The HoTT Game

Release 0.1

Ken Lee, Joseph Hua

Aug 28, 2023

CONTENTS

The Homotopy Type Theory (HoTT) Game is a project written by mathematicians for mathematicians interested in HoTT and no experience in proof verification, with the aim of introducing cubical agda as a tool for trying out mathematics in HoTT.

To get started with the HoTT Game, go to Getting Started.

This game was created by Joseph Hua, Ken Lee, and Bendit Chan.

CHAPTER

ONE

GETTING STARTED

1.1 The HoTT Game

The Homotopy Type Theory (HoTT) Game is a project written by mathematicians for mathematicians interested in HoTT and no experience in proof verification, with the aim of introducing cubical agda as a tool for trying out mathematics in HoTT. This page will help you get the Game working for you.

1.1.1 Agdapad

The HoTT Game can be played entirely in your browser using Agdapad. If you want a quick start to the game without installing anything, you can go straight to the site, create your own `agda` session, and start playing.

Specifically, once you open an `agda` session, you should see a welcome page (which contains useful information). Around the top left of the screen there is a folder icon. Click on the folder icon, and open the directory `TheHoTTGame`. This contains everything you need for the game.

Many thanks to Ingo Blechschmidt for incorporating the game into `Agdapad`.

1.1.2 Installing Agda and the Cubical Agda library

A more long-term setup for using `agda` would be to install it locally. If you would like detailed instructions on how to install agda and a supportive text editor then we recommend you follow instructions on *Installation*. More general instructions follow:

Our Game is in `agda`, which can be installed following instructions on their site. It is recommended that you use `agda` in the text editor emacs, specifically we recommend doom emacs, but it can also work in atom and vs-code.

Once you have `emacs` and `agda`, clone the cubical library repository (version 0.3) and make sure `agda` knows where your copy of the cubical library is by following instructions on the library management page. In short: locate (or create) your `libraries` file and add a line

```
the-directory/cubical.agda-lib
```

to it, where `the-directory` is the location of `cubical.agda-lib` on your computer.

Get the HoTT Game by cloning our repository into a folder and then making sure that `agda` knows where the HoTT Game is by adding a line

```
the-directory/TheHoTTGame.agda-lib
```

to your `libraries` file as above.

Try opening `1FundamentalGroup/Quest0.agda` in Emacs and do `Ctrl-c Ctrl-l`. Some text should be highlighted, and any `{!!}` should turn into `{ }`.

1.1.3 Where to start?

You can start with trinitarianism if you are interested in how logic, type theory and category theory come together as different ways to view the same thing. If you are more interested in homotopy theory, try *Fundamental Group of the Circle* where we show that the fundamental group of S^1 is . We recommend starting with *Fundamental Group of the Circle* for intuition, then going to trinitarianism.

1.1.4 How to start?

To start with *Fundamental Group of the Circle* (for example), go to *Quest 0 - Working with the Circle* and follow the instructions there. Any `agda` should happen in `Agdapad` or your local copy of the repository.

1.1.5 Emacs issues

If you can't figure out `emacs` or forget some command, then try consulting the *Emacs and Unicode Commands* page.

1.1.6 Special thanks

We would like to thank Kevin Buzzard for introducing us to formalization, making the Natural Numbers Game for learning `lean` (which inspired this project), and helping the game gain publicity. We would also like to thank all those involved in the HoTT community and writing the HoTT book, especially Steve Awodey, who helped us post about the Game on homotopytypetheory.org. Lastly we thank all those who are trying out the game and giving us feedback. If you would like to get involved, simply open and issue and let us know what you would like to do.

1.2 Installation

1.2.1 Overview

To get things up and running you will need five things

1. `agda` installed on your computer. This is what will check if your code makes sense. (Automatic if using the `Nix` installation)
2. A text editor, for example `doom emacs`, `atom`, `vs-code`, or `vim`. This is an environment for you to edit files in.
3. support for `agda2-mode` or `agda-mode` in your text editor. This should do syntax highlighting for your code (pretty colours) and make sure the text editor has the right shortcuts.
4. A clone of the cubical library. (Automatic if using the `Nix` installation.)
5. A clone of the HoTT Game, which is our code.

There are many ways to install `agda`. On this page we will try to describe some ways to install it. There are roughly three ways:

- Using `Nix`. If you use windows this is probably the easiest and most rewarding method. It involves getting the `NixOS`, a linux operating system inside your computer, so you also get to try out linux.
- Installing `agda` and the cubical library yourself

- Using VS-code and getting the agdaLanguageServer

Text editors

Important

No matter the text editor you choose, you will need emacs installed somewhere on your computer, as agda-mode relies on emacs in the background.

Whilst we will assume you use doom emacs in our guides (since it is the hardest to get used to), there are other options :

- atom :

  Here is a set of instructions by Andrew Swan for getting agda working in atom.

- vs-code : Here is a set of instructions for getting agda working in vs-code (scroll down to installation). You might be able to skip steps 1, 2 and 3 by enabling agdaMode.connection.agdaLanguageServer in the settings. (We haven't tried this - feedback is welcome.)
- vim : Here is a set of instructions for getting agda working in vim. (We haven't tried this - feedback is welcome).

1.2.2 Installing agda

Here we give instructions for installing agda on each operating system. If you have specific advice / issues specific to your operating system then please let us know in issues. Another source for information is official installation guide, but our advice might be more relevant to you.

Debian and Ubuntu

Ubuntu already should have a version of emacs installed. If not, go to a terminal and type in

```
sudo apt-get install emacs
```

To get agda, go to a terminal and type in

```
sudo apt install agda-bin
```

Now you need to set up agda-mode (is this necessary if you can get agda-mode in emacs? - feedback welcome) :

```
sudo apt install agda-mode
```

followed by

```
agda-mode setup
```

You can check the version of agda by doing agda --version in the terminal.

MacOS

This will give you both `agda` and `agda-mode` at once.

- Open a terminal.
- We will directly clone the `agda` repo for development version. First use `cd` ("change directory") in the terminal to navigate to where you want to place the `agda` library. Then do the following

  ```
  git clone https://github.com/agda/agda.git
  ```

  This gets a copy of the `agda` repo.
- Go into folder of agda repo then do

  ```
  cabal update
  make install
  ```

  This will compile `agda` to make it usable.
- Once process is finished, you can check `agda` is installed and its version by doing the following in `terminal` :

  ```
  agda --version
  ```

This is all you need to get `agda` and `agda-mode`, now you just need a text editor.

Windows

We used powershell as the terminal, but others probably work too.

> **Warning:** Always use powershell as admin.

For the prerequisites

- install chocolatey: follow instructions on their page
- In (admin) powershell do (via chocolatey, cabal) - `choco install ghc` - `choco install cabal` - `cabal update` In order to make `cabal` see `ghc`, close and reopen the terminal before doing the next steps. You might want to also try `refreshenv` for this. - `cabal install happy` - `cabal install alex`

Now to install `agda`, first try using `cabal` by doing `cabal install make` in the terminal. If this works then go with "using cabal", if not then try "using stack"

- You should have installed `make` with `cabal install make` by this point, if not do so now.
- Directly clone the repo for development version. *You can choose where to put this* by navigating to some specific folder in the terminal and doing

  ```
  git clone https://github.com/agda/agda.git
  ```

- It should create a folder called `agda` (a copy of the github repo). You should do `cd agda` to go into that folder, then once you're in there do

  ```
  make install
  ```

  which installs `agda` using `make` (it says "run the file called `MAKEFILE` from the folder").
- Once installation is finished, try typing `agda --version` in powershell to check the version.

- Get stack using the installer here.
- Run `stack upgrade` in the terminal
- Doing `cabal get Agda` in the terminal will create a folder called `Agda-2.6.2` *where you are at in the terminal. You can choose where to put this* by navigating to some specific folder in the terminal using `cd FILENAME`.
- Once you have created this `Adgda-2.6.2`, go into it by doing `cd Agda-2.6.2`.
- In the folder `Agda-2.6.2`, there should be a file called `stack-9.0.1.yaml`. Now you can try doing `stack --stack-yaml stack-9.0.1.yaml install` in the terminal (when you're in the folder `Agda-2.6.2`) to run that file.
- Once installation is finished, try typing `agda --version` to check the version.

In either case we should have `agda` *and* `agda-mode`. So we should just need to get a text editor.

1.2.3 Installing `doom emacs`

Here we give instructions for installing `doom emacs` on each operating system. If you have specific advice / issues specific to your operating system then please let us know in issues.

Linux

We have experience difficulties with getting `doom` on `ubuntu` specifically, so you *might* be better off using *one of the other options*, in particular `atom` appears to work well. Try installing `doom emacs` according to the instructions on their github repository. A quick guide follows:

1. Go to a terminal and type in

```
git clone --depth 1 https://github.com/hlissner/doom-emacs ~/.emacs.d

~/.emacs.d/bin/doom install
```

You'll probably want to answer "yes" to the options unless you know better. We recommend you add `~/.emacs.d/bin` to your `PATH` so you can call doom directly and from anywhere; accomplish this by going to the file `~/.bashrc` located in your home directory (or `~/.zshrc` file if you use zsh as your shell) and adding the line `export PATH=$PATH:~/.emacs.d/bin` at the end.

This should give you `doom emacs`. You might need to restart your computer and or `emacs` to make sure everything works correctly.

MacOS

Make sure you have the right version of git.

Do the following in a terminal to get `doom emacs`.

```
# required dependencies
brew install git ripgrep

# optional dependencies but install them anyway
brew install coreutils fd

# Installs clang. This may take a long time.
xcode-select --install
```

(continues on next page)

(continued from previous page)

```
# For fonts
brew install fontconfig

# Installs emacs-mac wth sexy icon
brew tap railwaycat/emacsmacport
brew install emacs-mac --with-modules --with-emacs-sexy-icon

# Make an app link in Applications
ln -s /usr/local/opt/emacs-mac/Emacs.app /Applications/Emacs.app

# doom emacs
git clone https://github.com/hlissner/doom-emacs ~/.emacs.d
~/.emacs.d/bin/doom install

# so that you can use 'doom' anywhere
export PATH=”$HOME/.emacs.d/bin:$PATH”
```

This should give you `doom emacs`. You might need to restart your computer and or `emacs` to make sure everything works correctly.

