) p (fsuc a)
  ≡ fsuc (subst Fin p a)
subst-fsuc-reorder = _
```

On the left side of the `≡`, we see a subst applying to `(Fin ∘ suc)`, whereas on the right side, we see `subst` applied to just `Fin`, this is required because the path connects naturals, `x ≡ y`, and `a : Fin x`, but `fsuc a : Fin (suc x)`. This is what `transport-reorder` accounts for.

```
transport-reorder
  : ∀ {ℓ ℓ'} {A : Type ℓ} (B : A → Type ℓ') {x y : A}
  → (f : A → A) (g : {z : A} → B z → B (f z)) (p : x ≡ y) (a : B x)
```

[2] HoTT = Homotopy Type Theory

```
  → transport (λ i → B (f (p i))) (g a)
  ≡ g (transport (λ i → B (p i)) a)
```

To implement this, the idea is that we create a two dependent paths, that is a path that maps beteen different types across a secondary 'path family'. Each path uses transport-filler to create a dependent path between both components effectively creating two triangles (the transport-filler), 'glued' at a the edge `refl {g a}`. This gives us a dependent path `composite`, where the dependent component, `λ j → B (f (p (~ j)))` of `step1` is the exact symmetric form of the dependent component of `step2`, `λ j → B (f (p j))`. This means we can substitute the dependent component of `composite`, applying the groupoid cancellation law, to obtain a path with a constant family, i.e. a simple (non-dependent) path as required.

TODO: Diagram?

```
transport-reorder B f g p a =
  let
    step1 : (λ j → B (f (p (~ j))))
      [ transport (λ i → B (f (p i))) (g a)
      ≡ g a
      ]
    step1 = symP (transport-filler (λ i → B (f (p i))) (g a))
    step2 : (λ j → B (f (p j)))
      [ g a
      ≡ g (transport (λ i → B (p i)) a)
      ]
    step2 = congP (λ i o → g o) (transport-filler (λ i → B (p i)) a)
    composite : (λ i → B ((sym (cong f p) • (cong f p)) i))
      [ transport (λ i → B (f (p i))) (g a)
      ≡ g (transport (λ i → B (p i)) a)
      ]
    composite = compPathP' {B = B} step1 step2
  in
    -- Our path index goes out and back along the same path,
    -- so contract that to a point to give a non-dependent path.
    subst (λ o → PathP (λ i → B (o i))
                  (transport (λ i → B (f (p i))) (g a))
                  (g (transport (λ i → B (p i)) a)))
          (lCancel (cong f p)) composite

-- Synonym for transport-reorder
subst-reorder
  : ∀ {ℓ ℓ'} {A : Type ℓ} (B : A → Type ℓ') {x y : A}
  → (f : A → A) (g : {z : A} → B z → B (f z)) (p : x ≡ y) (a : B x)
  → subst B (cong f p) (g a)
  ≡ g (subst B p a)
subst-reorder B f g p a = transport-reorder B f g p a
```

We then apply the above construction to the subst2 case, where there are two simultaneous substiution paths. This can't use the above approach directly, because dependent paths only have a single path as their family. Instead we construct the path family to be the product of the two substitution paths `p`, `q`, but the overall idea is the same as `transport-reorder`.

```
subst2-reorder
  : ∀ {ℓ ℓ'} {A A' : Type ℓ}
```

```
    (B : A → A' → Type ℓ') (B' : A → A' → Type ℓ')
    {x y : A} {w z : A'}
  → (g : {z : A} {z' : A'} → B z z' → B' z z')
  → (p : x ≡ y) (q : w ≡ z) (a : B x w)
  → subst2 B' p q (g a) ≡ g (subst2 B p q a)
subst2-reorder B B' g p q a =
  let
    step1 : (λ j → B' (p (~ j)) (q (~ j)))
      [ transport (λ i → B' (p i) (q i)) (g a) ≡ g a ]
    step1 = symP (transport-filler (λ i → B' (p i) (q i)) (g a))

    step2 : (λ j → B' (p j) (q j))
      [ g a ≡ g (transport (λ i → B (p i) (q i)) a) ]
    step2 = congP (λ i o → g o) (transport-filler (λ i → B (p i) (q i)) a)

    composite : (λ i → B' ((sym p • p) i) ((sym q • q) i))
      [ transport (λ i → B' (p i) (q i)) (g a) ≡ g (transport (λ i → B (p i) (q i)) a) ]
    composite = compPathP' {B = λ x → B' (proj₁ x) (proj₂ x)}
                           step1 step2
  in subst2 (λ o □ → PathP (λ i → B' (o i) (□ i))
                   (transport (λ i → B' (p i) (q i)) (g a))
                   (g (transport (λ i → B (p i) (q i)) a)) )
       (lCancel p) (lCancel q) composite
```

This is the last use of the pattern. This states that a constructor in some variable `y` is equivalent to constructing in an equivalent value `x` and then substituting along the path between `x` and `y`. We will use this for base case constructors such as `fzero`.

```
resubst : ∀ {ℓ ℓ'} {A : Type ℓ} (B : A → Type ℓ')
        → (c : (z : A) → B z)
        → {x y : A} (p : x ≡ y)
        → c y ≡ subst B p (c x)
resubst B c {x = x} {y = y} p =
  let step1 : (λ i → B (p (~ i))) [ c y ≡ c x ]
      step1 i = c (p (~ i))
      step2 : (λ i → B (p i))
            [ c x
            ≡ subst B p (c x)
            ]
      step2 = transport-filler (λ i → B (p i)) (c x)
      composite : (λ i → B ((sym p • p) i))
        [ c y
        ≡ subst B p (c x)
        ]
      composite = compPathP' {B = B} step1 step2
  in subst (λ o → PathP (λ i → (B (o i))))
                 (c y)
                 (subst B p (c x)))
           (lCancel p) composite
```

Working with dependent paths directly is difficult and there is limited support in for path syntax to join depdendent paths together. This appears to be a limitation of Agda Cubical Library as well as 1Lab. I spent some time trying to create my own path syntax for `compPathP'`, but it would result in either unsolved metas, or syntax requiring so many additional arguments, that it wasn't worth using. It

is unclear to me whether this is a limitation of Cubical Agda itself, or whether an additional function could be added to the library to add ergonomics with depdendent paths. My apprach has been to avoid them as much as possible.

3.2.2 Properties of Sum Types

A (simple) sum type is another name for a disjoint union, or tagged union. It is denoted in Agda as `⊎`, with constructors `inl` and `inr` for the 'left' and 'right' parts of the union, respectively. There are a few lemmas missing in the Agda Cubical library; we prove them here.

This first lemma states that the two sides of the set are disjoint. We do this using a predicate `P` and `subst` to construct an element of `⊥`:

```
inl≠inr : {A B : Type} → (a : A) → (b : B) → inl a ≠ inr b
inl≠inr {A} {B} a b p = subst P p tt
  where
    P : A ⊎ B → Type
    P (inl _) = ⊤
    P (inr _) = ⊥
```

We can apply a similar trick to the above to prove injectivity for both `inl` and `inr`. (Provided by Naïm Camille Favier at https://agda.zulipchat.com/#narrow/chan cubical/topic/.E2.9C.94.20Splitting.20an.20interval.20in.20Agda.20Cubical). The idea is that we want to have a predictate that produces a path from `x` to the argument provided, in the `inl` case. Start with `refl {inl x}`, and `subst` the endpoint along the path `p` to `inl y`. This results in a path in `x ≡ y` as required. Note that the `inr` case is impossible, and only used for type-checking the predicate.

```
inl-injective : {A B : Type} (x y : A) → inl x ≡ inl y → x ≡ y
inl-injective {A} {B} x y p = subst P p refl
  where
    P : A ⊎ B → Type
    P (inl a) = x ≡ a
    P (inr b) = ⊥

inr-injective : {A B : Type} (x y : B) → inr x ≡ inr y → x ≡ y
inr-injective {A} {B} x y p = subst P p refl
  where
    P : A ⊎ B → Type
    P (inl a) = ⊥
    P (inr b) = x ≡ b
```

This lemma states that if you have `⊎`-map (which takes two functions that act on the left and right), on two identity functions, then the overall effect is the same as identity. This just comes down to using function extensionality, the notion that a pair of functions are equal if they produce the same outputs given the same inputs. Our proof that these two functions are exetnsionally equal is given as `e`, which is straight-forward.

```
⊎-map-id≡id : {A B : Type} → ⊎-map (id {A = A}) (id {A = B}) ≡ id
⊎-map-id≡id {A = A} {B} = funExt e
  where
```

```
    e : (x : A ⊎ B) → ⊎-map id id x ≡ id x
    e (inl x) = refl
    e (inr x) = refl
```

The same trick as above is applied to `⊎-map-∘`, which is a function that lets us say the that composition of two `⊎-map` pairs is the same as a single ⊎-map of the composition.

```
⊎-map-∘ : {A A' A'' B B' B'' : Type}
        → (f : A → A') (f' : A' → A'') (g : B → B') (g' : B' → B'')
        → ⊎-map f' g' ∘ ⊎-map f g ≡ ⊎-map (f' ∘ f) (g' ∘ g)
⊎-map-∘ f f' g g' = funExt e
  where
    e : (x : _) →
          (⊎-map f' g' ∘ ⊎-map f g) x ≡ ⊎-map (f' ∘ f) (g' ∘ g) x
    e (inl x) = refl
    e (inr x) = refl
```

Finally for this section, we define the simple lemma that states that if `mapMaybe` results in `nothing` then the input must have been `nothing`.

```
mapMaybeNothing : ∀ {A B : Type} → {x : Maybe A} {f : A → B}
                → map-Maybe f x ≡ nothing → x ≡ nothing
mapMaybeNothing {x = nothing} map≡∅ = refl
mapMaybeNothing {x = just x} map≡∅ = absurd (¬just≡nothing map≡∅)
```

3.3 Choice of Finite Set Representation

This project is a modification of an email I sent to Thorsten Altenkirch and Mark Williams on 30th June 2025.

In this project, finite sets have initially been defined in terms of the inductive type `Fin`. However, this representation does not carry over cleanly to cubical Agda. While the 1Lab library provides some support, it is considerably less developed than Agda's cubical library. Time was therefore spent transferring definitions from the 1Lab library to Agda cubical.

An important theorem that we want to prove is the equivalence of two ways of composing finite sets,

```
+≅⊎ : ∀ {x y} → Fin x ⊎ Fin y ≅ Fin (x + y)
+≅⊎ = _
```

Progress has been made up to Lemma 1.3. However, difficulties arose in the definition of `+→⊎`, where the use of a `with` clause obscured information required for type checking. This caused proof complexity to grow substantially. To address the issue, `+≅⊎` was redefined, but this in turn lead to obstacles in proving the following property:

```
ℕ≡→Fin≡ : ∀ {x : ℕ} (a b : Fin' x) → Fin'.lower a ≡ Fin'.lower b → a ≡ b
ℕ≡→Fin≡ {x} a b lowerEq = {!!}
```

The difficulty reduces to showing,

```
irr-subst : ∀ (a x y : ℕ) → (ix : Irr (a < x)) → (iy : Irr (a < y))
          → x ≡ y → {!ix ≡ iy!}
irr-subst = {!!}
```

Here, the central challenge lies in the fact that `ix` and `iy` are considered to have different types, which prevents even the statement of the goal. One potential direction is to employ the dependent form of `ap`, namely `apd`:

```
apd : ∀ {a b} {A : I → Type a} {B : (i : I) → A i → Type b}
    → (f : ∀ i (a : A i) → B i a) {x : A i0} {y : A i1}
    → (p : PathP A x y)
    → PathP (λ i → B i (p i)) (f i0 x) (f i1 y)
apd f p i = f i (p i)
```

At present, the use of `apd` does not seem natural, which likely reflects a lack of experience reasoning in explicitly homotopical terms. To address this, time was taken to review the HoTT book and Robert Harper's lecture series on homotopy type theory. CITE

It is worth noting that the analogous property is trivial to prove when `x` and `y` are of the same type, which suggests that the difficulty does not lie with `Irr` itself:

```
irr-is-prop : (X : Type) → (x y : Irr X) → x ≡ y
irr-is-prop X a b = refl
```

An advantage of 1Lab is that categories are defined cubically, without resorting to setoids. While this may become a limitation in later stages, its impact remains to be seen. It would in principle be possible to contribute additional material back to 1Lab, but this was agreed to be a distraction from the aims project in the available time.

The cubical library offers a broader range of definitions for `Fin` and more utilities for working with them. Consequently, the natural strategy is to commit to one definition of `Fin` that proves most convenient. At the time of writing, I had proven a few basic properties about virtual sets using the Agda standard library. The results have been partially ported to 1Lab, but it appears that porting instead to the cubical library would require a similar level of effort. The main difficulty with 1Lab lies in its particular definition of `Fin`, which complicates reasoning. Several alternatives are available:

```
-- Indexed inductive type
-- From agda-stdlib
private
  variable n : ℕ
data Fin₁ : ℕ → Type where
  fzero : Fin₁ (suc n)
  fsuc : (i : Fin n) → Fin₁ (suc n)

-- Bounded naturals (with irrelevant proof)
-- From 1lab
record Fin₂ (n : ℕ) : Type where
  constructor fin
  field
    lower : ℕ
```

```
    ⦃ bounded ⦄ : Irr (lower < n)

-- Agda Cubical main definition
Fin₃ : ℕ → Type
Fin₃ n = Σ[ k ∈ ℕ ] k < n

-- Alternative definition in Agda Cubical
Fin₄ : ℕ → Type
Fin₄ zero = ⊥
Fin₄ (suc n) = ⊤ ⊎ (Fin n)

-- Agda Cubical main except that <ᵗ is defined by computing the
-- trichotomy of two integers, and is either ⊤ or ⊥, making it
-- propositional.
_<ᵗ_ : (n m : ℕ) → Type
n <ᵗ zero = ⊥
zero <ᵗ suc m = ⊤
suc n <ᵗ suc m = n <ᵗ m


Fin₅ : ℕ → Type
Fin₅ n = Σ[ k ∈ ℕ ] k <ᵗ n
```

Although it would be possible to define an indexed inductive type in the familiar style, a comment in the cubical library explicitly advises against this approach:

```
-- Currently it is most convenient to define these as a subtype of the
-- natural numbers, because indexed inductive definitions don't behave
-- well with cubical Agda. This definition also has some more general
-- attractive properties, of course, such as easy conversion back to
-- ℕ.
```

This discourages redefining `Fin` in the traditional way. Instead, the trichotomous definition appears to offer the cleanest solution, since propositionality suffices for the purposes of this work. There is no need to track *how* `i < x` holds, only that it does.

In light of this, it seems preferable to transition to the standard cubical library at an early stage in the project. While this involved rewriting a substantial portion of the work completed so far, much of the necessary material already exists in the cubical library. The remaining work of porting was relatively straightforward. Before beginning this process, I took some time to familiarise fully with the cubical library's definitions to avoid reimplementing existing lemmas.

3.4 Definition of `Fin`

This section contains definitions and lemmas about finite sets, abbreviated 'Fin'. Specifically, for an natural `n : ℕ`, `Fin n` is a canonical finite set of size `n`.

3.4.1 Basic definition of `Fin`

We choose to define our own definition of Fin, which is identical to the one in the standard library (but not in the cubical library).

Unless otherwise stated, all definitions in this section originate from the standard library.

```
data Fin : ℕ → Type where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc : {n : ℕ} → Fin n → Fin (suc n)
```

Next we define some numerals, which will make it easier to construct definitions of injective functions.

```
f0 : Fin (1 ℕ.+ x)
f0 = fzero
f1 : Fin (2 ℕ.+ x)
f1 = fsuc f0
f2 : Fin (3 ℕ.+ x)
f2 = fsuc f1
f3 : Fin (4 ℕ.+ x)
f3 = fsuc f2
f4 : Fin (5 ℕ.+ x)
f4 = fsuc f3
f5 : Fin (6 ℕ.+ x)
f5 = fsuc f4
f6 : Fin (7 ℕ.+ x)
f6 = fsuc f5
f7 : Fin (8 ℕ.+ x)
f7 = fsuc f6
f8 : Fin (9 ℕ.+ x)
f8 = fsuc f7
f9 : Fin (10 ℕ.+ x)
f9 = fsuc f8
```

Next we construct an eliminator for `Fin`, that is a function that takes a map from all the ways to construct a `Fin`, and returns a certain arbitrary type. It saves explicitly pattern matching, which can be clearer in some cases.

```
elim : ∀ {A : {n : ℕ} → Fin (suc n) → Type }
     → ({n : ℕ} → A {n} fzero)
     → ({n : ℕ} → (a : Fin (suc n)) → A a → A (fsuc a))
     → ((m : ℕ) → (a : Fin (suc m)) → A a)
elim {A = A} z s m fzero = z
elim {A = A} z s (suc m) (fsuc a) = s a (elim {A = A} z s m a)
```

We define a predecessor function `pred` and a function `fmax` that returns the largest element in `Fin (suc x)`.

```
pred : {n : ℕ} → Fin (suc (suc n)) → Fin (suc n)
pred fzero = fzero
pred (fsuc n) = n

fmax : ∀ {x} → Fin (suc x)
fmax {zero} = fzero
fmax {suc x} = fsuc (fmax {x})
```

Next we inductively define methods to go to and from `ℕ`. The first is straight-forward, but the second requires a explicit proof that the input value will fit within the `Fin` set.

```
toℕ : ∀ {n} → Fin n → ℕ
toℕ fzero = zero
toℕ (fsuc x) = suc (toℕ x)
```

`fromℕ`, is considerably more complex as it requires adding an absurd case, requiring the monotonicity of `predℕ`.

```
fromℕ : ∀ {n} → (a : ℕ) → (a < n) → Fin n
fromℕ {zero} a a<0 = absurd {A = Fin 0} (¬-<-zero {a} a<0)
fromℕ {suc n} zero _ = fzero
fromℕ {suc n} (suc a) sa<sn = fsuc (fromℕ {n} a (pred-<-pred {a} {n} sa<sn))
```

Next we turn to absurdities and negations. `fzero≠fsuc` and `fsuc≠fzero` state the fact that we cannot have a value that is constructed two distinct ways. This is a fact that is true for all inductive types in general, but in Cubical Agda, an explicit transport is required to demonstrate this. We transport the element `tt : ⊤` (unit), into an inhabitant in `⊥` (the uninhabited type) which is an absurdity as required.

```
fzero≠fsuc : ∀ {x : ℕ} {i : Fin x} → fzero ≠ fsuc i
fzero≠fsuc {x} p = transport (cong P p) tt
  where
    P : {x : ℕ} → Fin (suc x) → Type
    P {x} fzero = ⊤
    P {x} (fsuc a) = ⊥

fsuc≠fzero : ∀ {x : ℕ} {i : Fin x} → fsuc i ≠ fzero
fsuc≠fzero = ≠sym fzero≠fsuc
```

Next we say that the empty `Fin` is equivalent to the canonical empty type `⊥`.

```
Fin0≃⊥ : Fin 0 ≃ ⊥
Fin0≃⊥ = (λ ()) , record { equiv-proof = λ y → absurd y }

Fin0-absurd : {A : Type ℓ} → Fin 0 → A
Fin0-absurd ()
```

Finally, we prove that `fsuc` is an injective function. This is another property of inductive definitions: *There is only one way to construct an element of an inductive type*.

```
fsuc-injective : ∀ {n} {i j : Fin n} → fsuc {n} i ≡ fsuc {n} j → i ≡ j
fsuc-injective {zero} {()} {()}
fsuc-injective {suc n} {i} {j} p = cong pred p
```

3.4.2 Trichotomy on `Fin`

A Total Order on `Fin`

Next we define an order operator on Fin. My plan was to make a type that is *propositional* for any two pairs of `Fin` using *trichotomy*, which is a type with three constructors for `<`,`≡`, and `>`.

We desire an type `Trichotomyᶠ` with the following properties:

1. We can compare two naturals and decide an order (greater, less or equal) between them.
2. such a comparison is `propositional`, there the instance of the type is unique for that pair.

3. It is decidable for any two pairs of `Fin`.
4. It is heterongeneous: It can compare Fin with distinct types.

The following inductive types satisfy all four desiderata above. We redefine an equivalence for heterogenity (point 4).

```
data _<ᶠ_ : {x y : ℕ} (a : Fin x) → (b : Fin y) → Type where
  <fzero : {b : Fin y} → (fzero {x}) <ᶠ fsuc b
  <fsuc : {a : Fin x} {b : Fin y} → a <ᶠ b → fsuc a <ᶠ fsuc b

data _≈ᶠ_ : {x y : ℕ} (a : Fin x) → (b : Fin y) → Type where
  ≈fzero : (fzero {x}) ≈ᶠ (fzero {y})
  ≈fsuc : {a : Fin x} {b : Fin y} → a ≈ᶠ b → fsuc a ≈ᶠ fsuc b
```

We also define a negation type in the usual way.

```
_≉ᶠ_ : Fin x → Fin y → Type
a ≉ᶠ b = ¬ a ≈ᶠ b
```

Properties on Order and Equivalence in `Fin`

We wll show that `_≈ᶠ_` is an equivalence relation (for transitivity see below `≈ᶠ-trans`), and prove some baisc properties inductively.

First, we symmetric and reflexivity of `≈ᶠ`.

```
≈fsym : a ≈ᶠ b → b ≈ᶠ a
≈fsym ≈fzero = ≈fzero
≈fsym (≈fsuc a≈b) = ≈fsuc (≈fsym a≈b)

≉fsym : a ≉ᶠ b → b ≉ᶠ a
≉fsym a≉b b≈a = a≉b (≈fsym b≈a)

≈refl : a ≈ᶠ a
≈refl {a = fzero} = ≈fzero
≈refl {a = fsuc a} = ≈fsuc (≈refl {a = a})
```

Next we show that `≈ᶠ` and `≡` are bi-converable.

```
≡→≈ᶠ : {a b : Fin x} → a ≡ b → a ≈ᶠ b
≡→≈ᶠ {a = a} a≡b = subst (a ≈ᶠ_) a≡b ≈refl

≈ᶠ→≡ : {a b : Fin x} → a ≈ᶠ b → a ≡ b
≈ᶠ→≡ ≈fzero = refl
≈ᶠ→≡ (≈fsuc a≈b) = cong fsuc (≈ᶠ→≡ a≈b)
```

And likewise for their complement,

```
≢→≉ᶠ : {a b : Fin x} → a ≢ b → a ≉ᶠ b
≢→≉ᶠ {a = a} a≢b a≈b = a≢b (≈ᶠ→≡ a≈b)

≉ᶠ→≢ : {a b : Fin x} → a ≉ᶠ b → a ≢ b
≉ᶠ→≢ a≉b a≡b = a≉b (≡→≈ᶠ a≡b)
```

We redefine `fzero≠fsuc` and `fsuc≠fzero` for `≈ᶠ`. Noteice that here we can immediately discharge the input as no inductive cases match.

```
fzero≉fsuc : fzero {x} ≉ᶠ fsuc b
fzero≉fsuc ()
```

```
fsuc≠fzero : fsuc a ≠ᶠ fzero {y}
fsuc≠fzero ()
```

Injectivity of ≈fsuc, and the negation of the predecessor.

```
≈fsuc-injective : fsuc a ≈ᶠ fsuc b → a ≈ᶠ b
≈fsuc-injective (≈fsuc a≈b) = a≈b

≠fpred : fsuc a ≠ᶠ fsuc b → a ≠ᶠ b
≠fpred a'≠b' a≈b = a'≠b' (≈fsuc a≈b)
```

Now we define a weak order simply by summing the posibilities.

```
_≤ᶠ_ : (a : Fin x) (b : Fin y) → Type
_≤ᶠ_ {x = x} {y = y} a b = (a <ᶠ b) ⊎ (a ≈ᶠ b)


≈ᶠ-trans : ∀ {x} {a : Fin x} {b : Fin y} {c : Fin z} → a ≈ᶠ b → b ≈ᶠ c → a ≈ᶠ c
≈ᶠ-trans ≈fzero ≈fzero = ≈fzero
≈ᶠ-trans (≈fsuc a≈b) (≈fsuc b≈c) = ≈fsuc (≈ᶠ-trans a≈b b≈c)


<ᶠ-trans : ∀ {x} {a : Fin x} {b : Fin y} {c : Fin z} → a <ᶠ b → b <ᶠ c → a <ᶠ c
<ᶠ-trans <fzero (<fsuc b<c) = <fzero
<ᶠ-trans (<fsuc a<b) (<fsuc b<c) = <fsuc (<ᶠ-trans a<b b<c)
```

Weakening of `<ᶠ`,

```
<-suc : ∀ (a : Fin x) → a <ᶠ fsuc a
<-suc fzero = <fzero
<-suc (fsuc a) = <fsuc (<-suc a)

≤-pred : ∀ {a : Fin x} {b : Fin y} → fsuc a ≤ᶠ fsuc b → a ≤ᶠ b
≤-pred (inl (<fsuc a<b)) = inl a<b
≤-pred (inr (≈fsuc a≈b)) = inr a≈b

fsuc≤fsuc : a ≤ᶠ b → fsuc a ≤ᶠ fsuc b
fsuc≤fsuc (inl a<b) = inl (<fsuc a<b)
fsuc≤fsuc (inr a≈b) = inr (≈fsuc a≈b)
```

we can convert between `a < fsuc b` and `a ≤ᶠ b`.

```
≤ᶠ→<ᶠ : {a : Fin x} {b : Fin y} → a ≤ᶠ b → a <ᶠ fsuc b
≤ᶠ→<ᶠ {b = b} (inl a<b) = <ᶠ-trans a<b (<-suc b)
≤ᶠ→<ᶠ (inr ≈fzero) = <fzero
≤ᶠ→<ᶠ (inr (≈fsuc a≈b)) = <fsuc (≤ᶠ→<ᶠ (inr a≈b))

<ᶠ→≤ᶠ : {a : Fin x} {b : Fin y} → a <ᶠ fsuc b → a ≤ᶠ b
<ᶠ→≤ᶠ {a = fzero} {fzero} a<b' = inr ≈fzero
<ᶠ→≤ᶠ {a = fzero} {fsuc b} 0<b' = inl <fzero
<ᶠ→≤ᶠ {a = fsuc a} {fsuc b} (<fsuc a<b) = fsuc≤fsuc (<ᶠ→≤ᶠ a<b)


¬a<a : {a : Fin x} → ¬ a <ᶠ a
¬a<a {a = fsuc a} (<fsuc a<a) = ¬a<a a<a


fsuc≤fsuc→<fsuc : (a≤b : a ≤ᶠ b) → ≤ᶠ→<ᶠ (fsuc≤fsuc a≤b) ≡ <fsuc (≤ᶠ→<ᶠ a≤b)
fsuc≤fsuc→<fsuc (inl x) = refl
fsuc≤fsuc→<fsuc (inr x) = refl
```

Now we will show mutual exclusion of the three cases:

```
<ᶠ→≯ : {a : Fin x} {b : Fin y} → a <ᶠ b → a ≯ᶠ b
<ᶠ→≯ {a = fzero} {b = fsuc b} <fzero a≈b = fzero≯fsuc a≈b
<ᶠ→≯ {a = fsuc a} {b = fsuc b} (<fsuc a<b) a≈b =
  <ᶠ→≯ {a = a} {b = b} a<b (≈fsuc-injective a≈b)

<ᶠ→≱ : {a b : Fin x} → a <ᶠ b → a ≱ b
<ᶠ→≱ a<b = ≯ᶠ→≱ (<ᶠ→≯ a<b)

≤ᶠ→≯ᶠ : {a : Fin x} {b : Fin y} → a ≤ᶠ b → ¬ b <ᶠ a
≤ᶠ→≯ᶠ (inl (<fsuc a<b)) (<fsuc a>b) = ≤ᶠ→≯ᶠ (inl a<b) a>b
≤ᶠ→≯ᶠ (inr a≈b) a>b = <ᶠ→≯ a>b (≈fsym a≈b)

<ᶠ→≯ᶠ : {a : Fin x} {b : Fin y} → a <ᶠ b → ¬ b <ᶠ a
<ᶠ→≯ᶠ a<b b<a = ¬a<a (<ᶠ-trans a<b b<a)
```

Next we define restrictions of Fin to a smaller domain.

```
fin-restrict-< : ∀ {x} {b : Fin (suc x)} (a : Fin y)
               → a <ᶠ b → Fin x
fin-restrict-< {x = suc x} fzero <fzero = fzero
fin-restrict-< {x = suc x} (fsuc a) (<fsuc a<b) = fsuc (fin-restrict-< a a<b)

fin-restrict-≤ : ∀ {x} {b : Fin x} (a : Fin y)
               → a ≤ᶠ b → Fin x
fin-restrict-≤ a a≤b = fin-restrict-< a (≤ᶠ→<ᶠ a≤b)

fin-restrict-<≡fin-restrict-≤
  : ∀ {x} {y} → {b : Fin x} (a : Fin y) → (a≤b : a ≤ᶠ b)
  → fin-restrict-< a (≤ᶠ→<ᶠ a≤b) ≡ fin-restrict-≤ a a≤b
fin-restrict-<≡fin-restrict-≤ a a≤b =
  refl
```

Definition of Trichotomy

Finally we are ready to define `Trichotomyᶠ`. This is an inductive type that is one of the 3 possibilites: less than, equal, or greater than.

```
data Trichotomyᶠ {x y} (a : Fin x) (b : Fin y) : Type where
  flt : a <ᶠ b → Trichotomyᶠ a b
  feq : a ≈ᶠ b → Trichotomyᶠ a b
  fgt : b <ᶠ a → Trichotomyᶠ a b

open Trichotomyᶠ
```

We also will make use of bichotomy, which in this context splits on less or equal (`fle`), or greather than (`fgt`).

```
data Dichotomyᶠ {x y} (a : Fin x) (b : Fin y) : Type where
  fle : a ≤ᶠ b → Dichotomyᶠ a b
  fgt : b <ᶠ a → Dichotomyᶠ a b

open Dichotomyᶠ
```

Now we will write a function that will decide which of the three cases apply. This is done by handling the base cases in `_≟ᶠ_`, and in

the successor-successor case, recursing and passing the result into the successor function `_≟ᶠ-suc_`, which handles the induction case.