Windows

NixOS and WSL2

If you came from the NixOS and WSL2 instructions then go to the *linux section*.

There are detailed instructions for getting `doom emacs` on windows here.

The advice given there for installing fonts *might not work*. If it doesn't work, try installing a font (for example Iosevka) by following these instructions. Then go to `.doom.d/config.el` and add the line (anywhere)

```
(setq doom-font (font-spec :family "Iosevka SS04" :size 18 :weight 'medium))
```

Here the font name is `Iosevka SS04`. You can also change the font size and weight.

Operating system specific issues

If you have specific advice or issues specific to your operating system then please let us know in issues.

1.2.4 Getting `agda2-mode` or `agda-mode` support for your text editor

If you have decided to use `doom emacs` then you can get `agda2-mode` inside `doom emacs` (details below). For other text editors, you must first install `agda-mode`, and then find the relevant ad-on to the text editor to support `agda-mode` (details below).

Getting agda2-mode on doom emacs

Here we install agda2-mode in Doom Emacs. Note that this is *not* agda itself, but syntax highlighting and shortcuts for agda.

- Do the shortcut M-x in doom emacs. (See Emacs Commands for how to do shortcuts in doom emacs.) A window should pop up where you can type things. Type in :

  ```
  package-install
  ```

  Press enter and type in agda2-mode.

- Now do the shortcut SPC f p. A selection of files should appear. Type in init.el and hit enter (RET).
- Now you are in init.el. Look for the lang section and uncomment agda. Save the file and close doom emacs using SPC q q. (If you came from the Nix installation guide replace agda with (agda +local) instead.)
- Open terminal. To make the configurations of doom emacs up to date, do

  ```
  doom sync
  ```

  If there are no errors, you should have agda2-mode in doom emacs.

Getting agda-mode on atom

1. In atom select

- Edit > Preferences (GNU/Linux)
- Atom > Preferences (macOS)
- File > Settings (Windows)

2. Select Install from the side menu.
3. Type agda into the search box.
4. Install the packages agda-mode and language-agda

1.2.5 Check the agda and agda-mode installations

Once you have installed agda, a text editor, and support for agda-mode in your text editor, you should test it.

Make a test.agda file anywhere you'd like.

- Open test.agda in doom emacs.
- Type in

  ```
  open import Agda.Builtin.Nat
  ```

- Use C-c C-l to load the file. An **Agda Information** window should pop up and if all goes well, there should be nothing in it.
- Use C-c C-d then enter Nat. The output in the agda info window should be Set.

Congratulations, you now have agda and can use emacs bindings for agda. However, you have nothing more than the builtin types. So we need to get the library.

1.2.6 Getting the cubical library

The HoTT Game currently requires the cubical-0.3 library. We walk through an *example* of an installation of the cubical-0.3 library. See the Agda documentation for more about libraries.

- Go here. Under 'version 0.3', download the 'Source Code' file in either formats zip or tar.gz.
- Open the 'Source Code' file. It should turn into a folder which contains a folder called 'cubical'. Choose a place for it to permanently stay, this can be anywhere you like.
- Rename the folder 'cubical' to 'cubical-0.3'. Inside it, there should be a cubical.agda-lib file with contents

  ```
  name: cubical-0.3
  include: .
  depend:
  flags: --cubical --no-import-sorts
  ```

  This is the file that tells agda "this is a library" when agda looks into this folder. You can place the folder (now) called cubical-0.3 anywhere you like. For the sake of this guide, let's say you put it in a place so that the path is LOCATION/cubical-0.3.

Now we need to tell agda this cubical-0.3 library exists, so that it will look for it when an agda file uses code from it.

- Open a terminal and do

  ```
  agda -l fjdsk Dummy.agda
  ```

- Assuming you don't already have an agda library called fjdsk, you should see an error message of the form

  ```
  Library 'fjdsk' not found.
  Add the path to its .agda-lib file to
    'BLAHBLAHBLAH/libraries'
  to install.
  Installed libraries:
    none
  ```

  The BLAHBLAHBLAH/libraries is where we tell agda of the location of libraries.

  Examples in common operating systems :

  - On linux this might look something like :

    ```
    /home/USERNAME/.agda/libraries
    ```

    where USERNAME is your username on your computer.

  - On MacOS this might look something like :

    ```
    /Users/USERNAME/.agda/libraries
    ```

    where USERNAME is your username on your computer.

  - On windows this might look something like :

    ```
    C:\Users\USERNAME\AppData\Roaming\agda\libraries
    ```

    where USERNAME is your username on your computer.

- Navigate to `home/USERNAME` or `Users/USERNAME` or `C:\Users\USERNAME\AppData\Roaming\agda` using `cd`.
- Do the following to see hidden files :

```
ls -la
```

- *If there is no* `.agda` (`agda` for windows) *folder*, *simply create one* by doing

```
mkdir .agda

(or mkdir agda for windows)
```

  If you do `ls -la` again, you should see `.agda` in the list.

- Go into that folder by doing

```
cd .agda
```

- Check the contents of `.agda` by doing `ls -la`. Create a file `libraries` if there isn't one already. Inside it, put

```
LOCATION/cubical-0.3/cubical.agda-lib
```

  Save the file and close it.

- Restart the terminal. Now do `agda -l fjdsk Dummy.agda` in the terminal again. This time the error message should be

```
Library 'fjdsk' not found.
Add the path to its .agda-lib file to
   'BLAHBLAHBLAH/libraries'
to install.
Installed libraries:
   cubical-0.3
      (LOCATION/cubical-0.3/cubical.agda-lib)
```

  Congratulations, `agda` is now aware of the existence of the `cubical-0.3` library.

1.2.7 Getting The HoTT Game

The HoTT Game is also an `agda` library so we need to repeat the above process for it.

- In a terminal, navigate to where you would like to put the HoTT Game, as with the cubical library it can go anywhere. (You can use `cd` to navigate folders.)
- Use `git clone https://github.com/thehottgame/TheHoTTGame.git`. This should copy the HoTT Game repository as a folder called `TheHoTTGame`. For the purposes of this guide, let's say you have put the HoTT Game in your computer at the path

```
LOCATION1/TheHoTTGame
```

  Inside it, you should see many files, one of which should be `TheHoTTGame.agda-lib`.

- Go back to `BLAHBLAHBLAH/libraries` and add the following line

```
LOCATION1/TheHoTTGame/TheHoTTGame.agda-lib
```

- In `terminal`, use `agda -l fjdsk Dummy.agda` again. The error message should now look something like

```
Library 'fjdsk' not found.
Add the path to its .agda-lib file to
  'BLAHBLAHBLAH/libraries'
to install.
Installed libraries:
  cubical-0.3
    (LOCATION/cubical-0.3/cubical-0.3.agda-lib)
  TheHoTTGame
    (LOCATION1/TheHoTTGame/TheHoTTGame.agda-lib)
```

- In Doom Emacs, open `TheHoTTGame/1FundamentalGroup/Quest0.agda` and do `C-c C-l` (`Control-c Control-l`). If all went correctly, the text should be highlighted and you should be ready to go. Congratulations, you can now play the HoTT Game.

1.2.8 Installing with Nix

Linux and MacOS

`Nixpkgs` maintains a set of `agda` libraries that can be added to a derivation managed by the nix package manager, see here for details. The file `shell.nix` in our repository contains a derivation that will add `emacs`, `agda`, the `agda standard library`, and `cubical agda` to your local nix store and subsequently to a local shell environment by adding these locations to your `PATH`.

However, because user configurations for `emacs` are mutable, it will not (easily) manage your (emacs configuration) dot-files, so we will use the underlying `emacs` provided by `nixpkgs` but install `doom emacs` normally in your local user's environment.

1. Install `doom emacs` (or whichever text editor you prefer) via the method described for your operating system *here*. (If you are on Windows with NixOS on WSL2 then you are a linux user for the rest of the installation and should do everything in a terminal inside NixOS.)
2. Get `agda2-mode` support to `doom` (or whichever editor you prefer) via the method described *above*.
3. Clone our repository into a folder by going to some directory using `cd` and doing

   ```
   git clone https://github.com/thehottgame/TheHoTTGame.git
   ```

   This can be done anywhere you like.
4. Install `Nix` (*not* `NixOS`) using following the guidance on the official site. We install the single-user version for linux (compare this with what is written on the official website):

   ```
   sh <(curl -L https://nixos.org/nix/install) --no-daemon
   ```

   If you are on MacOS this will be different, and if you are on Windows using NixOS then this should also be exactly what you need.
5. Open a terminal, and use *cd* to navigate to the folder `TheHoTTGame`, which was cloned before. In `TheHoTTGame`, do

   ```
   nix-shell
   ```

   It might be that you need to restart your computer for this to work, and you might need to wait a little bit for it to start working, it might stay blank for a while. Later booting of nix-shell should be faster than the first.

This should open up a Nix shell (inside your usual terminal), from which you can do all the usual things in a terminal and more. The above mentioned packages should automatically be loaded on your PATH. The above is all defined by the package set in shell.nix in the folder TheHoTTGame.

6. Each time you wish to use agda (in particular its libraries), you should do step 5 to load the requisite packages onto the PATH so that they can be found.
7. If you got doom, go back to .doom.d/init.el and make sure that instead of uncommenting ;; agda in the ;; lang, *replace* it with (agda +local) to tell doom to use the agda-mode version specified by the local environment. Once the file is saved, sync doom from within the nix-shell that was loaded above:

```
doom sync
```

8. You can now load the agda source code in this by starting doom from the nix-shell:

```
doom run .
```

Open the file 0Trinitarianism/Quest0.agda and tell agda-mode to load and check it by doing SPC m l (space, m and l, in that order.) If everything is configured correctly, you should get nice colors and any {!!} will become interactive holes to fill.

Windows

First have a read of the previous section for Linux and MacOS for an overview, since once you get NixOS with WSL2, you will be using a Linux operating system anyway.