```
_≟ᶠ-suc_ : ∀ {x} → (a : Fin x) (b : Fin y)
         → Trichotomyᶠ a b → Trichotomyᶠ (fsuc a) (fsuc b)
(a ≟ᶠ-suc b) (flt a<b) = flt (<fsuc a<b)
(a ≟ᶠ-suc b) (feq a≈b) = feq (≈fsuc a≈b)
(a ≟ᶠ-suc b) (fgt b<a) = fgt (<fsuc b<a)

_≟ᶠ_ : ∀ (a : Fin x) (b : Fin y) → Trichotomyᶠ a b
fzero ≟ᶠ fzero = feq (≈fzero)
fzero ≟ᶠ fsuc b = flt <fzero
fsuc a ≟ᶠ fzero = fgt <fzero
fsuc a ≟ᶠ fsuc b = (a ≟ᶠ-suc b) (a ≟ᶠ b)
```

There is an obvious map for `Trichotomyᶠ` to `Dichotomyᶠ`, which we can immediately use to decide bichotomy.

```
Trichotomy→Dichotomyᶠ
  : ∀ {x} {a : Fin x} {b : Fin y}
  → Trichotomyᶠ a b → Dichotomyᶠ a b
Trichotomy→Dichotomyᶠ (flt a<b) = fle (inl a<b)
Trichotomy→Dichotomyᶠ (feq a≈b) = fle (inr a≈b)
Trichotomy→Dichotomyᶠ (fgt b<a) = fgt b<a

_≤?ᶠ_ : (a : Fin x) (b : Fin y) → Dichotomyᶠ a b
a ≤?ᶠ b = Trichotomy→Dichotomyᶠ (a ≟ᶠ b)
```

Finally we define case splitting on bichotomy, which works like an if statement on bichotomy.

```
case≤?ᶠ : {A : Type} {m : ℕ} (a b : Fin m) → A → A → A
case≤?ᶠ a b x y = case (a ≤?ᶠ b) of
  λ{ (fle _) → x
   ; (fgt _) → y }
```

3.4.3 Proof of Propositionality of `Trichotomyᶠ`

We now prove the desired property for this trichotomy:

For any two pairs of `Fin`-elements, of possibly differing types, their trichotomy type is propositional, meaning that there is at most one of the three cases that holds for a given pair, and that has size at most 1.

We can go one step further, using the fact that a *contraction* is an inhabited *proposition*, and that *decidability* implies there is an inhabited case, we can conclude that `Trichotomyᶠ a b` is a *contraction* for any `Fin`-elements `a`, and `b`. These details are left omitted, as they weren't required for this project.

`isPropTrichotomyᶠ` follows from the mutual exclusions proven above. Propositionality of Dichotomy follows from this, but that has also been omitted for space.

```
isProp≈ᶠ : {a : Fin x} {b : Fin y} → isProp (a ≈ᶠ b)
isProp≈ᶠ ≈fzero ≈fzero = refl
```

```
isProp≈ᶠ (≈fsuc u) (≈fsuc v) = cong ≈fsuc (isProp≈ᶠ u v)

isProp<ᶠ : {a : Fin x} {b : Fin y} → isProp (a <ᶠ b)
isProp<ᶠ <fzero <fzero = refl
isProp<ᶠ (<fsuc u) (<fsuc v) = cong <fsuc (isProp<ᶠ u v)

isPropTrichotomyᶠ : {a : Fin x} {b : Fin y} → isProp (Trichotomyᶠ a b)
isPropTrichotomyᶠ (flt u) (flt v) = cong flt (isProp<ᶠ u v)
isPropTrichotomyᶠ (flt u) (feq v) = absurd (<ᶠ→≉ u v)
isPropTrichotomyᶠ (flt u) (fgt v) = absurd (<ᶠ→≯ᶠ u v)
isPropTrichotomyᶠ (feq u) (flt v) = absurd (<ᶠ→≉ v u)
isPropTrichotomyᶠ (feq u) (feq v) = cong feq (isProp≈ᶠ u v)
isPropTrichotomyᶠ (feq u) (fgt v) = absurd (<ᶠ→≉ v (≈fsym u))
isPropTrichotomyᶠ (fgt u) (flt v) = absurd (<ᶠ→≯ᶠ v u)
isPropTrichotomyᶠ (fgt u) (feq v) = absurd (<ᶠ→≉ u (≈fsym v))
isPropTrichotomyᶠ (fgt u) (fgt v) = cong fgt (isProp<ᶠ u v)
```

Further results used relating to `Fin` are omitted from this report, but found in the repository under CITE.

3.5 `InjFun` Representation

3.5.1 Definition of `InjFun`

We begin by defining the concept of injectivity. The standard way to define a injective function in type theory is to construct, using a dependent sum (Σ), the notion of a function paired with a proof of injectivity. Next we construct a simple injective function, `↣-id`, which is made up of the identity function, paired with a trivial proof that `id` is in fact injective. The notation for `_↣_` comes from the categroy theoretic notation of a monic arrow.

```
is-injective : {X Y : Type} → (f : X → Y) → Type
is-injective {X} f = ∀ (x y : X) → f x ≡ f y → x ≡ y

_↣_ : (X Y : Type) → Type
X ↣ Y = Σ (X → Y) is-injective

↣-id : (X : Type) → X ↣ X
↣-id X = (λ x → x) , (λ x y p → p)
```

Now we can define an alias for `Fin` and a type constructor call `InjFun`, with a square-bracket syntax, which is the function represention of finite injective functions:

```
⟦_⟧ : ℕ → Type
⟦_⟧ = Fin

-- 'InjFun'
[_↣_] : ℕ → ℕ → Type
[ X ↣ Y ] = ⟦ X ⟧ ↣ ⟦ Y ⟧
```

We introduce the notation of representing Fin functions as a list of indexes. The following represents a function that maps 0 to 1, 1 to 2 and 2 to 0. The indexes list in order of domain element, where

that element is mapped to. This is injective since no number appears twice. Note that there is some ambiguity with the notation in that we haven't specified the size of the codomain. Figure 3.1 is a graphical representation of the same function.

$$(1\ 2\ 0)$$

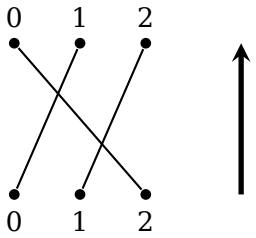


Figure 3.1: Plot of $(1\ 2\ 0)$. The bottom row is the domain and the top row is the codomain. The arrow indicates the direction from domain to codomain.

Note that the notation `⟦_⟧` is an alias for `Fin`, used to suggest the notion that `Fin` is a function from the objects of a category (`ℕ`) to the sets that those objects denote. It is used to hint at the notion of 'semantic brackets' from denotational semantics. We also use capital letters to represent the indexes in this section to hint that they are objects in a category, and not simply naturals.

We construct a helper function for the function part of `[_↠_]`.

```
FinFun : ∀ (A B : ℕ) → Type
FinFun A B = ⟦ A ⟧ → ⟦ B ⟧
```

3.5.2 Composition of Injective Function

We define composition on injective functions, `↠`, by composing the underlying functions and chaining the injectivity proof. The unit and associativity laws hold by definitional equality in this encoding, so the proofs are `refl`. This is intentional: we picked a representation that makes the algebra easy, and Agda is powerful enough to recognize this. These four lemmas are most of what we need to construct a category of injective functions.

```
infixl 5 _↠∘↠_

_↠∘↠_ : (B ↠ C) → (A ↠ B) → (A ↠ C)
(f , inj₁) ↠∘↠ (g , inj₂) = (f ∘ g) , λ x y eq → inj₂ _ _ (inj₁ _ _ eq)

↠∘↠-idR : ∀ {X Y : Type} (f : X ↠ Y) → f ↠∘↠ ↠-id X ≡ f
↠∘↠-idR (f , f-inj) = refl

↠∘↠-idL : ∀ {X Y : Type} (f : X ↠ Y) → ↠-id Y ↠∘↠ f ≡ f
↠∘↠-idL (f , f-inj) = refl
```

```
↠∘↠-assoc : ∀ {A B C D : Type} (h : C ↠ D) (g : B ↠ C) (f : A ↠ B)
          → h ↠∘↠ (g ↠∘↠ f) ≡ (h ↠∘↠ g) ↠∘↠ f
↠∘↠-assoc h g f = refl
```

We additionally write a function to convert between *paths* and injections.

```
≡to↠ : ∀ {A B} → A ≡ B → A ↠ B
≡to↠ p = transport p , λ x y q → transport-inj p q

refl-to-↠-is-id : ∀ {A} → fst (≡to↠ (refl {x = A})) ≡ id
refl-to-↠-is-id =
  funExt (λ x →
    fst (≡to↠ refl) x ≡⟨ refl ⟩
    transport refl x ≡⟨ transportRefl x ⟩
    x ∎)
```

3.5.3 Properties of `InjFun`

Next we show that `is-injective` is a proposition when applied to a `FinFun`. This is done by making use of a chain of library lemmas that strip off one lambda/Π at a time, before using the fact that `Fin` is a set, and implicitly using the fact that `x ≡ y` is a proposition whenever the containing type is a set.

%TODO: Define prop and set

```
isProp-is-injective : {m n : ℕ} → (f : FinFun m n) → isProp (is-injective f)
isProp-is-injective {m} f = isPropΠ λ x → isPropΠ λ y → isProp→ (isSetFin x y)
```

The importance of this lemma is it means that there at most one way that any particular function can be represented in as an injective function, so a pair of injective finite functions are equal if and only if their injection form are equal. It also lets us prove the following lemma that each `InjFun` is a set, meaning that there are no 'hidden' homotopic behaviour other than simple equality between members with the same type.

```
isSetInjFun : {m n : ℕ} → isSet [ m ↠ n ]
isSetInjFun {m} {n} =
  isSetΣ (isSet→ isSetFin)
         (λ f → isProp→isSet (isProp-is-injective f))
```

3.5.4 Equivalence Relation on `InjFun`

We define a heterogeneous relation on `InjFun` that allows us to compare two injective functions across equal types which might not be definitionally equal, meaning Agda can't normalize them to the same value. For example, `X + Y ≡ Y + X` for `X Y : ℕ` in the sense of path equality, but they don't both reduce to the same normal form.

```
infix 8 _≈_

record _≈_ {A B X Y : ℕ} (f : [ A ↠ X ]) (g : [ B ↠ Y ]) : Type where
  field
```

```
    p : A ≡ B
    q : X ≡ Y
    path : (λ i → cong₂ FinFun p q i) [ fst f ≡ fst g ]
```

We then show that is is an equivalence relation, that is a relation that is (a) reflexive, (b) symmetric, and (c) transitive. This means informally that for all `A B C X Y Z : 𝕟, f : [ A ↣ X ], g : [ B ↣ Y ], g : [ C ↣ Z ]`,

(a) `f ≡ f`

(b) `f ≡ g → g ≡ f`

(c) `f ≡ g → g ≡ h → f ≡ h`

```
≈refl : {A X : 𝕟} (f : [ A ↣ X ]) → f ≈ f
≈refl {A} {X} f = record
  { p = refl
  ; q = refl
  ; path = λ i x → fst f x
  }

≈sym : ∀ {A B X Y : 𝕟} {f : [ A ↣ X ]} {g : [ B ↣ Y ]} → f ≈ g → g ≈ f
≈sym {A} {B} {X} {Y} {f} {g} f≈g = record
  { p = sym p
  ; q = sym q
  ; path = λ i → path (~ i)
  }
  where
    open _≈_ f≈g
```

Transitivity is the only difficut one. We construct it in a module for convenience. TODO: Insert diagram

```
module Trans {A B C X Y Z : 𝕟}
             {f : [ A ↣ X ]}
             {g : [ B ↣ Y ]}
             {h : [ C ↣ Z ]}
             (g≈h : g ≈ h) (f≈g : f ≈ g) where

  open _≈_ f≈g renaming (p to p1; q to q1; path to path1)
  open _≈_ g≈h renaming (p to p2; q to q2; path to path2)
  r1 : FinFun A X ≡ FinFun B Y
  r1 i = FinFun (p1 i) (q1 i)
  r2 : FinFun B Y ≡ FinFun C Z
  r2 i = FinFun (p2 i) (q2 i)
  ≈trans : f ≈ h
  ≈trans = record
    { p = p1 • p2
    ; q = q1 • q2
    ; path = path'
    }
    where
      c2 : cong₂ FinFun (p1 • p2) (q1 • q2) ≡
           cong₂ FinFun p1 q1 • cong₂ FinFun p2 q2
      c2 = cong₂-• FinFun p1 p2 q1 q2
      path : (λ j → (cong₂ FinFun p1 q1 • cong₂ FinFun p2 q2) j) [ fst f ≡ fst h ]
      path = compPathP path1 path2
```

```
      path' : (λ j → (cong₂ FinFun (p1 • p2) (q1 • q2)) j) [ fst f ≡ fst h ]
      path' = subst⁻ (λ ○ → (λ j → ○ j) [ fst f ≡ fst h ]) c2 path

  infixl 4 _○≈_
  _○≈_ : f ≈ h
  _○≈_ = ≈trans

open Trans using (≈trans; _○≈_) public
```

We also define reverse composition, and a path syntax derived from the one in the cubical standard library for ≡.

```
-- reverse composition
infixl 4 _≈○_
_≈○_ : {A B C X Y Z : ℕ}
       {f : [ A ↣ X ]}
       {g : [ B ↣ Y ]}
       {h : [ C ↣ Z ]}
       (f≈g : f ≈ g) (g≈h : g ≈ h) → f ≈ h
f≈g ≈○ g≈h = ≈trans g≈h f≈g

infixr 2 _≈⟨_⟩_
_≈⟨_⟩_ : {A B C X Y Z : ℕ}
       → (f : [ A ↣ X ]) → {g : [ B ↣ Y ]} → {h : [ C ↣ Z ]}
       → f ≈ g → g ≈ h → f ≈ h
_ ≈⟨ f≈g ⟩ g≈h = g≈h ○≈ f≈g

infix 3 _∎
_∎ : {A X : ℕ} → (f : [ A ↣ X ]) → f ≈ f
f ∎ = ≈refl f
```

Finally we construct a transport operation for `InjFun` types. (See Section 3.2.1.)

```
≈transport : ∀ {A B X Y : ℕ} → A ≡ B → X ≡ Y → [ A ↣ X ] → [ B ↣ Y ]
≈transport {A = A} {B = B} p q f = X↣Y ↣○↣ f ↣○↣ B↣A
  where
    B↣A = ≡to↣ (cong Fin (sym p))
    X↣Y = ≡to↣ (cong Fin q)


≈transport-fun : ∀ {A B X Y : ℕ} → A ≡ B → X ≡ Y
                 → ([[ A ]] → [[ X ]]) → [[ B ]] → [[ Y ]]
≈transport-fun p q f =
  subst Fin q ∘ f ∘ subst Fin (sym p)
```

Next we define a filler to show that a function has a path to its transported version.

```
≈transport-filler : ∀ {A B X Y : ℕ} → (p : A ≡ B) → (q : X ≡ Y)
                    → (f : [ A ↣ X ]) → f ≈ ≈transport p q f
≈transport-filler {A = A} {B} {X} {Y} p q f =
  record { p = p ; q = q ; path = path }
  where
    B↣A = ≡to↣ (cong Fin (sym p))
    X↣Y = ≡to↣ (cong Fin q)
    P = cong₂ FinFun p q
    path : (λ i → cong₂ FinFun p q i)
      [ fst f
      ≡ subst Fin q ∘ fst f ∘ subst Fin (sym p)
```

```
    ]
  path = subst2-filler FinFun p q (fst f)
```

`from≡` is for the simple case where the indexes are the same at both ends. In this case, `path` is a simple (non-dependent) path.

```
from≡ : ∀ {A X : ℕ} → {f g : [ A ↣ X ]}
      → fst f ≡ fst g → f ≈ g
from≡ path = record
  { p = refl
  ; q = refl
  ; path = path
  }
```

After using this definition for a bit, I found that support for dependent paths, which are paths in which the type of the path transforms along with the value, in the cubical library is significantly behind what is available as 'simple' (non-dependent) paths.

3.5.5 Isomorphisms

We will also be making use of isomorphism types. These are an essential component of cubical type theory, providing a way to formalize the idea that two types can be considered equivalent when there exist mutually inverse functions between them. `Iso A B` is the isomorphism type between `A` and `B`, often drawn as `A ≅ B` in mathematical notation. In this context, I will use a slightly differnt convention and notation to access the standard defnition. I use `A ≅ B` to denote the isomorphism type, and have different accessors. Suppose `f : A ≅ B` is an isomorphism. Then it has four components: - `f ^ : A → B` - the forward function - `f ⁻¹ : B → A` - the inverse function - `rinv : f ^ ∘ f ⁻¹ ≡ id` - right inverse - `linv : f ⁻¹ ∘ f ^ ≡ id` - left inverse

rinv and linv are the function extensionality version of the one in the definition of `Iso`.

```
infix 1 _≅_

_≅_ : (A B : Type) → Type
A ≅ B = Iso A B
```

We introduce an adapter. A symmetric function `flip-≅`, and a transitive function `_≅∘≅_`.

```
mkIso : {A B : Type}
      → (f : A → B) → (g : B → A)
      → f ∘ g ≡ id → g ∘ f ≡ id
      → Iso A B
mkIso {A} {B} f g ri li =
  iso f g r l
  where
    l : (a : A) → g (f a) ≡ a
    l a = cong (λ o → o a) li
    r : (b : B) → f (g b) ≡ b
    r b = cong (λ o → o b) ri

flip-≅ : (A ≅ B) → (B ≅ A)
```

```
flip-≅ A≅B = invIso A≅B

infixl 9 _≅∘≅_

_≅∘≅_ : (B ≅ C) → (A ≅ B) → (A ≅ C)
g ≅∘≅ f = compIso f g
```

Next we create a function to convert between isomorphisms and injections.

```
≅to↣ : (A ≅ B) → (A ↣ B)
≅to↣ f =
  fun f , inj
  where
    open Iso
    inj : is-injective (fun f)
    inj x y eq =
      x
        ≡⟨ sym (cong (λ ○ → ○ x) (linv f)) ⟩
      (inv f ∘ fun f) x
        ≡⟨ refl ⟩
      inv f (fun f x)
        ≡⟨ cong (inv f) eq ⟩
      inv f (fun f y)
        ≡⟨ refl ⟩
      (inv f ∘ fun f) y
        ≡⟨ cong (λ ○ → ○ y) (linv f) ⟩
      y ∎
```

3.5.6 Coproduct Map for Injective Functions

This construction defines an injection on a sum type by combining injections for each summand. The mapping function, `⊎-map`, applies the respective injections to either side of the sum. Injectivity is proven by a case analysis: (a) `inl` and `inr` are disjoint, so their images can't clash; (b) if both inputs come from the same side, injectivity follows from the branch's injectivity and the fact that constructors themselves are injective. The proof utilizes previously established lemmas about sums.

```
↣-map-⊎ : {A B C D : Type} → (A ↣ B) → (C ↣ D) → ((A ⊎ C) ↣ (B ⊎ D))
↣-map-⊎ {A} {B} {C} {D} f g = h , inj
  where
    h : (A ⊎ C) → (B ⊎ D)
    h = ⊎-map (fst f) (fst g)

    inj : (x y : A ⊎ C) → h x ≡ h y → x ≡ y
    inj (inl a₁) (inl a₂) hx≡hy =
        cong inl
      $ snd f a₁ a₂
      $ inl-injective (fst f a₁) (fst f a₂)
      $ hx≡hy
    inj (inl a₁) (inr c₂) hx≡hy =
      absurd (inl≠inr (fst f a₁) (fst g c₂) hx≡hy)
    inj (inr c₁) (inl a₂) hx≡hy =
      absurd (inl≠inr (fst f a₂) (fst g c₁) (sym hx≡hy))
    inj (inr c₁) (inr c₂) hx≡hy =
```

```
  cong inr
$ snd g c₁ c₂
$ inr-injective (fst g c₁) (fst g c₂)
$ hx≡hy
```

3.5.7 Relating Identity Functions to `transport`

Te final equalities of this section export the idea that `transport p` and `subst B p` are *paths of functions* (i.e., the identity deforms into the transport operator). These are obtained from `transport-filler` using the function extensionality principle (`funExt`).

```
id≡transport : ∀ {ℓ} {A B : Type ℓ} (p : A ≡ B)
             → (λ i → A → p i) [ id ≡ transport p ]
id≡transport p = funExt (transport-filler p)

id≡subst : ∀ {ℓ ℓ'} {A : Type ℓ} {x y : A}
  → (B : A → Type ℓ') (p : x ≡ y)
  → (λ i → B x → B (p i)) [ id ≡ subst B p ]
id≡subst B p = funExt (subst-filler B p)
```

3.5.8 Splitting a Finite Set

To define an tensor on a pair of finite functions, the idea is that we will split the domain, pefrom the injective map separately, and then join the domains back into a single domain. We define these operations and show that they are inverses by constructing an isomorphism.

We start with the split function `+→⊎`, which takes a single domain, and splits it into a left and a right part in such a way that any value less than `X` will be sorted on the left and any values greater than or equal to `X` will appear on the right part. We make use of a helper function `step` which handles the successor case, taking the output of the recursive call and shifting the left domain up by 1. This gives us one diraction of the bi-conversion.

```
inc : ∀ (X Y : ℕ) → ⟦ X ⟧ ⊎ ⟦ Y ⟧ → ⟦ suc X ⟧ ⊎ ⟦ Y ⟧
inc X Y (inl a) = inl (fsuc a)
inc X Y (inr a) = inr a

+→⊎ : ∀ (X Y : ℕ) → ⟦ X + Y ⟧ → ⟦ X ⟧ ⊎ ⟦ Y ⟧
+→⊎ zero Y a = inr a
+→⊎ (suc X) Y fzero = inl fzero
+→⊎ (suc X) Y (fsuc a) = inc X Y (+→⊎ X Y a)
```

To write the join operation `⊎→+`, we need to define a pair of operations on `Fin`: `fshift` and `finject` are complementary functions that have ranges that don't overlap but do cover the codomain. We will use this property when defining `⊎≅+`.

```
fshift : (x : ℕ) → {y : ℕ} → Fin y → Fin (x + y)
fshift zero a = a
fshift (suc x) a = fsuc (fshift x a)
```

```
finject : {x : ℕ} → (y : ℕ) → Fin x → Fin (x + y)
finject {suc _} _ fzero = fzero
finject {suc x} y (fsuc a) = fsuc (finject {x} y a)
```

Now the join function, decides based on which side the input came in from whether to inject (keeping the values the same), or whether to shift them up by `X`.

```
⊎→+ : ∀ (X Y : ℕ) → ⟦ X ⟧ ⊎ ⟦ Y ⟧ → ⟦ X + Y ⟧
⊎→+ X Y (inl a) = finject Y a
⊎→+ X Y (inr a) = fshift X a
```

We prove a lemma that if the right domain is empty then splitting will result in an empty right set (proof ommitted).

```
+→⊎-X-0≡inl : (X : ℕ) (a : ⟦ X + 0 ⟧)
            → +→⊎ X 0 a ≡ inl (subst Fin (+-zero X) a)
```

Now we can define the two cancelation laws, called *section* and *retraction* in homotpy type theory CITE. These proofs are left out of this dissertation for space, but they amount to pattern matching and chaining equivalences. Finally we join these together into an isomorphism, proving equivalence of the two representations (split and joined). We can then use these euquivalences to create a tensor product as we'll see next.

```
sect : ∀ (X Y : ℕ) → section {A = ⟦ X ⟧ ⊎ ⟦ Y ⟧} (⊎→+ X Y) (+→⊎ X Y)
retr : ∀ (X Y : ℕ) → retract {A = ⟦ X ⟧ ⊎ ⟦ Y ⟧} (⊎→+ X Y) (+→⊎ X Y)

⊎≅+ : ∀ (X Y : ℕ) → Iso (⟦ X ⟧ ⊎ ⟦ Y ⟧) ⟦ X + Y ⟧
⊎≅+ X Y = iso (⊎→+ X Y) (+→⊎ X Y) (sect X Y) (retr X Y)
```

3.5.9 Tensor Product on `InjFun`

We now move to detail operations and properties on `InjFun`.

First we define compositional identity (`Id`) and monoidal unit (`𝟘`).

```
Id : ∀ {X} → [ X ↣ X ]
Id = (λ x → x) , λ x y eq' → eq'

𝟘 : [ 0 ↣ 0 ]
𝟘 = ↣-id ⟦ 0 ⟧
```

We use additive notation (`𝟘` and `⊕`) for the tensor product and identity because our tensor operation merges two injections via the coproduct map `↣-map-⊎`, summing both domain and codomain sizes. The tensor is defined as a composition of three steps (which appear from right to left in the definition): `- ≅to↣ (flip-≅ (⊎≅+ k m))` splits the input into `Fin k` or `Fin m`. `- ↣-map-⊎ f g` applies `f` or `g`, depending on the split. `- ≅to↣ (⊎≅+ l n)` combines the outputs.

The Agda code is:

```
tensor : ∀ {k l m n : ℕ} → [ k ↣ l ] → [ m ↣ n ] → [ k + m ↣ l + n ]
tensor {k} {l} {m} {n} f g = ≅to↣ (⊎≅+ l n) ↣∘↣ ↣-map-⊎ f g ↣∘↣ ≅to↣ (flip-≅ (⊎≅+ k m))

infixl 30 _⊕_
```

```
_⊕_ : ∀ {k l m n : ℕ} → [ k ↣ l ] → [ m ↣ n ] → [ k + m ↣ l + n ]
f ⊕ g = tensor f g
```

`⊕` forms a coproduct structure on the category of injective functions.

We then prove some properties about this tensor. `Id⊕Id≡Id` states that placing two identity arrows 'side by side' results in another identity arrow. We define the function part first, and then use the fact that `is-injective` is a proposition to show the equality holds for the dependent sum.

```
id⊕id≡id : {m n : ℕ} → ⊎→+ m n ∘ ⊎-map id id ∘ +→⊎ m n ≡ id
id⊕id≡id {m} {n} =
  ⊎→+ m n ∘ ⊎-map id id ∘ +→⊎ m n
    ≡⟨ cong (λ ○ → ⊎→+ m n ∘ ○ ∘ +→⊎ m n) ⊎-map-id≡id ⟩
  ⊎→+ m n ∘ +→⊎ m n
    ≡⟨ funExt (⊎+sect m n) ⟩
  id ∎


Id⊕Id≡Id : {m n : ℕ} → Id {m} ⊕ Id {n} ≡ Id {m + n}
Id⊕Id≡Id {m} {n} = cong₂ _,_ id⊕id≡id s
  where r : subst is-injective id⊕id≡id (snd (Id {m} ⊕ Id {n})) ≡ snd (Id {m + n})
        r = isProp→is-injective id (subst is-injective id⊕id≡id (snd (Id {m} ⊕ Id {n}))) (snd (Id {
        s : (λ i → is-injective (id⊕id≡id i))
          [ snd (Id {m} ⊕ Id {n}) ≡ snd (Id {m + n}) ]
        s = compPathP' (subst-filler is-injective id⊕id≡id (snd (Id {m} ⊕ Id {n}))) r
```

For convenience we have a short-hand for adding an identity arrow on the left or right.

```
ladd : ∀ {l m : ℕ} → (A : ℕ) → [ l ↣ m ] → [ A + l ↣ A + m ]
ladd {l} {m} A f = (Id {A}) ⊕ f

radd : ∀ {l m : ℕ} → (A : ℕ) → [ l ↣ m ] → [ l + A ↣ m + A ]
radd {l} {m} A f = f ⊕ (Id {A})
```

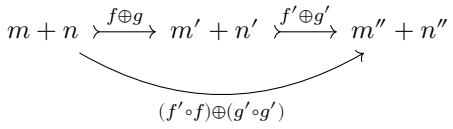


Figure 3.2: Direct sum of injections preserves composition: the composition of direct sums matches the direct sum of compositions, i.e., this diagram commutes.

Next we show that for the operation `_⊕_`: The property `⊕-preserves-∘` demonstrates that the direct sum (coproduct) of injective functions is compatible with composition: composing two pairs of injections separately and then taking their direct sum yields the same result as first taking the direct sums and then composing those. Formally, for injections `f`, `f'`, `g`, `g'`, the equation `(f' ∘ f) ⊕ (g' ∘ g) = (f' ⊕ g') ∘ (f ⊕ g)` ensures the tensor operation respects function composition, and that the sum operation acts functorially on the category of injective functions. See figure 3.2.

```
⊕-preserves-∘
  : ∀ {m m' m'' n n' n''}
  → (f : [ m ↠ m' ]) (f' : [ m' ↠ m'' ]) (g : [ n ↠ n' ]) (g' : [ n' ↠ n'' ])
  → ((f' ↠∘↠ f) ⊕ (g' ↠∘↠ g)) ≈ ((f' ⊕ g') ↠∘↠ (f ⊕ g))
```

Now we begin defining the associator α for tensor products:

- `α-p-dom` and `α-p-cod` is the domain and codomain indexes respectively.
- `α-p` is the path between the type of the right associated function and the left associated depdendent sum.
- `α-p-fun` is the same as `α-p` but for just the function part.
- `α-iso` is the map as an isomorphism, which is for proving associator is a natural isomorphism.
- Finally, α is the right-to-left transport, and α^{-1} is the left-to-right transport.

```
module _ {l l' m m' n n' : ℕ} where
  α-p-dom : l + (m + n) ≡ (l + m) + n
  α-p-dom = +-assoc l m n

  α-p-cod : l' + (m' + n') ≡ (l' + m') + n'
  α-p-cod = +-assoc l' m' n'

  α-p : [ (l + (m + n)) ↠ (l' + (m' + n')) ]
      ≡ [ ((l + m) + n) ↠ ((l' + m') + n') ]
  α-p = cong₂ [_↠_] (+-assoc l m n) (+-assoc l' m' n')

  α-p-fun : (Fin (l + (m + n)) → Fin (l' + (m' + n')))
          ≡ (Fin ((l + m) + n) → Fin ((l' + m') + n'))
  α-p-fun = cong₂ FinFun α-p-dom α-p-cod

  α-iso : Iso [ (l + (m + n)) ↠ (l' + (m' + n')) ]
              [ ((l + m) + n) ↠ ((l' + m') + n') ]
  α-iso = pathToIso α-p

  α : [ (l + (m + n)) ↠ (l' + (m' + n')) ]
    → [ ((l + m) + n) ↠ ((l' + m') + n') ]
  α = Iso.fun α-iso

  α⁻¹ : [ ((l + m) + n) ↠ ((l' + m') + n') ]
      → [ (l + (m + n)) ↠ (l' + (m' + n')) ]
  α⁻¹ = Iso.inv α-iso
```

We make use of a handy result: For a pair of `InjFun`'s to be equal, it is sufficient that their function parts are equal.