1. Get WSL2 following instructions here. You might also like to follow a video guide. Reboot your system.
2. By default WSL2 will get ubuntu, which is fine, but is not the operating system we will use. We want to get NixOS, which we can do by following instructions in the quick start section of this github page. Reboot your system.
3. Reopen NixOS and follow the *rest of the installation instructions* as if you are a linux user.

1.3 Emacs and Unicode Commands

1.3.1 Agdapad

Agdapad uses emacs but not doom emacs, so only the agda shortcuts (below) are relevant.

1.3.2 Notation

- SPC means space bar
- C-x means Ctrl-x
- M-x means Alt-x for non-Macs and Option-x for Macs
- S-x means Shift-x
- RET means enter

Example: C-c C-l in Agda files is Ctrl-c, let go, Ctrl-l. For the input of unicode characters go to the end of this page or visit *this site <https://agda.readthedocs.io/en/latest/tools/emacs-mode.html#keybindings>*.

1.3.3 General Doom Emacs usage

The ‘ambient mode’ is called *evil mode* and follows vim-like bindings. The following commands are for *evil mode*:

- `SPC h b b` to look for bindings (keyboard shortcuts)
- `SPC f f` to find files. can use `TAB` for auto-completing paths and `Backspace` to go up a directory
- `h j k l` for left down up right
- `SPC b k` to kill ‘buffers’ (any little window is a buffer). In general `SPC b` gives you many options for buffers.
- `SPC w k` to kill unwanted windows (emacs can get split up into many windows) In general `SPC w` gives you many options for windows.
- `i` to go into *insert mode* (in insert mode you can insert text) and `ESC` or `C-g` to go back to *evil mode*.
- `C-_` to undo (be careful with this, undo can go too far; going into and out of insert mode is considered “one change” in *evil mode*, so undoing might undo a lot of changes made in *insert mode*).
- `r` to redo (be careful with this, redo can go too far).
- `SPC h '` to look up how to write a symbol. (Put your cursor on the symbol first.)

1.3.4 Agda usage

Important: To insert text in the `agda` file use `i` to enter *insert mode*. To escape *insert mode* do `ESC` or `C-g`. All the commands below should be done whilst in *insert mode*.

- Load : `C-c C-l` loads the file
- Check the goal : `C-c C-,` checks goal of the hole your cursor is in.
- Fill the goal : `C-c C-SPC` fills hole your cursor is in.
- Refine the goal : `C-c C-r` refines the hole your cursor is in.
- Case on `x` : `C-c C-c` does cases on `x`, where `x` is in the hole your cursor is in.
- Deduce : `C-c C-d` asks you to give it term / point `x`, it deduces the type / space that `x` belongs to
- Normalise : `C-c C-n` asks you to give it term / point `x`, it ‘reduces’ `x` to its ‘simplest (normalised) form’
- Combo : `C-c C-.` does `C-c C-,` and `C-c C-d`
- Looking up definitions : in `agdapad`, clicking on something with the wheel of your mouse looks up the definitions of that thing (try clicking on `Type` for example). In `doom emacs`, `M-SPC c d` looks up the definition of the thing you are hovering over.

You can find more commands for `agda` in `emacs` here.

1.3.5 Unicode commands

In general follow the guidance given above to learn unicode commands. However here are some commonly used ones to get you started

- insert `\to` for →
- insert `\==` for
- insert `\==n` for
- insert `\bot` for
- insert `\top` for
- insert `\neg` for ¬
- insert `\GS` or `\Sigma` for
- insert `\cong` for
- insert `\^` for superscript, e.g. `S\^1` for S^1
- insert `\bN` for and `\bZ` for
- insert `\.` for
- insert `\sqcup` for

You can find more common symbols here.

1.4 Getting Git on MacOS

On certain older versions of `MacOS` one needs to get the right version of `git`.

1.4.1 Check the version

Check if you have the *right version of ``git`* <gettingGit>`_. Macs come with `git` pre-installed. You can open `terminal` and type

```
git --version
```

to see what version of `git` you have. It is most likely outdated if you've never used `git` before.

1.4.2 Get the right version

To get the latest version visit this site .

To tell your computer to use the correct version of `git`, we need to do the following :

- Open `terminal` and do the following to bring yourself to your home "directory".

  ```
  cd
  ```

- Do the following to show all files in this "directory".

  ```
  ls -la
  ```

  Amongst these we are interested in a file called `.zprofile` or `.bash_profile` if your mac is older.

- Look at the top of your terminal window and you should see `zsh` or `bash` if you're on an older mac. This is the "shell" that your mac is using for `terminal`. If it is `zsh`,

  ```
  open .zprofile
  ```

  This should open the file `.zprofile` with `text editor`. Now add the following to the end of the file

  If you terminal was using `bash` instead, do

  ```
  open .bash_profile
  ```

  This should open `.bash_profile` with `text editor`. Now add the the following to the end of the file

  Once you've done this, save the files and close them.

- Restart `terminal` and do the following again

  You should see the version has now updated.

CHAPTER

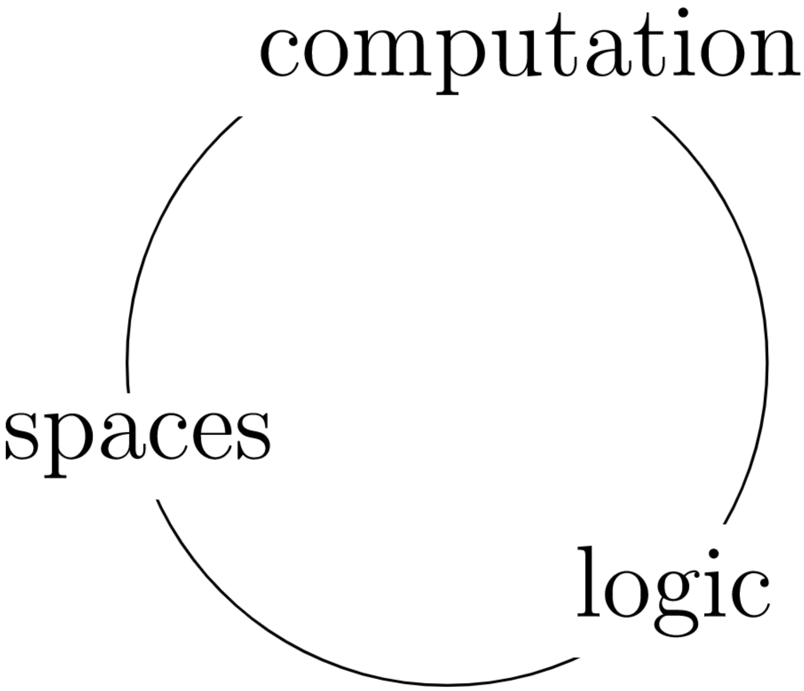
TWO

TRINITARIANISM

2.1 Overview

This arc introduces the setting “a place to do maths”. The “types” that will populated this “place” will have three interpretations:

- Proof theoretically, with types as propositions
- Type theoretically, with types as programs
- Category theoretically (geometrically), with types as objects (spaces) in a category (the space of spaces)



2.1.1 Terms and Types

Here are some things that we could like to have in a 'place to do maths'

- objects to reason about (E.g.)
- recipes for making things inside objects (E.g. `n + m` for `n` and `m` in naturals.)
- propositions to reason with (E.g. `n = 0` for `n` in naturals.)
- a notion of equality

In proof theory, types are propositions and terms of a type are their proofs. In type theory, types are programs / constructions and terms are algorithms / recipes. In category theory, types are objects (spaces) and terms are generalised elements (points in the space).

2.1.2 Non-dependent Types

- false / empty / initial object
- true / unit / terminal object
- or / sum / coproduct
- and / pairs / product
- implication / functions / internal hom

2.1.3 Dependent Types

- predicate / type family / bundle
- substitution / substitution / pullback (of bundles)
- existence / type / total space of bundles
- for all / type / space of sections of bundles

2.1.4 What is 'the Same'?

The last missing piece is a notion of *equality*. How HoTT treats equality is where it deviates from its predecessors.

2.2 Quest 0 - Terms and Types

There are three ways of looking at `A : Type`.

- proof theoretically, "`A` is a proposition"
- type theoretically, "`A` is a construction"
- geometrically / categorically, "`A` is a space and `Type` is the category of spaces".

A first example of a type construction is the function type. Given types `A : Type` and `B : Type`, we have another type `A → B : Type` which can be seen as

- the proposition "`A` implies `B`"
- the construction ways to convert `A` recipes to `B` recipes"

- the space of maps from A to B, i.e. maps from A to B correspond to points of A → B.
- internal hom of the category Type

To give examples of this, lets make some types first.

2.2.1 Part 0 - True / Unit / Terminal object

```
data  : Type where
  tt :
```

It reads “ is an inductive type with a constructor tt”, with interpretations

- is a proposition “true” and there is a proof of it, called tt.
- is a construction “top” with a recipe called tt
- is the singleton space
- is a terminal object: every object has a morphism into given by ·   tt

In general, the expression a :   A is read “a is a term of type A”, and has interpretations interpretations,

- a is a proof of the proposition A
- a is a recipe for the construction A
- a is a point in the space A
- a is a generalised element of the object A in the category Type.

The above tells you how we *make* a term of type . Lets see an example of *using* a term of type :

```
TrueToTrue :   →
TrueToTrue = {!!}
```

- enter C-c C-l (this means Ctrl-c Ctrl-l). Whenever you do this, agda will check the document is written correctly. This will open the *Agda Information* window looking like

  ```
  ?0 :   →
  ?1 :
  ?2 :
  ```

  This says you have three unfilled holes.
- Now you can fill the first hole.
- Navigate to the hole { } using C-c C-f (forward) or C-c C-b (backward)
- Enter C-c C-r. The r stands for *refine*. Whenever you do this whilst having your cursor in a hole, agda will try to help you.
- You should see x → { }. This is agda notation for x   { } and is called abstraction, think “ for let”.
- Navigate to the new hole
- Enter C-c C-, (this means Ctrl-c Ctrl-comma). Whenever you make this command whilst having your cursor in a hole, agda will check the *goal*.
- The goal (*Agda information* window) should look like

```
Goal:
————————————————————————————————————————————————————————————
x :
```

you have a proof/recipe/generalized element `x :`   and you need to give a proof/recipe/generalized element of

- Write the proof/recipe/generalized element `x` of   in the hole
- Press `C-c C-SPC` to fill the hole with `x`. In general when you have some term (and your cursor) in a hole, doing `C-c C-SPC` will tell `agda` to replace the hole with that term. `agda` will give you an error if it cant make sense of your term.
- The `*Agda Information*` window should now only have two unfilled holes left, this means `agda` has accepted your proof.

```
?1 :
?2 :
```

There is more than one proof (see `Quest0Solutions.agda`). Here is an important one:

```
TrueToTrue' :   →
TrueToTrue' x = { }
```

- Navigate to the hole and check the goal.
- Note `x` is already taken out for you.
- You can try type `x` in the hole and `C-c C-c`
- `c` stands for cases”. Doing `C-c C-c` with `x` in the hole tells `agda` to do cases on `x`”. The only case is `tt`.