```
funPath↠InjFunPath : {m m' : ℕ} → (f g : [ m ↠ m' ])
                   → fst f ≡ fst g → f ≡ g
```

We will need ⊎-assoc which gives an isomorphism between the two ways of associating sums three values with a sum. We abbreviate the definition, which is spelled out fully in VSet.Data.Sum CITE.

```
⊎-assoc : {A B C : Type} → A ⊎ (B ⊎ C) ≅ (A ⊎ B) ⊎ C
⊎-assoc = record
  { fun = f
```

```
  ; inv = g
  ; leftInv = retr
  ; rightInv = sect
  }
  where
    f : {A B C : Type} → A ⊎ (B ⊎ C) → (A ⊎ B) ⊎ C
    g : {A B C : Type} → (A ⊎ B) ⊎ C → A ⊎ (B ⊎ C)
    sect : section f g
    retr : retract f g
```

Proving tensor associativity in the general case requires proving this.

```
assoc-ext' : {l l' m m' n n' : ℕ}
  → (f : Fin l → Fin l') (g : Fin m → Fin m') (h : Fin n → Fin n')
  → ∀ x
  → (⊎→+ (l' +ℕ m') n'
      (⊎-map (⊎→+ l' m') id
        (⊎-map (⊎-map f g) h
          (⊎-map (+→⊎ l m) id
            (+→⊎ (l +ℕ m) n
              (x)))))))
  ≡ (subst Fin α-p-cod
      (⊎→+ l' (m' +ℕ n')
        (⊎-map id (⊎→+ m' n')
          (⊎-map f (⊎-map g h)
            (⊎-map id (+→⊎ m n)
              (+→⊎ l (m +ℕ n)
                (subst Fin (sym α-p-dom)
                  (x)))))))))
```

I expect that proving this should be relatively straight-forward by splitting it into parts, however this was a lemma that I haven't had time to return to.

3.5.10 Trace operations on `InjFun`

The final construction for the dependent sum '`InjFun`' approach is the `trace` operator. In principle, a trace is an operator that decreases the domain and codomain of an injective function by the same amount. It will be easiest to start thinking about a trace of 1, that we call `pred` (for predecessor), and then define the trace operator as the repeated `pred` operations.

Start by considering an injective function, the graph of which is given by figure 3.3 as step 0. The information flows in one direction, indicated by the arrow on the right. This example is bijective but we don't require surjection. Now the predecessor operation ('pred' or 'trace 1') on this function is visualized as a backward 'feedback' edge from 0 in the codomain back to 0 in the domain. We consider each node to have at most one input and one output, meaning that 'd0' (short for domain element 0) can't be accessed from an input function, since it already has one output and one input. Likewise 'c0' (codomain element 0) already has one input and output, so there are no extra 'connection points' to take the output from. Hence the domain has been restricted to $\{1, 2\}$ and the range to $\{1, 2, 3\}$, as given

in 3.3 step 2. By convention, we shift all the values down by a fixed amount so that we always have a domain and codomain composed of a consecutive sequence starting from 0. The final result is given by figure 3.3 as step 3.

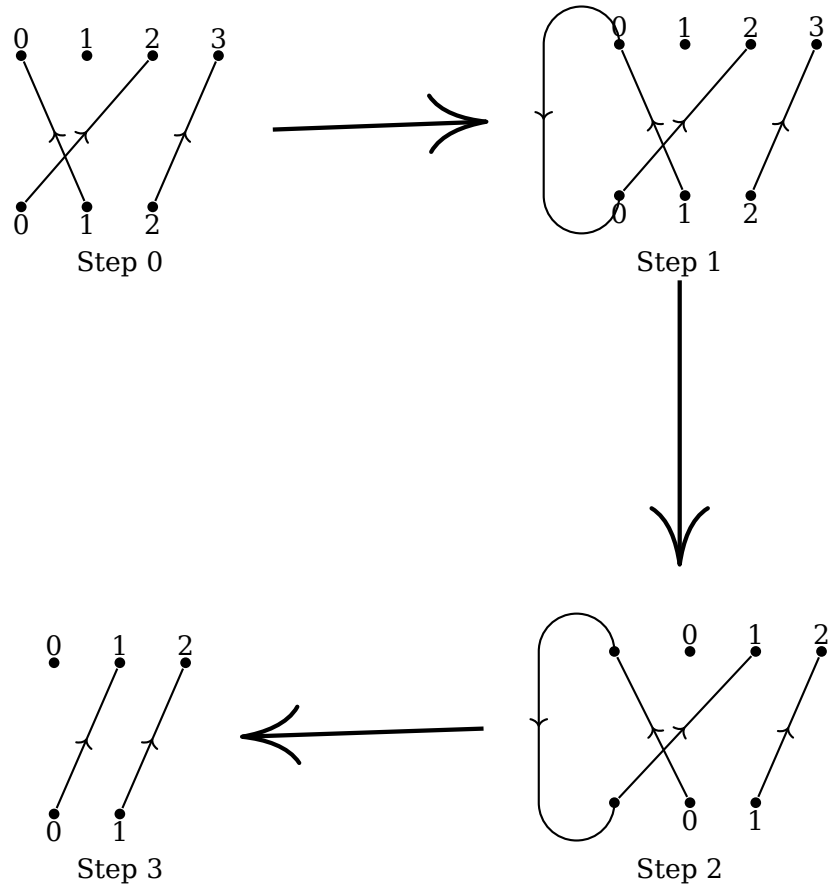


Figure 3.3: Step by step procedure for $Tr_1((2\ 0\ 3))$. Start with the graph of some injective function (step 0), add a cycle (step 1), shift the indices (step 2), and finally join source directly up with destination, (step 3) to get (12).

We implement this by making use of subtypes of $\mathbb{N}$ specifically we want to formally define the notion of a finite set with an element taken out. We do this by defining a 'minus operator':

```
record _\_ (A : ℕ) (a : Fin A) : Type where
  constructor _-_
  field
    val : Fin A
    ne : a ≢ val
open _\_
```

The idea is that we will use this definition to TODO

It is simply a pointed finite set, made up of a selected element `a : Fin A`, such that each instance has a value `val : Fin A` paired with a

proof of distinctness with `a: ne : a ≢ val`. We then prove some simple properties:

```
s_–0 : {A : ℕ} (a : Fin A) → suc A ∖ fzero
s a –0 = fsuc a – fzero≢fsuc {i = a}

0–s_ : {A : ℕ} (a : Fin A) → suc A ∖ fsuc a
0–s a = fzero – fsuc≢fzero {i = a}
```

`s a –0` removes 0 from Fin (suc A), leaving `fsuc a`, while `0–s a` removes a successor element `fsuc a`, leaving 0. Next we construct successor and predecessor functions on minus sets:

```
∖-suc : {A : ℕ} {a : Fin A} → A ∖ a → suc A ∖ fsuc a
∖-suc {suc A} (b – a≢b) = fsuc b – ≢cong fpred a≢b

∖-pred : {A : ℕ} {a : Fin A} → (b : suc A ∖ fsuc a) → (val b ≢ fzero) → A ∖ a
∖-pred {suc A} (fzero – a≢b) 0≢0 = absurd (0≢0 refl)
∖-pred {suc A} (fsuc b – a≢b) _ = b – ≢cong fsuc a≢b
```

Next we define insert and delete operations. Insert maps a finite set to a finite set one larger, but with a hole. Delete on the other hand goes in the opposite direction, mapping a finite set with a hole, to a finite set one smaller. Both maps are bijections and respect order, although that isn't proven here.

```
ins : {x : ℕ} → (a : ⟦ suc x ⟧) → ⟦ x ⟧ → (suc x ∖ a)
ins {suc x} fzero b = fsuc b – fzero≢fsuc
ins {suc x} (fsuc a) fzero = fzero – fsuc≢fzero
ins {suc x} (fsuc a) (fsuc b) =
  fsuc c – λ sa≡sc →
    let a≡c = fsuc-injective {suc x} {a} {c} sa≡sc
    in ne (ins a b) a≡c
  where
    c = val (ins a b)

del : {x : ℕ} → (a : ⟦ suc x ⟧) → (suc x ∖ a) → ⟦ x ⟧
del {ℕ.zero} fzero (fzero – 0≢0) = absurd (0≢0 refl)
del {suc x} fzero (fzero – 0≢0) = absurd (0≢0 refl)
del {suc x} fzero (fsuc b – a≢b) = b
del {suc x} (fsuc a) (fzero – a≢b) = fzero
del {suc x} (fsuc a) (fsuc b – a'≢b') =
  fsuc (del {x} a (b – ≢cong fsuc a'≢b'))
```

3.5.11 Definition of `pred`

We are now ready to define `Pred`. First fix a function `f'` with domain `suc x` and range `suc y`. The pred operator cannot apply if either the domain or range is empty, since it subtracts off an element each time.

Our approach will be to chain maps, inserting a hole, mapping it with `f'`, swapping that location with 0 and finally removing the bubble.

```
module Pred {x y : ℕ} (f' : [ suc x ↣ suc y ]) where
  open _∖_
```

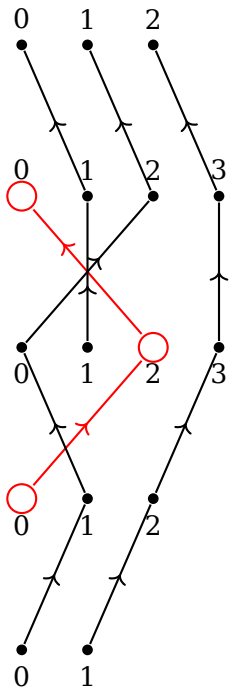


Figure 3.4: Chain of compositions to implement the trace of $(2\ 0\ 3)$. From bottom to top: (a) 'ins fzero' inserts a hole at location f0, (b) The the function is applied and the hole is traced through from 0 to 2, (c) Next the hole and position 0 are swapped.

First we split f' into its components:

```
f = fst f'
f-inj = snd f'
```

Now we construct the function-part of `pred`, this works as follows: Given an input `i`, we shift it up to `fsuc i`, then, using the fact that `f` is injective, we show that `f (fsuc i)` is not mapped to the output of the trace path.

```
g : ⟦ x ⟧ → ⟦ y ⟧
g i =
  let f0≠fsi : f fzero ≠ f (fsuc i)
      f0≠fsi f0≡fsi = fzero≠fsuc (f-inj fzero (fsuc i) f0≡fsi)
  in del (f fzero) (f (fsuc i) — f0≠fsi)
```

Next we prove injectivity, by chaining injectivity proofs. `di` is a proof that del is injectivity

```
g-inj : is-injective g
g-inj b₁ b₂ gb₁≡gb₂ =
  let
    (c₁ — z≠c₁) = s b₁ —0
    (c₂ — z≠c₂) = s b₂ —0
  in
  fsuc-injective {i = b₁} {j = b₂}
```

```
    (f-inj c₁ c₂
      (del-inj
        (f fzero)
        (f c₁ - λ fz≡fc₁ → z≢c₁ (f-inj fzero c₁ fz≡fc₁))
        (f c₂ - λ fz≡fc₂ → z≢c₂ (f-inj fzero c₂ fz≡fc₂))
        gb₁≡gb₂))
```

Finally we construct the injective function `g'`, and use a module trick to generate the operation `pred` from the previous construction of `g'` from `g'`.

```
g' : [ X ↣ Y ]
g' = g , g-inj

open Pred using () renaming (g' to pred) public
```

The operator `trace` comes from repeating `pred A` times.

```
trace : {X Y : ℕ} (A : ℕ) → (f : [ A + X ↣ A + Y ]) → [ X ↣ Y ]
trace zero f = f
trace (suc A) f = trace A (pred f)
```

This definition turned out to be cumbersome to work with as it required explicitly transporting the proof of distinctness around.

I saw that lemmas were difficult to prove and saw that an alternative approach using inductive types.

3.6 `Inj` Representation

This section documents the construction of injective functions on finite sets using an inductive approach. This contrasts the `InjFun` representation from section 3.5 that uses a depdendent sum of a function and its proof of injectivity. Below we will define `Inj` and build up from basic operations on `Fin` up to a trace operator.

3.6.1 Definition of `Inj`

I have taken the approach of using a basic inductive definition for injective functions. This was because the previous way of doing it was messy, and ultimately hid the true behaviour that I wanted to extract with a trace. This meant that all of the proofs relied on a long chain of isomorphisms that weren't strong enough to capture the behaviour that we cared about, namely adding and removing links to modify a funciton to build a trace.

Additionally carrying around the proof meant they had to be modified together, and may have been the reason I was experiencing performance reduction when type checking Agda.

I noticed that I wasn't getting the benefit I expected from all of these abstractions and that ultimately proving these isomorphisms were distracting from the main aim which is ensure that a trace structure is formed from Virtual Sets. I do think this method could have worked if I had enough time, but the problem is that I didn't have the time to spare.

In the aid of this simplicity, I decided to switch to the following structure to represent injective Fin funcitons:

```
data Inj : ℕ → ℕ → Type where
  nul : ∀ n → Inj 0 n
  inc : ∀ {m n} → (b : Fin (suc n))
      → (inj : Inj m n)
      → Inj (suc m) (suc n)
```

This replaces the previous definition:

```
[_↣_] : ℕ → ℕ → Type
[ X ↣ Y ] = Σ (Fin X → Fin Y) is-injective
```

The way it works is that `nul` introduces an empty function from `Fin 0` to some `Fin n`. Then each subsequent `inc` adds a single link shifting the domain and codomain such that it is impossible to construct a non-injective function.

I represent these functions as vectors of finite values, where the position corresponds to each domain element, and the number is what that element is mapped to. For example, let's introduce compact notation to represent `Inj` functions.

We present the same function as an example in section 3.5.1, represented in figure 3.5.

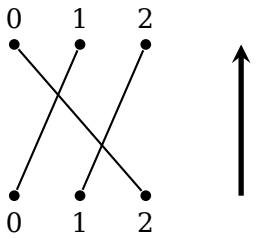


Figure 3.5: Plot of $(1\ 2\ 0)$. The bottom row is the domain and the top row is the codomain. The arrow indicates the direction from domain to codomain.

In represented as an `Inj`, we build up from `nul 0` adding edges as we go. We call `0` the 'slack' of the function, which is always equal to the size of the codomain minus the domain. `inc`, `insert` and `remove` operations all preserve the slack as we will see. `$` in the below is the function application operator, used to make function application right-associative, saving the use of parentheses.

```
f : Inj 3 3
f = inc f1 $ inc f1 $ inc f0 $ nul 0
-- f = inc f1 (inc f1 (inc f0 (nul 0)))
```

This is read right to left. First we start with `nul 0` which is an empty function with an empty codomain. The difference between the size of the domain and the codomain is specified entirely by this `nul` funciton.

0. Starting with `nul 0` means we having a slack of 0, which means we are construction a bijection, although this needs to be proven.

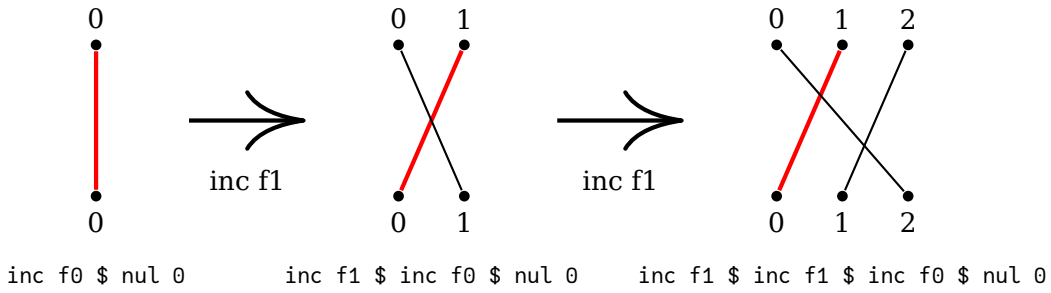


Figure 3.6: Construction of $(1\ 2\ 0)$ starting from `inc f0 $ nul 0`.

In a sense, it is the measure of how close to a surjection the funciton is. Since the funciton is injective by construction, we know that the size of the range is equal to the range. So the value given to nul is the difference between the size of the codomain and the domain. This value remains constant as links are added.

1. Next we add a single link from f0 to f0. This is the only choice we have at this stage because we're creating a function from Fin 1 to Fin 1 from `nul 0 : Fin 0 → Fin 0`.
2. The second link we add gives us two choices. We can either be parallel to the first link or cross over it. (`inc f0 $ inc f0 $ nul 0`) is two parallel links. (`inc f1 $ inc f0 $ nul 0`) is two crossing links. Note that these are the only two bijections from Fin 2 to Fin 2 available. We choose to cross in this example.
3. Finally the third link can either cross over both (`f2`), cross over just one (`f1`), or not cross at all (`f0`). We choose the middle option and end up with a cycle. Note that after nul 0, we three choices, $1 \times 2 \times 3 = 3!$. Every inc that is added increases the type of the Fin added by 1. Therefore, starting with `nul 0` to make `Inj m m`, there are m! options, which indicates that we're representing all possibilities. (Though it still needs to be checked that there is only one way to construct each function.)

The general formula for the number of injective finite functions between any two sets of size m and n is:

$$\frac{n!}{(n-m)!}$$

This has the nice properties that all constructions are necessarily injective, since it is impossible to construct two outputs that are the same, for example (0 0) : Fin 2 cannot be constructed. This is because each insertion 'splices' the output domain, which means shifting all links to all codomain elements that are greater than or equal to the new link's codomain element. We'll introduce splice operations in the next section.

3.6.2 'Splice' operations on `Fin`

In the definition of `Inj`, we have taken the approach of using a basic inductive definition for injective functions. This was because the dependent sum representation (`InjFun`) of doing it was messy, and ultimately hid the true behaviour that I wanted to extract with a trace. This meant that all of the proofs relied on a long chain of isomorphisms that weren't strong enough to capture the behaviour that we cared about, namely adding and removing links to modify a funciton.

Additionally carrying around the proof meant they had to be modified together, and may have been the reason I was experiencing performance reduction when type checking Agda.

I noticed that I wasn't getting the benefit I expected from all of these abstractions and that ultimately proved these isomorphisms were distracting from the main aim which is ensure that a trace structure is formed from Virtual Sets. I do think this method could have worked if I had enough time, but the problem is that I didn't have the time to spare.

The first function that needs to be written for this definition to be usable is `apply`. Specifically from the construction of Inj function, we need to be able to every value from the domain into the codomain. To see how to construct this, suppose we have an `f : Inj m n` with m non-zero. For an example, consider the function,

```
f = inc f3 $ inc f0 $ inc f1 $ inc f0 $ inc f0 $ nul 0
```

This is diagrammed in Figure 3.7.

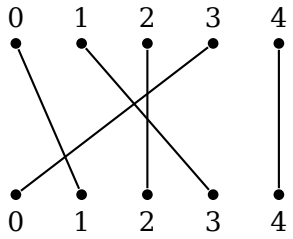


Figure 3.7: Plot of `f = inc f3 $ inc f0 $ inc f1 $ inc f0 $ inc f0 $ nul 0`.

For the `fzero` case, notice that `f f0` can just be read of as the target of the last inserted edge, it's just what the last inserted link associates to. For example, since we have `f ≡ inc f3 _`, we know that the first link we read off is $(0, 3)$. Thus we can fill in that case,

```
apply : ∀ {m n} → Inj m n → Fin m → Fin n
```

Let `g` stand for the the remainder of function (with `inc f3 $` removed), which is diagrammed suggestively for easier comparison with f in figure 3.8. Then by construction, we have `f = inc f3 g`.

`fsplice b` maps `Fin x` to `Fin (suc x)` in such a way that `b` is not in the codomain. Essentially it increments all values equal or greater

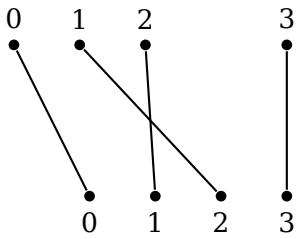


Figure 3.8: Plot of `g = inc f0 $ inc f1 $ inc f0 $ inc f0 $ nul 0`.

than b, keeping the values less than `b` unchanged. This is visualized in Figure 3.9. and defined below:

```
fsplice : ∀ {x} → Fin (suc x) → Fin x → Fin (suc x)
fsplice fzero a = fsuc a
fsplice (fsuc b) fzero = fzero
fsplice (fsuc b) (fsuc a) = fsuc (fsplice b a)
```

We call the first argument the 'pivot'. It is necessarily true that `fsplice b a ≢ a`, which we show below. We will use this to recurse in the definition of apply. The base case just reads off the target, which is the first `inc` expanded. The second case, recurses on `g`, and then uses the splice to ensure there is space for `b` in the codomain. This is what ensures that all `Inj` functions are injective, since it's not possible for fsplice to map an element to the pivot:

```
fsplice≢b : ∀ {x} → (b : Fin (suc x)) → (a : Fin x) → fsplice b a ≢ᶠ b
fsplice≢b fzero a = fsuc≢fzero
fsplice≢b (fsuc b) fzero = fzero≢fsuc
fsplice≢b (fsuc b) (fsuc a) ne =
  let rec≢b = fsplice≢b b a
  in rec≢b (≈fsuc-injective ne)
```

We now have what we need to finish `apply`. To get the successor case, suppose we have `suc n`, we need to recurse on `g`. This gives us the correct values for all outputs that are less than `b`, in this case `f3`. The final operation we need, we call a `fsplice`, one of a family of operations on Fin that provide utilities for operating on Fin types. Putting that together we obtain:

```
apply (inc b g) fzero = b
apply (inc b g) (fsuc a) = fsplice b (apply g a)
```

Next we define a function `fcross` (see Figure 3.10). is in some ways the opposite to `fsplice`. The idea is that given a pivot point `b`, it creates a funciton that `merges' the adjacent domain elements located at`b`and`b + 1'. This is useful when you want to swap the order of insertion of two values, the one moving out expands its domain by one and the pivot location encodes the relative order of two values. While the one moving further into the structure 'loses' information about the order of the two values.

```
fcross : ∀ {x : ℕ} → (b : Fin x) → (a : Fin (suc x)) → Fin x
fcross fzero fzero = fzero
```

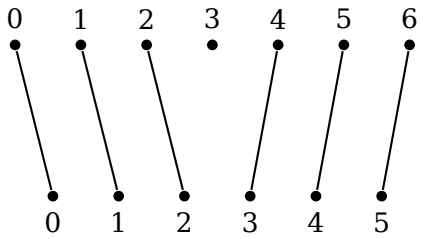


Figure 3.9: Plot of `fsplice 3` on `x` = 5.

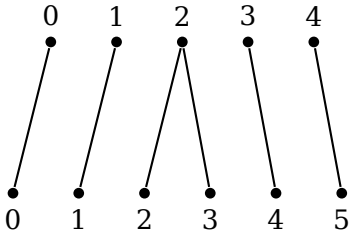


Figure 3.10: Plot of fcross 2 on x = 4.

```
fcross (fsuc b) fzero = fzero
fcross fzero (fsuc a) = a
fcross (fsuc b) (fsuc a) = fsuc (fcross b a)
```

The last major operation is called `fjoin`. The idea is that a join is exactly the opposite to `fsplice`. This will be used to construct an inverse of function. We implement by making use of 'trichotomy' on the pivot 'b' with the assumption that the input is distinct from the pivot. Then if `a` is less than the pivot, we restrict the domain with `fin-restrict-<` but keep the value unchanged, and if it's larger, we decrease it by one. It is split up into two functions for ease of working with `fjoin`. If the function was part of a `where` clause then it would be inaccessible outside the function, and if it were split on a `with` clause then it acts the same as a where clause, creating an inaccessible funciton. This is a limitation of the current version of Agda.

```
fjoin-cases : ∀ {x : ℕ} → (b a : Fin (suc x)) → a ≠ᶠ b
                → Trichotomyᶠ a b → Fin x
fjoin-cases b a a≠b (flt a<b) = fin-restrict-< a a<b
fjoin-cases b a a≠b (feq a≈b) = absurd (a≠b a≈b)
fjoin-cases b (fsuc a) a≠b (fgt b<a) = a

fjoin : ∀ {x : ℕ} → (b a : Fin (suc x)) → a ≠ᶠ b
          → Fin x
fjoin b a a≠b = fjoin-cases b a a≠b (a ≟ᶠ b)


fcross0≡0 : ∀ {m} → (a : Fin (suc m))
          → fcross a fzero ≡ fzero
fcross0≡0 fzero = refl
fcross0≡0 (fsuc a) = refl
```

```
fcross0s≡pred : ∀ {m} → (a : Fin (suc m))
              → fcross f0 (fsuc a) ≡ a
fcross0s≡pred a = refl


fcross-fcross-fsplice
  : ∀ {m} → (b : Fin (suc (suc m))) (c : Fin (suc m))
  → (fcross (fcross c b) (fsplice b c)) ≡ c
fcross-fcross-fsplice fzero fzero = refl
fcross-fcross-fsplice fzero (fsuc c) = refl
fcross-fcross-fsplice (fsuc b) fzero = fcross0≡0 b
fcross-fcross-fsplice {m = suc m} (fsuc b) (fsuc c) =
  cong fsuc (fcross-fcross-fsplice b c)
```

A key lemma is the injectivity of fjoin. The proof is non-trivial requiring significant pattern matching. This is a general limitation I have found with this approach compared with the `InjFun` approach.

```
fjoin-isInjective
  : {n : ℕ} → (a b c : Fin (suc n))
  → (b≠a : b ≠ᶠ a) → (c≠a : c ≠ᶠ a)
  → fjoin a b b≠a ≡ fjoin a c c≠a → b ≡ c
```

<!-

```
fjoin-isInjective fzero fzero c 0≠0 c≠0 q = absurd (0≠0 ≈refl)
fjoin-isInjective fzero (fsuc b) fzero b'≠0 0≠0 q = absurd (0≠0 ≈refl)
fjoin-isInjective {suc n} fzero (fsuc b) (fsuc c) b'≠a c'≠a q =
  cong fsuc p
  where p : b ≡ c
        p = b
              ≡⟨ sym (fjoin≡fcross f0 (fsuc b) b'≠a) ⟩
            fjoin f0 (fsuc b) b'≠a
              ≡⟨ q ⟩
            fjoin f0 (fsuc c) c'≠a
              ≡⟨ fjoin≡fcross f0 (fsuc c) c'≠a ⟩
            c ∎
fjoin-isInjective (fsuc a) fzero fzero 0≠a' _ q = refl
fjoin-isInjective {suc n} (fsuc a) fzero (fsuc c) 0≠a' c'≠a' q
  with (fsuc c) ≟ᶠ (fsuc a)
... | flt (<fsuc c<a) = ≈ᶠ→≡ $ ≈ᶠ-trans ≈fzero $ ≈ᶠ-trans (≡→≈ᶠ q)
                      $ ≈fsuc (fin-restrict-<-a≈a c c<a)
... | feq (≈fsuc c≈a) = absurd (c'≠a' (≈fsuc c≈a))
fjoin-isInjective {suc n} (fsuc a) fzero (fsuc (fsuc c)) 0≠a' c'≠a' q
    | fgt (<fsuc c>a) = absurd (fzero≠fsuc q)
fjoin-isInjective {suc n} (fsuc a) (fsuc b) fzero b'≠a' 0≠a' q
  with (fsuc b) ≟ᶠ (fsuc a)
... | flt (<fsuc b<a) = sym $ ≈ᶠ→≡ $ ≈ᶠ-trans ≈fzero $ ≈ᶠ-trans (≡→≈ᶠ (sym q))
                      $ ≈fsuc (fin-restrict-<-a≈a b b<a)
... | feq (≈fsuc b≈a) = absurd (b'≠a' (≈fsuc b≈a))
fjoin-isInjective {suc n} (fsuc a) (fsuc (fsuc b)) fzero b'≠a' 0≠a' q
    | fgt (<fsuc b>a) = absurd (fzero≠fsuc (sym q))
fjoin-isInjective {n = suc n} (fsuc a) (fsuc b) (fsuc c) b≠a c≠a q =
  cong fsuc (fjoin-isInjective a b c (≠fpred b≠a) (≠fpred c≠a) r)
  where
    r : fjoin a b (≠fpred b≠a) ≡ fjoin a c (≠fpred c≠a)
    r = fsuc-injective (
      fsuc (fjoin a b (≠fpred b≠a))
        ≡⟨ sym (fsuc-fjoin a b (≠fpred b≠a)) ⟩
```

```
      fjoin (fsuc a) (fsuc b) (≠fsuc (≠fpred b≠a))
        ≡⟨ fjoin-irrelevant (fsuc a) (fsuc b) (≠fsuc (≠fpred b≠a)) b≠a ⟩
      fjoin (fsuc a) (fsuc b) b≠a
        ≡⟨ q ⟩
      fjoin (fsuc a) (fsuc c) c≠a
        ≡⟨ fjoin-irrelevant (fsuc a) (fsuc c) c≠a (≠fsuc (≠fpred c≠a)) ⟩
      fjoin (fsuc a) (fsuc c) (≠fsuc (≠fpred c≠a))
        ≡⟨ fsuc-fjoin a c (≠fpred c≠a) ⟩
      fsuc (fjoin a c (≠fpred c≠a)) ∎)
```

3.6.3 Properties of `Inj`

Next we observe a couple of properties of `Inj`, as defined in section 3.6.1.

One key result that we desire is injectivity. We show this by matching values, and showing that they must be equal. In the first case, they are both `f0` so the result is immediate. When one value is `f0` and the other is a successor value, then we use the lemma that fsplice doesn't map to the pivot. In the final case we use the injectivity of fsplice and recurse.

```
Inj-isInjective : (f : Inj m n) → is-injective (apply f)
Inj-isInjective f fzero fzero fx≡fy = refl
Inj-isInjective (inc b f) fzero (fsuc y) fx≡fy =
  absurd (fsplice≠b b (apply f y) (≡→≈ᶠ (sym fx≡fy)))
Inj-isInjective (inc b f) (fsuc x) fzero fx≡fy =
  absurd (fsplice≠b b (apply f x) (≡→≈ᶠ fx≡fy))
Inj-isInjective (inc b f) (fsuc x) (fsuc y) fx≡fy =
  cong fsuc (Inj-isInjective f x y (fsplice-isInjective fx≡fy))
```

Another important result is extensionality of `Inj`, which is to say that `Inj` is entirely determined by its 'graph' (map from domain elements to codomain). Put another way, no two `Inj` structures act the same way when applied. This is an important property because it means to show equality, it's sufficient to prove it element-wise, which avoids expanding the inductive definition directly. It depends only on the injectivity of `fsplice` and some basic lemmas in the cubical library.