One proof says for any term `x :`   give `x` again. The other says it suffices to do the case of `tt`, for which we just give `tt`.

The same”

Are these proofs “the same”? What is “the same”?

(This question is deep and should be unsettling. The short answer is that they are *internally* but not *externally* the same.)

Built into the definition of   is the way `agda` can make a map out of   into another type `A`, which we have just used. It says to map out of   it suffices to do the case when `x` is `tt`”, or

- the only proof of   is `tt`
- the only recipe for   is `tt`
- the only point in   is `tt`
- the only one generalized element `tt` in

Lets define another type.

2.2.2 Part 1 - False / Empty / Initial object

```
data  : Type where
```

This reads " is an inductive type with no constructors", with interpretations

- is a proposition "false" with no proofs
- is a construction "bot" with no recipes
- is the empty space
- There are no generalized elements of (it is a strict initial object)

We can make a map from to any other type, in particular into .

```
explosion :  →
explosion x = {!!}
```

- Navigate to the hole and do cases on `x`.

`agda` knows that there are no cases so there is nothing to do! (See `Quest0Solutions.agda`) Our interpretations:

- "false" implies "true". In fact the same proof gives "false" implies anything (principle of explosion)
- One can convert recipes of to recipes of . In fact the same construction gives a recipe of any other construction since there are no recipes of .
- There is a map from the empty space to the singleton space. In fact given any space `A` , there is a map from the empty space to `A`.
- is has a map into . This is due to being initial in the category `Type`.

2.2.3 Part 2 - The natural numbers

We can also encode "natural numbers" as a type.

```
data  : Type where
  zero :
  suc :  →
```

Our interpretations are:

- has no interpretation as a proposition since there are "too many proofs" - mathematicians classically don't distinguish between proofs of a single proposition. (ZFC doesn't even mention logic internally, but type theory does.) In this sense constructions are *proof relevant* types.
- As a construction :
  - is a type of construction
  - `zero` is a recipe for
  - `suc` takes an existing recipe for and gives another recipe for .
- Categorically : is a natural numbers object in the category `Type`. This means it is equipped with morphisms `zero :   →` and `suc :   →` such that given any → `A` → `A` there exist a unique morphism → `A` such that the diagram commutes:

$$\begin{tikzcd}
\top \arrow[r, "zero"] \arrow[d] & \mathbb{N} \arrow[r, "suc"] \arrow[d] & \mathbb{N} \arrow[d] \\
\top \arrow[r, "a_0"'] & A \arrow[r, "a_{suc}"'] & A
\end{tikzcd}$$

- Geometrically : is a space with a point `zero` and for every point `n` in , there is another point `suc n` in .

To see how to use terms of type , i.e. to induct on , go to *Quest 1 - Dependent Types*.

2.2.4 Part 3 - Universes

You may have noticed the notational similarities between `zero :` and `:  Type`. The type `Type` has the following interpretations :

- As a construction : any type of construction is a recipe for `Type`.
- Geometrically : `Type` is a space of spaces. Each individual point in `Type` is a space.

This may have lead you to the question, `Type :  ?`. In type theory, we simply assert `Type :  Type₁`. But then we are chasing our tail, asking `Type₁ :  Type₂`. Type theorists make sure that every type (i.e. anything the right side of `:`) itself is a term (i.e. anything on the left of `:`), and every term has a type. So what we really need is

```
Type : Type₁, Type₁ : Type₂, Type₂ : Type₃,
```

These are called *universes*. The numberings of universes are called *levels*. It will be crucial that types can be treated as terms. This will allows us to

- talk about *predicates* i.e. “propositions depending on a variable”. E.g. the proposition “`n` is even” depends on a natural number `n`. See the next quest where we elaborate on this example.
- reason about “*structures*” such as “the structure of a group”, to express “for all groups, . . . ”
- do category theory without stepping out of the theory. (For experts, we have Grothendieck universes.)
- reason about when two types are “the same”, for example when are two definitions of the natural numbers “the same”? What is “the same”?

2.3 Quest 1 - Dependent Types

In a "place to do maths" we would like to be able to express and "prove" the statement

The statement

There exists a natural that is even.

The goal of this quest is to define what it means for a natural to be even.

2.3.1 Part 0 - Predicates / Dependent Constructions / Bundles

This requires the notion of a *predicate*. In general a predicate on a type `A : Type` is a term of type `A → Type`. For example,

```
isEven :  → Type
isEven n = ?
```

- Do `C-c C-l` to load the file.
- Navigate to the hole.
- Input `n` in the hole and do `C-c C-c`. You should now see

  ```
  isEven :  → Type
  isEven zero = {!!}
  isEven (suc n) = {!!}
  ```

  It says "to define a function on , it suffices to define the function on the *cases*, `zero` and `suc n`, since these are the only constructors given in the definition of ". This has the following interpretations :

  - propositionally, this is the *principle of mathematical induction*.
  - categorically, this is the universal property of a natural numbers object.

- Navigate to the first hole and check the goal. You should see

  ```
  Goal: Type
  -----------
  ```

  Fill the hole with , since we want `zero` to be even.

- Navigate to the second hole.
- Input `n` and do `C-c C-c` again. You should now see

  ```
  isEven :  → Type
  isEven zero = 
  isEven (suc zero) = {!!}
  isEven (suc (suc n)) = {!!}
  ```

  We have just used induction again.

- Navigate to the first hole and check the goal. `agda` should be asking for a term of type `Type`, so fill the hole with , since we don"t want `suc zero` to be even.
- Navigate to the next hole and check the goal. You should see in the `*Agda information*` window,

```
Goal: Type
--------------
n :
```

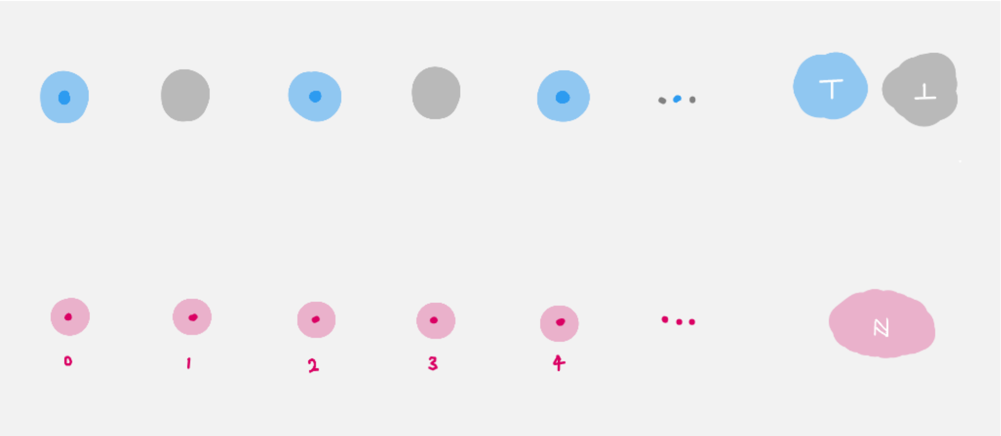
We are in the "inductive step", so we have access to the previous natural number.

- Fill the hole with `isEven n`, since we want `suc (suc n)` to be even *precisely when* `n` is even. The reason we have access to the term `isEven n` is again because we are in the "inductive step".
- There should now be nothing in the `*Agda information*` window. This means everything is working. (Compare your `isEven` with our solutions in `Quest2Solutions.agda`.)

2.3.2 Part 1 - Interpretations of Bundles

The interpretations of `isEven :  → Type` are

- Propositionally : Already mentioned, `isEven` is a predicate on .
- As a construction : `isEven` is a *dependent construction*. Specifically, `isEven n` is either or depending on `n :` .
- Geometrically : seen as a map from the space to the space of spaces `Type`, `isEven` assigns for every point `n` in a space `isEven n`. Pictorially, it looks like

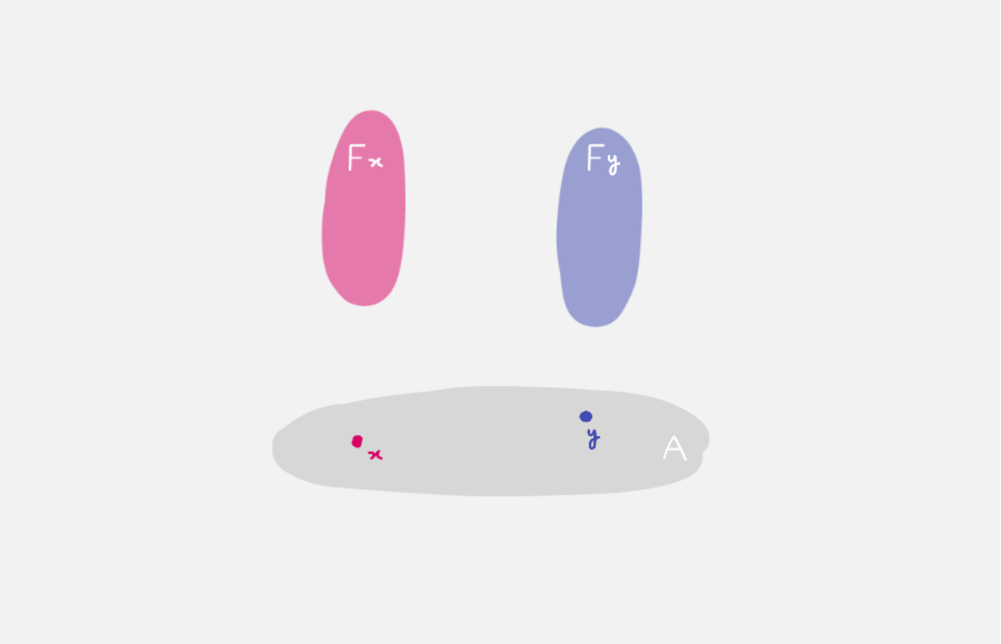


We say `isEven` is a *bundle of spaces over* , or simply *a bundle over* for short. The space `isEven n` lying above each `n` is called the *fiber over* `n`. In this particular example the fibers are either empty or singleton.