```
injExt : ∀ {m n} → (f g : Inj m n)
       → (∀ x → apply f x ≡ apply g x) → f ≡ g
injExt (nul _) (nul _) _ = refl
injExt (inc b f) (inc c g) f'x≡g'x =
  inc b f
    ≡⟨ cong (λ o → inc o f) (f'x≡g'x f0) ⟩
  inc c f
    ≡⟨ cong (inc c) f≡g ⟩
  inc c g ∎
  where
    fx≡gx : ∀ x → apply f x ≡ apply g x
    fx≡gx x =
      apply f x
        ≡⟨ (fsplice-isInjective
          ((f'x≡g'x (fsuc x))
          • sym (cong (λ o → fsplice o (apply g x)) (f'x≡g'x f0)))) ⟩
```

```
    apply g x ∎
  f≡g : f ≡ g
  f≡g = injExt f g fx≡gx
```

We prove with recursion. The base case is immediate, since there is only one `nul` function of each size. Now to suppose `∀ x → apply (inc b f) x ≡ apply (inc c g) x)`. We want to show `b ≡ c` and `f ≡ g`.

(`b ≡ c`): This is immediate from `apply (inc b f) f0 ≡ apply (inc c g) f0`.

(`f ≡ g`): We will do this by using induction by showing `∀ x → apply f x ≡ apply g x`. This is done by rewriting the assumed fact in terms of `fsplice`:

`fsplice b (apply f x) ≡ fsplice c (apply g x)`

then, using the fact `b ≡ c`, we can use the fact that fsplice is injective, to get, `apply f x ≡ apply g x` as we require. ∎

3.6.4 Elementary Operations on `Inj`

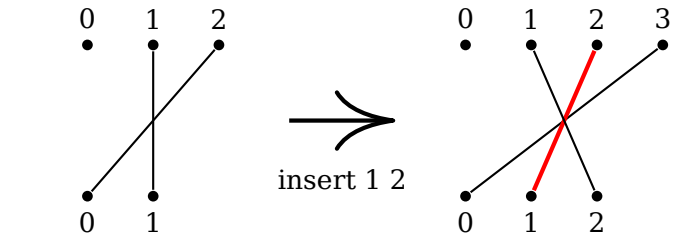
We define four elementary operations on our inductive injective function representation (`Inj`) that allow us to add, remove, and rearrange edges. These capture how the structure of injections can be manipulated while preserving injectivity.

Insert

The `insert` operation adds a new edge between a domain element and a codomain element, shifting the existing connections to preserve injectivity.

```
insert : ∀ {m n} → (a : Fin (suc m)) → (b : Fin (suc n))
       → (f : Inj m n) → Inj (suc m) (suc n)
insert fzero b f = inc b f
insert (fsuc a) b (inc c f) =
  inc (fsplice b c) (insert a (fcross c b) f)
```

Diagrammatically (Figure 3.11), inserting means creating a new edge, splicing it into the domain and codomain.



`inc f2 $ inc f1 $ nul 1` `inc f1 $ inc f2 $ inc f1 $ nul 1`

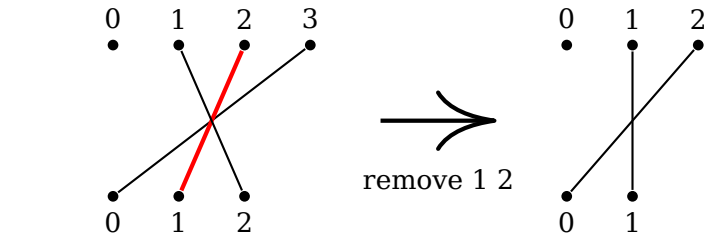
Figure 3.11: Example step of insert: adds a domain and codomain element, inserting the new edge.

Remove

The `remove` operation deletes one edge from the injection, shifting everything back down.

```
remove : ∀ {m n} → (a : Fin (suc m))
       → (f : Inj (suc m) (suc n)) → Inj m n
remove fzero (inc b f) = f
remove {suc m} {suc n} (fsuc a) (inc c f) =
  inc (fcross (apply f a) c) (remove a f)
```

This is illustrated in Figure 3.12: one edge is deleted and the codomain is compressed.



`inc f1 $ inc f2 $ inc f1 $ nul 1` `inc f2 $ inc f1 $ nul 1`

Figure 3.12: Remove deletes a domain/codomain edge, shifting as needed.

Bubble

The `bubble` operation allows us to "shift" all outputs around a pivot point in the codomain.

```
bubble : ∀ {m n} → (b : Fin (suc n))
       → (f : Inj m n) → Inj m (suc n)
bubble b (nul _) = nul _
bubble b (inc c f) =
  inc (fsplice b c) (bubble (fcross c b) f)
```

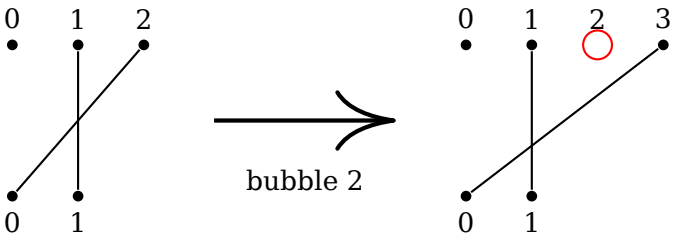
This operation is shown in Figure 3.13, where all the edges are redirected around the pivot `b`. Bubble is equivalent to composing `f` with an injective function

Excise

Finally, `excise` removes a domain element without shifting the codomain. It is implemented in terms of `remove` and `bubble`.

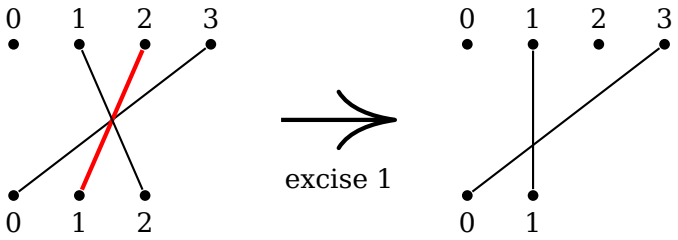
```
excise : ∀ {m n} → (a : Fin (suc m))
       → (f : Inj (suc m) (suc n)) → Inj m (suc n)
excise a f = bubble (apply f a) (remove a f)
```

As shown in Figure 3.14, excision removes a mapping while preserving the overall codomain mapping by bubbling around the target of the deleted edge. This is equivalent to first removing the edge then splicing at the target.



`inc f2 $ inc f1 $ nul 1` `inc f3 $ inc f1 $ nul 2`

Figure 3.13: Bubble creates a new unoccupied codomain slot at the highlighted position.



`inc f1 $ inc f2 $ inc f1 $ nul 1` `inc f3 $ inc f1 $ nul 2`

Figure 3.14: Excise removes highlighted domain/codomain edge, then bubbles its image to preserve codomain size.

3.6.5 Composition

From here we can define compose inductively, by seeing where each edge ends up under composition. `b ≡ apply (inc b f) f0`, so `apply (g ∘ʲʲ inc b f) f0 ≡ apply g b`, therefore the first link should be from f0 to `apply g b`. We then remove the link from the composition and recurse.

```
_∘ʲ_ : ∀ {l m n} (g : Inj m n) (f : Inj l m) → Inj l n
g ∘ʲ (nul _) = nul _
_∘ʲ_ {suc l} {suc m} {suc n} g (inc b f) = inc (apply g b) (remove b g ∘ʲ f)
```

If there is at least one edge, then it recurses on the first edge of the first funciton, tracing where it is applied to, then recursing on the composition of removing that edge from both graphs.

We want this resulting graph to respect the compatability of application with composition.

Lemma 1 (`apply-apply`)**.** *Given `f : Inj l m, g : Inj m n` then applying the composition should give you the same thing as applying them in sequence:*

`(f : Inj l m) (g : Inj m n) (a : ⟦ l ⟧) → apply g (apply f a) ≡ apply (g ∘ʲ f) a`

For all `a : Fin l`.

Proof: This is proven in the code in `VSet.Transform.Inj.Compose.Properties` as `apply-apply`. Although the result appears quite straight-forward conceptually, this required several sublemmas to prove, and a few hundred lines of Agda to formalize. We proceed by induction. In the `a = fzero` case, it is immediate from the definition of composition that `apply (g ∘ʲ inc b f) fzero ≡ apply g b`. The inductive case comes down to showing by equational reasoning that `apply g (fsplice b (apply f a)) ≡ fsplice (apply g b) (apply (remove b g ∘ʲʲ f) a)`. This is a non-trivial result, and is quite intricate.

Lemma 2 (`∘ʲ` is Assiciative)**.** *For all `f : Inj k l`, `g : Inj l m`, `h : Inj m n`',*

Then `h ∘ʲ (g ∘ʲ f) ≡ (h ∘ʲ g) ∘ʲ f`

We make use of extensionality that `∀ x → apply f x ≡ apply g x`, then `f ≡ g`.

Going back to associativity, it essentially comes down to showing `∀ x → apply (h ∘ʲ (g ∘ʲ f)) x ≡ apply ((h ∘ʲ g) ∘ʲ f) x`

which comes from repeatedly using `apply-apply` from both sides until the form below is reached. ∎

3.6.6 `Inj` Category Construction

Recall that a category is a mathematical structure that can be thought of as a directed grap with a certain composition on the edges (called morphisms) that is associative, and that each node (called objects) have an arrow from themselves to themselves that acts as an identity.

One of the goals of the project is to construct a category of injective functions on finite sets. If we choose to pick a canonical finite set for each cardinality, as we have been doing, then the objects of the category can be indexed by `ℕ` so we pick `ℕ` to be the set of objects. For morphisms we want injective functions, so we pick our inductive representation `Inj`. Now the rest is fairly straight-forward: - Composition is `_∘ʲ_`. - Identity morphisms are the identity inj functions.

```
idInj : ∀ {m} → Inj m m
idInj {zero} = nul zero
idInj {suc m} = inc fzero (idInj {m})
```

We have just proven that `_∘ʲ_` is associative, so we just need to check that `idInj` is a left and right identity, which follows from function extensionality, using `apply-apply`, and using the fact that `idInj` does in fact act as an identity function:

These are given by `∘ʲ-idR` and `∘ʲ-idL` for right and left identity respectively.

The final requirement is a peculuar feature of homotopy type theory, where we want to know that the types of arrows from one object to another (i.e., `Inj m n` for some `m n : ℕ`) have to be *sets*, meaning they have no deeper homotopic structure than mere identity of morphisms. Formally, the type of equalities between any two morphisms is a *proposition*.

Definition 13 (Set)**.** In homotopy type theory, a type X is a set if for every pair of elements $a, b \in X$, we have that their equality type `a ≡ b` is a proposition (defined below).

Definition 14 (Proposition)**.** A type X is a *proposition* if, for every pair of inhabitents $a, b \in X$, we have that `a ≡ b` is inhabited.

Lemma 3. *For every $n, m \in \mathbb{N}$, we have that `Inj m n` is a set.*

This result is given by `isSetInj` in `VSet.Data.Inj.Order`.

```
open Category

InjCat : Category _ _
InjCat = record
  { ob = ℕ
  ; Hom[_,_] = Inj
  ; id = λ {x} → idInj x
  ; _⋆_ = _∘⟨Inj⟩_
  ; ⋆IdL = ∘⟨Inj⟩-idR
  ; ⋆IdR = ∘⟨Inj⟩-idL
  ; ⋆Assoc = λ x y z → ∘⟨Inj⟩-assoc z y x
  ; isSetHom = isSetInj
  }


InjProdCat : Category _ _
InjProdCat = InjCat ×C InjCat


⊕-ob : InjProdCat .ob → InjCat .ob
⊕-ob (m , n) = m + n

⊕-hom : {x y : InjProdCat .ob} → InjProdCat [ x , y ]
      → InjCat [ ⊕-ob x , ⊕-ob y ]
⊕-hom (f , g) = f ⊕ g

⊕-id : {x : InjProdCat .ob}
     → ⊕-hom {x = x} {y = x} (InjProdCat .id)
     ≡ InjCat .id {x = ⊕-ob x}
⊕-id {(m , n)} =
  ⊕-hom {x = (m , n)} {y = (m , n)} (InjProdCat .id)
    ≡⟨ refl ⟩
  ⊕-hom {x = (m , n)} {y = (m , n)} (Id , Id)
    ≡⟨ refl ⟩
  Id {m} ⊕ Id {n}
    ≡⟨ Id⊕Id≡Id {m} {n} ⟩
  Id {m + n} ∎

⊕-seq : {x y z : InjProdCat .ob} (f : InjProdCat [ x , y ]) (g : InjProdCat [ y , z ])
      → ⊕-hom (f ⋆⟨ InjProdCat ⟩ g) ≡ (⊕-hom f) ⋆⟨ InjCat ⟩ (⊕-hom g)
⊕-seq {(l , l')} {(m , m')} {(n , n')} (f , f') (g , g') =
  ⊕-hom ((f , f') ⋆⟨ InjProdCat ⟩ (g , g'))
    ≡⟨ {!!} ⟩
  ⊕-hom (f ∘⟨Inj⟩ g , f' ∘⟨Inj⟩ g')
    ≡⟨ {!!} ⟩
  (f ⊕ f') ∘⟨Inj⟩ (g ⊕ g') ∎
```

```
tensorStr : TensorStr InjCat
tensorStr = record
  { _⊗_ = record
    { F-ob = ⊕-ob
    ; F-hom = ⊕-hom
    ; F-id = ⊕-id
    ; F-seq = ⊕-seq
    }
  ; unit = {!!}
  }
```

3.6.7 Defining Tensor on `Inj`

We now define a tensor product on `Inj`. We don't get as far as proving all of the coherence results, however due to time constraints, I wasn't able to finish the proof of the coherence results..

We start by defining a 'shift' operator, which is a bit like fshift, but applies to all result. The idea is that shift increases all of the output values, effectively shifting the window to the left.

```
shift1 : ∀ {m n} → (f : Inj m n) → Inj m (suc n)
shift1 (nul _) = nul _
shift1 (inc b f) = inc (fsuc b) (shift1 f)


shift : ∀ {m n} → (l : ℕ) → (f : Inj m n) → Inj m (l + n)
shift zero f = f
shift (suc l) f = shift1 (shift l f)
```

Next we define the tensor, as injecting in the left case, and shifting in the right case. and a unit.

```
infixl 30 _⊕_ -- \o+
tensor : ∀ {m m' n n'} → (f : Inj m m') → (g : Inj n n') → Inj (m + n) (m' + n')
tensor (nul m') g = shift m' g
tensor {n' = n'} (inc b f) g = inc (finject n' b) (tensor f g)
_⊕_ : ∀ {m m' n n'} → (f : Inj m m') → (g : Inj n n') → Inj (m + n) (m' + n')
f ⊕ g = tensor f g

𝟘 : Inj 0 0
𝟘 = nul 0
```

3.6.8 Associator on $\oplus$

```
open import Cubical.Data.Prod.Base
open import Cubical.Data.Sum.Base hiding (elim)
open import Cubical.Data.Nat.Base hiding (elim)
open import Cubical.Data.Nat.Order
open import Cubical.Data.Nat.Properties
open import Cubical.Data.List.Base hiding (elim; [_])
open import Cubical.Data.Maybe.Base hiding (elim)
open import VSet.Data.Fin.Base
open import VSet.Data.Fin.Order
open import VSet.Data.Fin.Splice
open import VSet.Data.Fin.Properties
```

```
open import VSet.Data.Inj.Base
open import VSet.Data.Inj.Order
open import VSet.Data.Inj.Properties
open import VSet.Transform.Inj.Inverse.Base

shift1-tensor : (f : Inj m m') (g : Inj n n')
              → (shift1 f) ⊕ g ≡ shift1 (f ⊕ g)
shift1-tensor {m} {m'} {n} {n'} (nul m') g = refl
shift1-tensor {m} {m'} {n} {n'} (inc b f) g =
  shift1 (inc b f) ⊕ g ≡⟨ refl ⟩
  inc (fsuc b) (shift1 f) ⊕ g
    ≡⟨ refl ⟩
  inc (finject n' (fsuc b)) (shift1 f ⊕ g)
    ≡⟨ cong₂ inc (finject-fsuc-reorder b) (shift1-tensor f g) ⟩
  inc (fsuc (finject n' b)) (shift1 (f ⊕ g))
    ≡⟨ refl ⟩
  shift1 (inc (finject n' b) (f ⊕ g))
    ≡⟨ refl ⟩
  shift1 (inc b f ⊕ g) ∎

shift-tensor-cast
  : (l' : ℕ) (f : Inj m m') (g : Inj n n')
  → (shift l' f) ⊕ g ≡ jcast refl (+-assoc l' m' n') (shift l' (f ⊕ g))
shift-tensor-cast {m} {m'} {n} {n'} zero f g =
  shift zero f ⊕ g ≡⟨ refl ⟩
  shift zero (f ⊕ g) ≡⟨ sym (jcast-loop _ _ _) ⟩
  jcast refl (+-assoc zero m' n') (shift zero (f ⊕ g)) ∎
shift-tensor-cast {m} {m'} {n} {n'} (suc l') f g =
  (shift (suc l') f) ⊕ g
    ≡⟨ refl ⟩
  (shift1 (shift l' f)) ⊕ g
    ≡⟨ shift1-tensor (shift l' f) g ⟩
  shift1 ((shift l' f) ⊕ g)
    ≡⟨ cong shift1 (shift-tensor-cast l' f g) ⟩
  shift1 (jcast refl (+-assoc l' m' n') (shift l' (f ⊕ g)))
    ≡⟨ {!!} ⟩
  jcast refl (cong suc (+-assoc l' m' n')) (shift1 (shift l' (f ⊕ g)))
    ≡⟨ refl ⟩
  jcast refl (+-assoc (suc l') m' n') (shift (suc l') (f ⊕ g)) ∎

-- jcast-reorder
--   : ∀ {m m' n n' : ℕ}
--   → (ϕ : ℕ → ℕ) (ψ : ℕ → ℕ) (η : {x y : ℕ} → Inj x y → Inj (ϕ x) (ψ y))
--   → (p : m ≡ m') (q : n ≡ n') (f : Inj m n)
--   → jcast (cong ϕ p) (cong ψ q) (η f)
--   ≡ η (jcast p q f)
-- jcast-reorder {zero} {zero} {n} {n'} ϕ ψ η p q (nul _) = {!!}
-- jcast-reorder {zero} {suc m'} {n} {n'} ϕ ψ η p q (nul _) = {!!}
-- jcast-reorder {suc m} {m'} {n} {n'} ϕ ψ η p q (inc b f) = {!!}

shift-tensor : (l' : ℕ) (f : Inj m m') (g : Inj n n')
              → (shift l' f) ⊕ g ≡ subst2 Inj refl (+-assoc l' m' n') (shift l' (f ⊕ g))
shift-tensor {m} {m'} {n} {n'} zero f g =
  shift zero f ⊕ g ≡⟨ sym (transportRefl (shift zero f ⊕ g)) ⟩
  transport refl (shift zero (f ⊕ g)) ≡⟨ refl ⟩
  subst2 Inj refl (+-assoc zero m' n') (shift zero (f ⊕ g)) ∎
shift-tensor {m} {m'} {n} {n'} (suc l') f g =
  (shift (suc l') f) ⊕ g
```

```
    ≡⟨ refl ⟩
  (shift1 (shift l' f)) ⊕ g
    ≡⟨ shift1-tensor (shift l' f) g ⟩
  shift1 ((shift l' f) ⊕ g)
    ≡⟨ cong shift1 (shift-tensor l' f g) ⟩
  shift1 (subst2 Inj refl (+-assoc l' m' n') (shift l' (f ⊕ g)))
    ≡⟨ sym (subst2-reorder Inj (λ x → x) suc shift1 refl
                           (+-assoc l' m' n') (shift l' (f ⊕ g))) ⟩
  subst2 Inj refl (+-assoc (suc l') m' n') (shift (suc l') (f ⊕ g)) ∎
```

module _ {l l' m m' n n' : ℕ} where α-p-dom : l + (m + n) ≡ (l + m) + n α-p-dom = +-assoc l m n

α-p-cod : l' + (m' + n') ≡ (l' + m') + n' α-p-cod = +-assoc l' m' n'

α-p : Inj (l + (m + n)) (l' + (m' + n')) ≡ Inj ((l + m) + n) ((l' + m') + n') α-p = cong₂ Inj (+-assoc l m n) (+-assoc l' m' n')

α-iso : Iso (Inj (l + (m + n)) (l' + (m' + n'))) (Inj ((l + m) + n) ((l' + m') + n')) α-iso = pathToIso α-p

α : Inj (l + (m + n)) (l' + (m' + n')) → Inj ((l + m) + n) ((l' + m') + n') α = Iso.fun α-iso

α⁻¹ : Inj ((l + m) + n) ((l' + m') + n') → Inj (l + (m + n)) (l' + (m' + n')) α⁻¹ = Iso.inv α-iso

assoc : {l l' m m' n n' : ℕ} → (f : Inj l l') (g : Inj m m') (h : Inj n n') → ((f ⊕ g) ⊕ h) ≡ transport (α-p {l} {l'}) (f ⊕ (g ⊕ h)) assoc {zero} {l'} {m} {m'} {n} {n'} (nul _) g h = (nul l' ⊕ g) ⊕ h ≡⟨ refl ⟩ (shift l' g) ⊕ h ≡⟨ shift-tensor l' g h ⟩ subst2 Inj refl (+-assoc l' m' n') (shift l' (g ⊕ h)) ≡⟨ refl ⟩ subst2 Inj (+-assoc zero m n) (+-assoc l' m' n') (nul l' ⊕ (g ⊕ h)) ∎ assoc {suc l} {suc l'} {m} {m'} {n} {n'} (inc b f) g h = (inc b f ⊕ g) ⊕ h ≡⟨ refl ⟩ (inc (finject m' b) (f ⊕ g)) ⊕ h ≡⟨ refl ⟩ inc (finject n' (finject m' b)) ((f ⊕ g) ⊕ h) ≡⟨ cong (λ ○ → inc ○ ((f ⊕ g) ⊕ h)) (finject-+ (suc l') m' n' b) ⟩ inc (subst (Fin ∘ suc) (+-assoc l' m' n') (finject (m' + n') b)) ((f ⊕ g) ⊕ h) ≡⟨ cong (inc (subst (Fin ∘ suc) (+-assoc l' m' n') (finject (m' + n') b))) (assoc f g h) ⟩ inc (subst (Fin ∘ suc) (+-assoc l' m' n') (finject (m' + n') b)) (subst2 Inj (+-assoc l m n) (+-assoc l' m' n') (f ⊕ (g ⊕ h))) ≡⟨ sym (subst2-inc-reorder (+-assoc l m n) (+-assoc l' m' n') (finject (m' + n') b) (f ⊕ (g ⊕ h))) ⟩ subst2 Injsuc (+-assoc l m n) (+-assoc l' m' n') (inc (finject (m' + n') b) (f ⊕ (g ⊕ h))) ≡⟨ refl ⟩ subst2 Inj (+-assoc (suc l) m n) (+-assoc (suc l') m' n') (inc b f ⊕ (g ⊕ h)) ∎

unassoc : (f : Inj l l') (g : Inj m m') (h : Inj n n') → (f ⊕ (g ⊕ h)) ≡ (α⁻¹ {l} {l'}) ((f ⊕ g) ⊕ h) unassoc {l} {l'} {m} {m'} {n} {n'} f g h = let α-p = α-p {l} {l'} {m} {m'} {n} {n'}
in (f ⊕ (g ⊕ h)) ≡⟨ sym (transport⁻Transport α-p (f ⊕ (g ⊕ h))) ⟩ transport (sym α-p) (transport α-p (f ⊕ (g ⊕ h))) ≡⟨ sym (cong (transport (sym α-p)) (assoc f g h)) ⟩ transport (sym α-p) ((f ⊕ g) ⊕ h) ∎

3.6.9 Tensor Category Construction (unfinished)

```
open Category

InjProdCat : Category _ _
InjProdCat = InjCat ×C InjCat
```

```
⊕-ob : InjProdCat .ob → InjCat .ob
⊕-ob (m , n) = m + n

⊕-hom : {x y : InjProdCat .ob} → InjProdCat [ x , y ]
      → InjCat [ ⊕-ob x , ⊕-ob y ]
⊕-hom (f , g) = f ⊕ g

⊕-id : {x : InjProdCat .ob}
     → ⊕-hom {x = x} {y = x} (InjProdCat .id)
     ≡ InjCat .id {x = ⊕-ob x}
⊕-id {(m , n)} =
  ⊕-hom {x = (m , n)} {y = (m , n)} (InjProdCat .id)
    ≡⟨ refl ⟩
  ⊕-hom {x = (m , n)} {y = (m , n)} (Id , Id)
    ≡⟨ refl ⟩
  Id {m} ⊕ Id {n}
    ≡⟨ Id⊕Id≡Id {m} {n} ⟩
  Id {m + n} ∎

⊕-seq : {x y z : InjProdCat .ob} (f : InjProdCat [ x , y ]) (g : InjProdCat [ y , z ])
      → ⊕-hom (f ⋆⟨ InjProdCat ⟩ g) ≡ (⊕-hom f) ⋆⟨ InjCat ⟩ (⊕-hom g)
⊕-seq {(l , l')} {(m , m')} {(n , n')} (f , f') (g , g') =
  ⊕-hom ((f , f') ⋆⟨ InjProdCat ⟩ (g , g'))
    ≡⟨ {!!} ⟩
  ⊕-hom (f ∘-j g , f' ∘-j g')
    ≡⟨ {!!} ⟩
  (f ⊕ f') ∘-j (g ⊕ g') ∎

tensorStr : TensorStr InjCat
tensorStr = record
  { _⊗_ = record
    { F-ob = ⊕-ob
    ; F-hom = ⊕-hom
    ; F-id = ⊕-id
    ; F-seq = ⊕-seq
    }
  ; unit = {!!}
  }
```

Chapter 4

Conclusion

4.1 Results

This dissertation formalizes two category constructions of the category of finite sets and injective funcitons, providing two foundations for further development.

- We defined two encodings of injective functions: `InjFun`, defined using a dependent sum; and `Inj`, defined inductively.
- We proved they both form a category
- We defined a tensor product for both of these.
- We sketched a construction of the monoid laws.
- We define a trace operator.
- We prove some basic properties (omitted from this report due to time).

All definitions were formalized in Cubical Agda, and most of the lemmas proven were formalized, though there are still significant holes which require further work to complete.

4.2 Discussion

The central research question (objective 6, and 7: whether the category **Inj** admits a trace satisfying the JSV axioms, which form the foundations for a full treatment of the **Int**-construction for compact closed completionremains only partially answered here. In practice, the proof burden fell on a long tail of elementary lemmas about finite sets, sums, and path manipulations. Midway through, I pivoted from the dependent-sum encoding to an inductive encoding in the hope of simplifying coherence proofs; however, both approaches incurred distinct overheads (transport management vs. structural bureaucracy), and neither reached a full monoidal (let alone traced) package within the project time-frame.

This project turned out to be much more challenging than initially anticipated. Neither of my approaches gave a simple way to go about the formalization of the result that injective maps on finite sets form a traced monoidal category. Progress was limited by the large amount of lemmas and sub-lemmas that need to be completed. After feeling like the progress had slowed with the depdent sum half-way though the project, I thought that starting again with an inductive approach would be significantly simpler. It turns out this assessment was wrong and both approaches had significant limitations that I was unable to overcome to complete the construction of even a symmetric monidal category let alone a traced monoidal category.

If I was doing this project again, I would shrink the scope significantly and commit to a single representation for the duration of the project. This would have been better for either approach if I'd stuck with it and proven the monoidal axioms before attempting traced monoidal categories and implementing the Int-construction [12].

Additionally I noticed that I hadn't allocated enough time to do the report justice, and I started it too late. I wasn't able to finish writing up the formalization of the tensor product on 'Inj', because I spent too much time trying to make progress on the code, to have something impressive to show at the end.

The project was very engaging, and I would be very interested in trying to complete the construction after I have submitted this dissertation.

Appendix A

References

Bibliography

[1] Guillaume Allais, Edwin Brady, Nathan Corbyn, Ohad Kammar, and Jeremy Yallop. Frex: dependently-typed algebraic simplification. *Proceedings of the ACM on Programming Languages*, 9(ICFP):1–29, 2025.

[2] Thorsten Altenkirch. From setoid hell to homotopy heaven? Talk slides, 2017. Functional Programming Laboratory, School of Computer Science, University of Nottingham.

[3] Steve Awodey. *Category Theory*. Oxford University Press, 2nd edition, 2010.

[4] John W Barrett and Bruce W Westbury. Spherical categories. *Advances in Mathematics*, 143(2):357–375, 1999.

[5] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2014.

[6] Cubical Agda: A standard library for Cubical Agda. `https://github.com/agda/cubical`, 2025. Version 0.9, compatible with Agda 2.8.0.

[7] Nils Anders Danielsson, Matthew Daggitt, Guillaume Allais, Andreas Abel, and contributors. Agda standard library. `https://github.com/agda/agda-stdlib`, 2014. Accessed: September 21, 2025.

[8] B. J. Day. Note on Compact Closed Categories. *Journal of the Australian Mathematical Society*, 24(3):309–311, 1977.

[9] Eelco Dolstra and NixOS contributors. Nix flakes. `https://github.com/NixOS/nix`, 2019. Accessed: September 21, 2025.

[10] Jason Z.S. Hu, Jacques Carette, and contributors. agda-categories: A Categories library for Agda. `https://github.com/agda/agda-categories`, 2025. Accessed September 2025.

[11] Simon Huber. Canonicity for cubical type theory. *Journal of Automated Reasoning*, 2017. Section added on propositional truncation; also see arXiv:1607.04156.

[12] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, April 1996.

[13] Tom Leinster. *Basic Category Theory*, volume 143 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 2014.

[14] Amélia Liao and contributors. 1lab: An online reference for homotopy type theory and cubical methods. `https://1lab.dev`, 2025. Accessed September 2025.

[15] Saunders Mac Lane. Coherence theorems for categories. *Annals of Mathematics*, 78:365–385, 1963.

[16] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 1998.

[17] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, 2020.

[18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[19] Christina O'Donnell. Virtual sets. `https://github.com/cdo256/virtualsets`, 2025. Accessed: Sept 21st, 2025.

[20] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton, 2013. Available online.

[21] The Coq Development Team. The coq proof assistant. Version 8.13, 2021.

[22] Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 1st International Conference on Homotopy Type Theory and Univalent Foundations*, 2016. See also: doi:10.1145/2854065.2854076.

[23] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31:e8, January 2021.

[24] Mark Williams. Virtual sets. Preprint, March 2025.

Appendix B

Agda Source Listings