Note: In the above picture, we have implicitly drawn as a bunch of "disconnected" points, i.e. a *discrete* space. See a later arc where this is justified.

- Categorically : `isEven` is an object in the over-category `Type↓`.

In general given a type `A : Type`, a *dependent type* `F` *over* `A` is a term `F : A → Type`. This should be drawn as a collection of space parameterised by the space `A`.



You can check if `2` is even by asking `agda` to "reduce" the term `isEven 2` (do `C-c C-n`, "n" for normalize) and type in `isEven 2`. (You can write in numerals since we are now secretly using from the cubical `agda` library.)

2.3.3 Part 2 - Using the Trinitarianism

We introduced new ideas through each perspectives, as each has their own advantage

- Types as propositions is often the most familiar perspective, and hence can offer guidance for the other two perspectives. However the current mathematical paradigm uses "proof irrelevance" (two proofs of the same proposition are always "the same"), which is *not* compatible with HoTT. We will expand on this later.
- Types as constructions conveys the way in which "data" is important, and should be preserved.
- Types as objects/spaces allows us to draw pictures, thus guiding us through the syntax with geometric intuition.

For each new idea introduced, make sure to justify it proof theoretically, type theoretically and categorically/geometrically.

2.4 Quest 2 - Sigma Types

We are still trying to express and "prove" the statement

The statement

There exists a natural that is even.

We will achieve this by the end of this quest.

2.4.1 Part 0 - Existence / Dependent Pair / Total Space of Bundles

Recall from *Quest 1 - Dependent Types* that we defined `isEven`. What's left is to be able write down "existence". In maths we might write

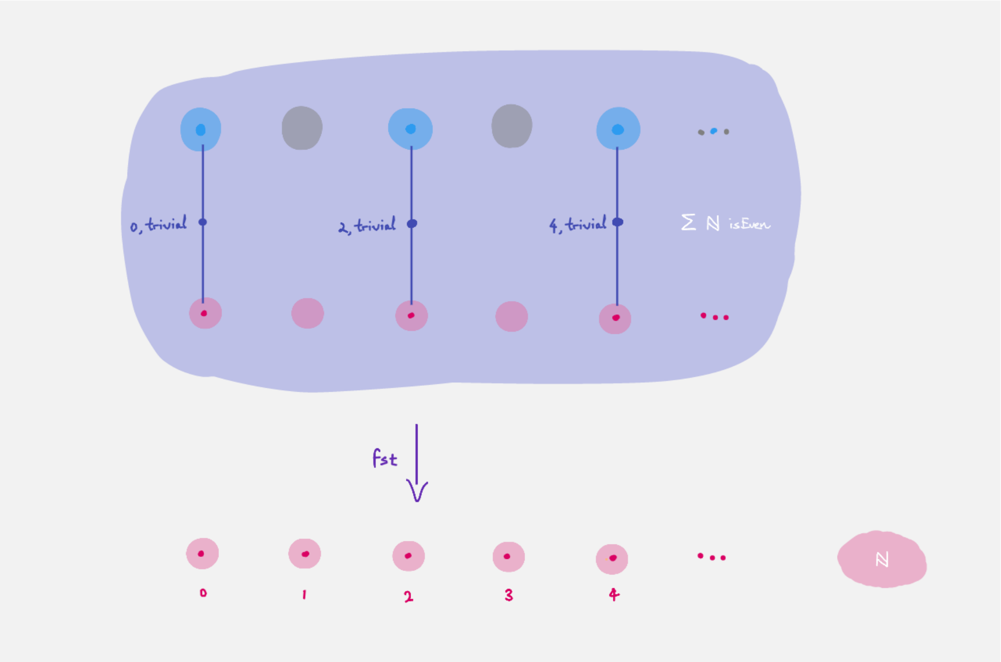
```
x   , isEven x
```

which in `agda` notation is

```
 isEven
```

This is called a *sigma type*, which has three interpretations:

- the proposition "there exists an even natural"
- the construction "keep a recipe `n` of naturals together with a recipe of `isEven n`"
- the total space of the bundle `isEven` over , which is the space obtained by putting together all the fibers. Pictorially, it looks like



which can also be viewed as the subset of even naturals, since the fibers are either empty or singleton. (It is a *subsingleton bundle*).

2.4.2 Part 1 - Making terms in Sigma Types

Making a term of this type has three interpretations:

- (giving a proof that there existence of an even natural amounts to giving) a natural `n :` and a proof `hn : isEven n` that `n` is even.
- pairing a recipe `n :` with a recipe `hn : isEven n`.
- (giving a point in the total space is giving) a point `n :` downstairs together with a point `hn : isEven n` in its fiber.

Now you can prove that there exists an even natural:

- Formulate the statement you need. Make sure you have it of the form

  ```
  Name : Statement
  Name = {!!}
  ```

- Load the file, go to the hole and refine the goal.
- If you formulated the statement right it should split into {!!} , {!!} and you can check the types of terms the holes require.
- Fill the holes. There are many proofs you can do!

In general when A : Type is a type and B : A → Type is a predicate/dependent construction/bundle over A, we can write the sigma type A B whose terms are pairs a , b where a : A and b : B a. In the special case when B is not dependent on a : A, i.e. it looks like a → C for some C : Type then A B is just

- the proposition "A and C" since giving a proof of this is the same as giving a proof of A and a proof of C
- a recipe a : A together with a recipe c : C
- B is now a *trivial bundle* since the fibers B a are constant with respect to a : A. In other words it is just a *product* A B A × C. For this reason, some refer to the sigma type as the *dependent product*, but we will avoid this terminology.

```
_×_ : Type → Type → Type
A × C =  A ( a → C)
```

agda supports the notation _×_ (without spaces) which means from now on you can write A × C (with spaces).

2.4.3 Part 2 - Using Terms in Sigma Types

There are two ways of using a term in a sigma type. We can extract the first part using fst or the second part using snd. Given x : A B there are three interpretations of fst and snd:

- Viewing x as a proof of existence fst x provides the witness of existence and snd provides the proof that the witness fst x has the desired property
- Viewing x as a recipe fst extracts the first component and snd extracts the second component
- Viewing x as a point in the total space of a bundle fst x is the point that x is over in the base space and snd x is the point in the fiber that x represents. In particular you can interpret fst as projection from the total space to the base space, collapsing fibers.

For example to define a map that takes an even natural and divides it by two we can do

```
div2 :   isEven →
div2 x = {!!}
```

- Load the file, go to the hole and case on x. You might want to rename fst$_1$ and snd$_1$.

  ```
  div2 :   isEven →
  div2 (fst₁ , snd₁) = {!!}
  ```

- Case on fst$_1$ and tell agda what to give for 0 , *, i.e. what "zero divided by two" ought to be.

  ```
  div2 :   isEven →
  div2 (zero , snd₁) = {!!}
  div2 (suc fst₁ , snd₁) = {!!}
  ```

- Navigate to the second hole and case on `fst₁` again. Notice that `agda` knows there is no term looking like `1 , *` so it has skipped that case for us.

```
div2 :   isEven →
div2 (zero , snd₁) = 0
div2 (suc (suc fst₁) , snd₁) = {!!}
```

- `(n + 2) / 2` should just be `n/2 + 1` so try writing in `suc` and refining the goal
- How do you write down `n/2`? Hint: we are in the "inductive step".

  Try dividing some terms by `2`:

- Use `C-c C-n` and write `div2 (2 , tt)` for example.
- Try dividing `36` by `2`.

Important observation : the two proofs `2 , tt` and `36 , tt` of the statement "there exists an even natural" are not "the same" in any sense, since if they were `div2 (2 , tt)` would be "the same" `div2 (36/2 , tt)`, and hence `1` would be "the same" as `18`.

"The same"

Are they "the same"? What is "the same"?

2.5 Quest 2 - Side Quests

2.5.1 A Tautology / Currying / Cartesian Closed

In this side quest, you will construct the following functions.

```
uncurry : (A → B → C) → (A × B → C)
uncurry f x = {!!}

curry : (A × B → C) → (A → B → C)
curry f a b = {!!}
```

These have interpretations :

- `uncurry` is a proof that "if `A` implies (`B` implies `C`)", then "(`A` and `B`) implies `C`". A proof of the converse is `curry`.
- currying
- this is a commonly occurring example of an *adjunction*. See here for a more detailed explanation.

Note that we have *postulated* the types `A, B, C` for you.

```
private
  postulate
    A B C : Type
```

In general, you can use this to introduce new constants to your `agda` file. The `private` ensures `A, B, C` can only be used within this `agda` file.

Tip: `agda` is space-and-indentation sensitive, i.e. the `private` applies to anything beneath it that is indented two spaces.

2.6 Quest 3 - Pi Types

We will try to formulate and prove the statement

Problem statement

The sum of two even naturals is even.

2.6.1 Part 0 - Defining Addition

To do so we must define + on the naturals. Addition takes in two naturals and spits out a natural, so it should have type → → .

```
_+_ :  →  →
n + m = ?
```

Try coming up with a sensible definition. It may not look the same as ours.

`n + 0` should be `n` and `n + (m + 1)` should be `(n + m) + 1`.

2.6.2 Part 1 - The Statement

Now we can make the statement that a sum of even naturals is even in `agda`. Make sure it is of the form

```
Name : Statement
Name = ?
```

The statement should be of the form `(x y :  A) → B` where `A` represents the *subset* of even naturals and `B` expresses what it means for the "sum of `x` and `y`" to be even.

Given `x y :    isEven` we want to show that their sum (really the sum of their fist components) is even, so we should give `isEven (x .fst + y .fst)`

Tip: `x .fst` is another notation for `fst x`. This works for all sigma types.

There are three ways to interpret this:

- For all even naturals `x` and `y`, their sum is even.
- `isEven (x .fst + y .fst)` is a construction depending on two recipes `x` and `y`. Given two recipes `x` and `y` of `isEven`, we break them down into their first components, apply the conversion `_+_`, and form a recipe for `isEven` of the result.
- `isEven (_ .fst + _ .fst)` is a bundle over the categorical product `isEven ×  isEven` and `SumOfEven` is a *section* of the bundle. This means for every point `(x , y)` in `isEven ×  isEven`, it gives a point in the fiber `isEven (x .fst + y .fst)`.

More generally given `A : Type` and `B : A → Type` we can form the *pi type* `(x : A) → B x : Type` (in other languages `(x : ), isEven n`), with interpretations :

- it is the proposition "for all `x : A`, we have `B x`", and each term of the pi type *is a collection of proofs ``bx : B x`*, one for each `x : A`.
- recipes of `(x : A) → B x` are made by converting each `x : A` to some recipe of `B x`. Indeed the function type `A → B` is the special case where the type `B x` is not dependent on `x`. Hence pi types are also known as *dependent function types*. Note that terms in the sigma type are pairs `(a , b)` whilst terms in the dependent function type are a collection of pairs `(a , b)` indexed by `a : A`
- Given the bundle `B : A → Type`, we have the total space `A B` which is equipped with a projection `fst : A B → A`. A term of `(x : A) → B x` is a section of this projection.

We are now in a position to prove the statement. Have fun!

2.6.3 Part 2 - Remarks

Important: Once you have proven the statement, check out our two ways of defining addition `_+_` and `_+'_` (in the solutions).

- Use `C-c C-n` to check that they compute the same values on different examples.
- Uncomment the code for `Sum'OfEven` in the solutions. It is just `SumOfEven` but with each `+` changed to `+'`.
- Load the file. Does the proof still work?

Our proof `SumOfEven` relied on the explicit definition of `_+_`, which means if we wanted to use our proof on someone else's definition of addition, it might not work anymore.

Important Question

But `_+_` and `_+'_` compute the same values. Are `_+_` and `_+'_` "the same"? What is "the same"?

2.7 Quest 3 - Side Quests

2.7.1 Decidability of `isEven`

Try to express and prove in `agda` the statement

Problem statement

Every natural number is even or not even.

We make a summary of what is needed:

- a definition of the type `A B` (input `\oplus`), which has interpretations
  - the proposition "`A` or `B`"
  - the construction with two ways of making recipes `left : A → A B` and `right : B → A B`.
  - the disjoint sum of two spaces

  - the coproduct of two objects `A` and `B`. The type needs to take in parameters `A : Type` and `B : Type`

    ```
    data __ (A : Type) (B : Type) : Type where
      ???
    ```

- a definition of negation. One can motivate it by the following
  - Define `A   B : Type` for two types `A : Type` and `B : Type`.
  - Show that for any `A : Type` we have `(A   )   (A → )`
  - Define `¬ :   Type → Type` to be ` A → (A → )`.
- a formulation and proof of the statement above

2.8 Quest 4 - Paths and Equality

If you have come here from *Fundamental Group of the Circle* then have a look at the *overview* to understand the philosophy of trinitarianism.

So far in *trinitarianism* there has been no mention of "equality"; we have never said what it meant for two types or two terms to be "the same". However, in *Fundamental Group of the Circle* we *have* expressed what it means for two *spaces* to look the same, by creating a path from one space to the next (usually by an isomorphism). Indeed we will take this to be our *definition* of (internal) equality.

We will often adopt the geometric perspective, but change perspectives when appropriate.

Universe levels

In the solutions we always use `Type u`, but just write `Type` here. There is no conceptual difference with using an arbitrary universe, but in practice we want to be as general as possible.

It is useful to stick to just using `Type`, and realise why it is not general enough when problems arise.

2.8.1 Part 0 - The Identity Type

The construction

Given `A : Type` and `x y :  A` we have a type `Id x y :  Type`, called the *identity type* of `x` to `y`.

```
data Id {A : Type} : (x y : A) → Type where

  rfl : {x : A} → Id x x
```

The construction takes in (implicit) argument `A : Type`, then for each pair of points `x y :  A` it returns a space `Id x y` with interpretations :

- `Id x y` is the proposition "`x` equals `y` (internally)" and for every `x`, we have a proof `rfl x` that "`x` is equal to itself (internally)". (Hence the name `rfl`, which is short for *reflexivity*.)
- The only recipe for the construction `Id x y` is given when `x` is the same recipe as `y`.
- `Id x y` is the space of paths from `x` to `y`, i.e. points in the space are paths from `x` to `y` in `A`. For every point `x` in `A`, there is the constant path `rfl x` at `x`.

- `Id` is a bundle over `A × A` and the diagonal map `A → A × A`, taking `x   (x , x)`, factors through `Id → A × A` (viewing `Id` as the total space `(A × A) Id`).

$$\begin{tikzcd}
A \arrow[r, "x \mapsto (x, x, \mathrm{refl})"] \arrow[dr, "\text{diagonal}"'] & \sum_{(x,y):A\times A} \mathrm{Id}(x,y) \arrow[d, "(x,y,p) \mapsto (x,y)"] \\
 & A \times A
\end{tikzcd}$$

Write this up in `0Trinitarinism/Quest4.agda`. We recommend you first try having the explicit argument for `rfl` in `rfl : (x : A) → Id x x`, so you can see exactly what is going on, but we will use `rfl` with an implicit argument `rfl : {x : A} → Id x x`.

Internal versus external equality

In the first perspective use the word "internal" since there is also the notion of "external equality" that we want to distinguish. In short `x` and `y` are externally equal if the computer believes they are the same term, i.e. the string of symbols they simplify (normalise) to are exactly the same.

If two terms are externally equal then they are internally equal, and the proof that they are internally equal is `rfl`. However, having a proof `p : Id x y` is not enough for the computer to recognise `x` as the same term as `y`.

Exercise - Symmetry

For `Id` to be a good notion of equality it should at least be an equivalence relation. It is reflexive by having `rfl` in the definition. We show that it is symmetric:

```
idSym : (A : Type) (x y : A) → Id x y → Id y x
idSym = {!!}
```

This has interpretations:

- Equality is symmetric
- We can turn recipes for the construction `Id x y` into recipes for the construction `Id y x`
- Paths can be reversed

Add this to the file `0Trinitarianism/Quest4.agda` and try showing it. We give a detailed explanation in the hints and solution.

Assume we have a space `A`, points `x y : A` and a proof of equality / recipe / path `p : Id x y`. It may help to view `Id x y` as a construction to think about how to proceed.

If you case on `p : Id x y` then you should see the following

```
idSym : (A : Type) (x y : A) → Id x y → Id y x
idSym A x .x rfl = {!!}
```

We interpret this as

- If `x` and `y` are equal by proof `p` and we want to show something about `x` `y` and `p`, then it suffices to consider the case when they are externally equal; that `y` is literally the term `x` and `p` is `rfl`.
- The only recipe we had for the construction `Id x y` is `rfl`, so we should be able to reduce to this case.
- To map out of `Id`, viewed as a total space, it suffices to map out of the diagonal.

$$\begin{tikzcd}
A \arrow[r, "x \mapsto (x, x, \mathrm{rfl})"] & \sum \mathrm{rfl} \arrow[r, hook] \arrow[dr, "f_{\mathrm{rfl}}"'] & \sum \mathrm{Id} \arrow[d, dashed, "f"] \\
& & M
\end{tikzcd}$$

Since we have reduced to the case for when both points are `x`, we can simply supply a point in `Id x x`. There is an obvious one.

```
idSym : (A : Type) (x y : A) → Id x y → Id y x
idSym A x .x rfl = rfl
```

The Geometric Perspective

We have *not* included a justification via the geometric perspective. This is because intuitively it's not quite obvious that to map out of the space of paths it suffices to map the constant path. We justify the mapping out property geometrically in a *side quest*.

We can also make the relevant arguments implicit. We will be using the following version from now on :

```
Sym : {A : Type} {x y : A} → Id x y → Id y x
```

Exercise - Transitivity

In `0Trinitarianism/Quest4.agda`, try to formalize (and then prove) the following interpretations of the same statement :

- `Id` is transitive, which says if `Id x y` and `Id y z` both hold, then so does `Id x z`.
- recipes for `Id x y` and `Id y z` can be made into recipes for `Id x z`.
- paths can be concatenated

```
idTrans : (A : Type) (x y z : A) → Id x y → Id y z → Id x z
idTrans = {!!}
```

You may wish to make some of the arguments implicit. We could also introduce notation that suggests concatenation:

```
_*_ : {A : Type} {x y z : A} → Id x y → Id y z → Id x z
_*_ = {!!}
```

We will use `_*_`.

There are multiple ways of defining this. Assuming `p : Id x y` and `q : Id y z` we could

- case on `p` and identify `x` and `y`
- case on `q` and identify `y` and `z`
- case on both `p` and `q`, identifying all three

```
_*_ : {A : Type} {x y z : A} → Id x y → Id y z → Id x z
rfl * q = q

_*0_ : {A : Type} {x y z : A} → Id x y → Id y z → Id x z
p *0 rfl = p

_*1_ : {A : Type} {x y z : A} → Id x y → Id y z → Id x z
rfl *1 rfl = rfl
```

These three definitions will work slightly differently in practice. We will use the first of the three, but you can use whichever you prefer.

Exercise - Groupoid Laws

The identity type satisfies some further properties, which you can formalize then prove. You may notice that they look almost like the axioms of a group, except a bit bigger - for example there is not just a single identity element (`refl` works at each point in the space).

Note that our solutions may differ to yours depending on your choice of how to define transitivity / concatenation.

- concatenating `rfl` on the left and right does nothing,

  ```
  rfl* : {x y : A} (p : Id x y) → Id (rfl * p) p
  rfl* = {!!}
  ```

(continues on next page)

(continued from previous page)

```
*rfl : {x y : A} (p : Id x y) → Id (p * rfl) p
*rfl = {!!}
```

The first says if you concatenate `rfl` on the left then it is equal to the original path.

```
rfl* : {x y : A} (p : Id x y) → Id (rfl * p) p
rfl* p = rfl

*rfl : {x y : A} (p : Id x y) → Id (p * rfl) p
*rfl rfl = rfl
```

Note that we needed to case on the path in the second proof due to our definition of concatenation.

Tip: If you are tired of writing `{A : Type} {x y : A}` each time you can stick

```
private
  variable
    A : Type
    x y : A
```

at the beginning of your `agda` file, and it will assume `A`, `x` and `y` implicitly whenever they are mentioned. Make sure it is indented correctly. Beware that anything declared like this will be an *implicit argument*.

We also recommend reading about the module system in `agda`.

- concatenating a path `p` with `Sym p` on the left and right gives `rfl`.

```
Sym* : {A : Type} {x y : A} (p : Id x y) → Id (Sym p * p) rfl
Sym* = {!!}

*Sym : {A : Type} {x y : A} (p : Id x y) → Id (p * Sym p) rfl
*Sym = {!!}
```

```
Sym* : {A : Type} {x y : A} (p : Id x y) → Id (Sym p * p) rfl
Sym* rfl = rfl

*Sym : {A : Type} {x y : A} (p : Id x y) → Id (p * Sym p) rfl
*Sym rfl = rfl
```

- Concatenation is associative

```
Assoc : {A : Type} {w x y z : A} (p : Id w x) (q : Id x y) (r : Id y z)
        → Id ((p * q) * r) (p * (q * r))
Assoc = {!!}
```

```
Assoc : {A : Type} {w x y z : A} (p : Id w x) (q : Id x y) (r : Id y z)
        → Id ((p * q) * r) (p * (q * r))
Assoc rfl q r = rfl
```

These axioms say that any type is a *groupoid*, with the above structure. This aligns well with the geometric perspective of types : in classical homotopy theory any space has a groupoid structure and any groupoid can be made into a space.

Recursor - The Mapping Out Property of `Id`

We may wish to extract the way we have made maps out of the identity type :

Mapping out property of Id

Assuming a space `A` and a point `x :  A`. Given a bundle `M : (y :  A) (p :  Id x y) → Type` over the "space of paths out of `x`", in order to make a map `{y :  A} (p :  Id x y) → M y p`, it suffices to give a point in `M x refl`. This is traditionally called the "recursor" of `Id`. (We have still not justified this geometrically.)

For example, in order to prove `*Sym :  {A : Type} {x y :  A} (p :  Id x y) → Id (p * Sym p) rfl`, we would choose our bundle `M` to be ` y p → Id (p * Sym p) rfl`, taking each `y :  A` and `p :  Id x y` to the space of paths from `(p * Sym p)` to `rfl` in `Id x x`. When we proved this in the previous section, `agda` figured out what `M` needed to be and just asked for a proof of the case `M x rfl`.

In `0Trinitarianism/Quest4.agda`, try formalising the mapping out property, and call it `outOfId`.

```
outOfId : (M : (y : A) → Id x y → Type) → M x rfl
  → {y : A} (p : Id x y) → M y p
outOfId = {!!}
```

Note that we have used the symbol `y` in the type of `M`, but it really is just a local variable and will not appear outside that bracket. We made the last `y` an implicit argument, since `p` contains the data of `y`.

```
outOfId : (M : (y : A) → Id x y → Type) → M x rfl
  → {y : A} (p : Id x y) → M y p
outOfId M h rfl = h
```

The proof is of course just casing on the path `p`, as we are trying to extract that idea.

2.8.2 Part 1 - The Path Space

If you came here from the quest on *Fundamental Group of the Circle* then you may be wondering why there has not been any mention of the *path space* `x  y`. The reason is that whilst and `Id` are meant to represent the same idea, the implementation of `Id` is simple - we were able to write it down; whereas the implementation of is "external", and purely existing in `cubical agda`. In this part we will show that the two are "the same" as spaces i.e. isomorphic, and after this we will only use for equality and paths (as is the convention in the cubical library).

We assert the following three axioms for the path space (we will add another (univalence) in later):

- If `x` is a point in some space then `refl` is a proof of `x  x`.
- The mapping out property, called `J` :

  ```
  J : (M : (y : A) → x  y → Type) → M x refl
    → {y : A} (p : x  y) → M y p
  ```

  This looks exactly like `outOfId`.
- The mapping out property applied to `refl` :

  ```
  JRefl : (M : (y : A) → x  y → Type) (h : M x refl)
    → J M h refl  h
  ```

> This says that when we feed `refl` to `J M h` it indeed gives us what we expect - something equal to `h`. Unfortunately, though (given correct `M` and `h`) `outOfId M h rfl` would *externally* be equal to `h`, `J M h refl` is *not externally equal* to `h`, but this is a `cubical agda` issue and not a HoTT issue.

Paths versus `Id`

The goal

Given two points `x y :  A`, the path type `x  y` is isomorphic to `Id x y`. We introduce isomorphism in *Quest 0 of the Fundamental Group arc*.

So we are trying to show

```
PathId : (x  y)  (Id x y)
PathId = {!!}
```

This involves a lot of small steps, which we split up into hints.

"Refining" in the hole will make it ask for the four components in the proof of an isomorphism.

```
PathId : (x  y)  (Id x y)
PathId = iso {!!} {!!} {!!} {!!}
```

To make an isomorphism we need to make maps forwards and backwards, these go in the first two holes.

```
Path→Id : x  y → Id x y
Path→Id = {!!}

Id→Path : Id x y → x  y
Id→Path = {!!}
```

To make the map forwards we will need to use `J` - the mapping out property of the path space. To map backwards we can use `outOfId` or just case on a path.

```
Path→Id : x  y → Id x y
Path→Id {A} {x} = J {!!} {!!}

Id→Path : Id x y → x  y
Id→Path rfl = {!!}
```

For the first, in order to state the motive we need the implicit arguments `A` and `x`.

```
Path→Id : x  y → Id x y
Path→Id {A} {x} = J ( y p → Id x y) rfl

Id→Path : Id x y → x  y
Id→Path rfl = refl
```

Filling in what we have so far and extracting the relevant lemmas we have

```
PathId : (x  y)  (Id x y)
PathId {A} {x} {y} = iso Path→Id Id→Path rightInv leftInv where

```

(continues on next page)

(continued from previous page)

```
   rightInv : section (Path→Id {A} {x} {y}) Id→Path
   rightInv = {!!}

   leftInv : retract (Path→Id {A} {x} {y}) Id→Path
   leftInv = {!!}
```

We have filled in the necessary implicit arguments for you.

Since `section Path→Id Id→Path` will first take in `p :  Id x y` we give such a `p` and case on it. It should of course just turn into `rfl`.

Since `retract Path→Id Id→Path` will first take in `p :  x  y` we directly use `J`.

```
PathId : (x  y)  (Id x y)
PathId {A} {x} {y} = iso Path→Id Id→Path rightInv leftInv where

   rightInv : section (Path→Id {A} {x} {y}) Id→Path
   rightInv rfl = {!!}

   leftInv : retract (Path→Id {A} {x} {y}) Id→Path
   leftInv = J {!!} {!!}
```

Checking the goal for `rightInv` we should see it requires a point in `Path→Id ( _ → x)  rfl`, which is the same as `Path→Id refl  rfl`. What's happened is `agda` knows that `Id→Path rfl` is just `refl` (they are externally equal), so instead of asking for a point of `Path→Id (Id→Path rfl)  rfl` it just asks for a proof of the reduced version. (In our heads we reduce `( _ → x)` to `refl` but `agda` does the opposite.)

We extract the above result as a lemma :

```
Path→IdRefl : Path→Id (refl {x = x})  rfl
Path→IdRefl = {!!}
```

Since `Path→Id` uses `J`, the only thing we can do here is use `JRefl` :

```
Path→IdRefl : Path→Id (refl {x = x})  rfl
Path→IdRefl {x = x} = JRefl ( y p → Id x y) rfl
```

For `leftInv`, giving the correct motive requires knowing what `retract` says. It should look like

```
leftInv : retract (Path→Id {A} {x} {y}) Id→Path
leftInv = J ( y p → Id→Path (Path→Id p)  p) {!!}
```

Checking the goal we should see it requires a point in `Id→Path (Path→Id refl)  refl`. It should be that we just can replace `Path→Id refl` with `rfl` using our lemma `Path→IdRefl :  Path→Id refl  rfl` - but we haven't proven anything about paths yet! Let us do so now : if `f :  A → B` is a function (in our case `Id→Path`) then if two of its inputs are the same `x  y` then so are the outputs, `f x  f y`.

```
cong : (f : A → B) (p : x  y) → f x  f y
cong = {!!}
```

We can prove this directly using `J` or via `Id`. (We call it `cong'` to avoid clashing with the library's version)

```
Cong : (f : A → B) → Id x y → Id (f x) (f y)
Cong f rfl = rfl

```

(continues on next page)

(continued from previous page)

```
cong' : (f : A → B) (p : x  y) → f x  f y
cong' {x = x} f = J ( y p → f x  f y) refl

cong'' : (f : A → B) (p : x  y) → f x  f y
cong'' f p = Id→Path (Cong f (Path→Id p))
```

From now on we will just use `cong` from the library, but you can try to continue with your own version. Now using `cong` we can define `leftInv`. Noting that externally `Id→Path rfl` is the same as `refl`, we just need to show that `Id→Path (Path→Id refl)  Id→Path rfl`.

```
leftInv : retract (Path→Id {A} {x} {y}) Id→Path
leftInv = J ( y p → Id→Path (Path→Id p)  p) (cong ( p → Id→Path p) Path→IdRefl)
```

Concluding that the two types are isomorphic is a good reason to accept them as "the same" in the sense that if two spaces are isomorphic then they share the same properties, because isomorphism should interact nicely with other constructions. We expand upon this point in *Part 3 - Univalence*.

2.8.3 Part 2 - Properties of the Path Space

In *Fundamental Group of the Circle* we assume a couple of results about the path space, which we list here :

- The basics : We can make `sym` (the analogue of `Sym`) and composition of paths (called `__`); we can show that paths also satisfy groupoid laws.
- We have already made `cong` in the previous part (in Hint 6).
- The function `pathToFun` which takes a path between spaces and converts it to a function between the spaces, following points along the path of spaces.
- The function `endPt` which follows a path along a bundle.

Some of these properties are what Homotopy Type theorists believe to be the absolute minimal necessary philosophical foundations for considering paths to be a good notion of equality :

- `refl`, `sym` and `__` give us that it is an equivalence relation
- `cong` tells us that any function respects equality.
- `endPt` and `pathToFun` approximately say that any predicate / family / bundle `B : A → Type` respects equality.

The Basics

The direct proof of these are a good exercise on `J`, or can be accomplished by porting over results from the identity type using `Path→Id` and `Id→Path`. We won't go through each proof, but it is worth noting that since equalities tend to be non-external, a little more work is required. To see solutions for this, please see `0Trinitarianism/Quest4Solutions.agda`.

Chains of Equalities

Something that will help organizing the above proofs and work later on is a notation for composition that suggests a "chain of equalities". Let's say that we want to show that `a + (b + c)  c + (a + b)` for naturals `a b c :  `. Then classically one might write

```
 a + (b + c)
by associativity
 (a + b) + c
by commutativity
 c + (a + b)
```

In `agda` we would have both proofs of associativity and commutativity. Let's call them `ha` and `hc` (in practice they would probably be something like `+assoc a b c` and `+comm (a + b) c`). Then using some cleverly defined notation, we can write in `agda`

```
example : (a b c : ) → a + (b + c)  c + (a + b)
example a b c =
    a + (b + c)
   ha
    (a + b) + c
   hc
    c + (a + b)
```

One you define `__` for composition of paths, you can get access to this notation by including the following code. Try figuring out why it works.

```
___ : (x : A) → x  y → y  z → x  z -- input \< and \>
_  xy  yz = xy  yz

_ : (x : A) → x  x -- input \qed
_  = refl

infixr 30 __
infix  3 _
infixr 2 ___
```

All of this is included in the solutions file.

pathToFun

The function `pathToFun` (originally called `transport` in the `cubical library`) has the following interpretations :

- If two propositions are equal then one implies the other.
- If two constructions can be identified then we can transport recipes of `A` over to recipes of `B`
- If two spaces look the same / if there is a path between spaces in the space of spaces then we can map one to the other (it turns out that we can make `pathToFun` always give us an isomorphism).

Try formalizing and defining `pathToFun` in `0Trinitarianism/Quest4.agda`.

```
pathToFun : A  B → A → B
```

Use `J` to reduce this to finding a map `A → A`, and choose the identity map.

```
id : A → A
id x = x

pathToFun : A  B → A → B
pathToFun {A} = J ( B p → (A → B)) id
```

Show that pathToFun sends refl to the identity map.

```
pathToFunRefl : pathToFun (refl {x = A})  id
pathToFunRefl = {!!}
```

Since the only thing we know about J is how it computes on refl, we apply that :

```
pathToFunRefl : pathToFun (refl {x = A})  id
pathToFunRefl {A} = JRefl ( B p → (A → B)) id
```

We might want to also make pathToFunReflx - which says what pathToFun refl does at each point.

```
pathToFunReflx : (x : A) → pathToFun (refl {x = A}) x  x
pathToFunReflx x = cong ( f → f x) pathToFunRefl
```

endPt

The function endPt (originally called subst in the cubical library) has the following meanings :

- If B is a predicate on A and x y are equal terms of A then B x implies B y. “We can substitute x for y in the proof of B x”.
- If B is a family of constructions dependent on terms of A and x y are identified recipes of A, then recipes of B x can be turned into recipes of B y. “We can substitute the recipe x for y in the recipe for B x”.
- If B is a bundle over the space A and we have a path x y between points in A, then we can follow any “lifted path” starting at some bx : B x to find its end point by : B y.

Predicates / families / bundles respect paths

If we have a predicate / family / bundle B as above and an equality x y in A, then we know that cong will give us an equality of *spaces* B x B y. However, only in the presence of pathToFun is this equality any use - surely if two spaces are equal then we should be able to transport points from one to the other. Hence endPt / pathToFun (often both referred to as transport) justify the statement “predicates / families / bundles” respect paths.

Try to formalize and prove endPt in 0Trinitarianism/Quest4.agda. Then show that it sends refl to what we expect.

One option it is a raw application of J.

```
endPt : (B : A → Type) (p : x  y) → B x → B y
endPt {x = x} B = J ( y p → B x → B y) id

endPtRefl : (B : A → Type) → endPt B (refl {x = x})  id
endPtRefl {x = x} B = JRefl (( y p → B x → B y)) id
```

We could also use cong and pathToFun as described above, however due to size issues that we have not addressed in our insufficiently general definition of cong, we have used the library’s version of cong. (Outside this quest we will be using the library’s version of these definitions.)

```
endPt' : (B : A → Type) (p : x  y) → B x → B y
endPt' B p = pathToFun (cong B p )
```

2.8.4 Part 3 - Univalence

Paths on Other Constructions

The path space tends to interact nicely with other constructions. We give a list of examples here to demonstrate this point :

- For points `(a0 , b0) (a1 , b1) :  A × B` in the product of two spaces we have that `(a0 , b0)  (a1 , b1)` is "the same" space as the product of path spaces `(a0  a1) × (b0  b1)`. Formally we express "the same" using an isomorphism :

  ```
  Path× : {A B : Type} (a0 a1 : A) (b0 b1 : B) → (__ {A × B} ( a0 , b0 ) ( a1 , b1␣
  ↪))  ((a0  a1) × (b0  b1))
  ```

  where we have some kind of product of spaces (however you wish to define it). We give a proof of this in `Quest4Solutions`; it is quite long but a good exercise in using `J`.

- For points `x y :  A  B` in the disjoint sum / coproduct of two spaces we have that the space `x  y` is one of the four cases

  - If they are both "from `A`" then `x  y` is "the same as" the corresponding path space in `A`
  - If they are respectively from `A` and `B` then `x  y` is "the same as" the empty space
  - If they are respectively from `B` and `A` then `x  y` is "the same as" the empty space
  - If they are both "from `B`" then `x  y` is "the same as" the corresponding path space in `B`

  We go through this example in detail *here*.

- If we have two functions `f g :  A → B` then `f  g` is "the same" space as `(a :  A) → f a  g a`. This is called "functional extensionality". The HoTT proof of this is not straight forward, but in the *side quests* we will go through a cubical-specific proof, which is much simpler.

Univalence

Now an important question arises from these considerations :

Important: We have nice ways of describing what paths between points in constructions are, but what should paths in the space of spaces be?

Looking back on this quest (an perhaps one's life experience) we might think "isomorphism" as it is our competing notion of "the same" for spaces. The univalence axiom says something along the lines of this :

Univalence

If two spaces are isomorphic then they are equal.

```
isoToPath : {A B : Type} → A  B → A  B
```

Actually univalence tends to refer to something slightly different, whilst this is a corollary of it. Refer to The HoTT Book for more details.

Hence any isomorphism we have shown can be upgraded to a path between spaces in `cubical agda`. For example `(x y)  (Id x y)` can now be shown.

2.9 Quest 4 - Side Quests

2.9.1 Functional Extensionality

We show that two dependent functions `f g` being equal is the same as them being equal when applied to each value of the domain. We call one of these directions *functional extensionality* :

```
funExt : {B : A → Type} {f g : (a : A) → B a} →
   ((a : A) → f a  g a) → f  g
funExt = {!!}
```

Write this up in `agda` and have a go at it.

We will cheat and use the native cubical definition of paths (rather than using our axiomatic approach with `J` and `JRefl` etc), since the HoTT proof of this is much more work. A path in `cubical agda` between two points `x` and `y` in a space `A` can be defined by just taking an arbitrary point `i` on the "interval" `I`, to a point in the space `A`, such that the end points agree. Assuming we have the bundle `B`, functions `f g`, a proof `h` of `(a :  A) → f a  ga`, we can refine, and `agda` will assume such an `i` for us.

```
funExt : {B : A → Type} {f g : (a : A) → B a} →
  ((a : A) → f a  g a) → f  g
funExt h =  i a → {!!}
```

Checking the goal you should see something like the following (we have extracted the important parts):

```
  Goal: B a
-----------------------------------
a : A
i : I
h : (a₁ : A) → f a₁  g a₁
g : (a₁ : A) → B a₁   (not in scope)
f : (a₁ : A) → B a₁   (not in scope)
B : A → Type   (not in scope)
A : Type   (not in scope)
---- Constraints ------------------------
?0 (i = i1) = g a : B a
?0 (i = i0) = f a : B a
```

We break this down :

- `agda` has assumed an arbitrary `i :  I` and `a :  A`, and is now asking for a point in `B a`.
- Let's call whatever we put in the goal `?0`; it has type `B a`. The constraints say that at the start and end points of `I` (called `i0` and `i1` respectively) `0? i` should be `f a` and `g a` respectively.
- To understand why `agda` also gave us an `a :  A` we can go back a step, removing `a`. You should see that the goal at that point was a dependent function that at the start and end points are `f` and `g` respectively.

- Try to complete the quest. You will need that given a path `p` and `i : I` along the interval, writing `p i` gives the corresponding point along the path `p`.

The hypothesis `h` applied to the point `a` gives us a path from `f a` to `g a` in `B a`.

```
funExt : {B : A → Type} {f g : (a : A) → B a} →
  ((a : A) → f a  g a) → f  g
funExt h =  i a → h a i
```

Now we can promote this to an isomorphism, hence an equality between `f  g` and `(a : A) → f a  g a`. Try to formalize and prove this.

funExtPath : (B : A → Type) (f g : (a : A) → B a) → (f g) ((a : A) → f a g a) funExtPath {A} B f g = isoToPath (iso fun (funExt B f g) rightInv leftInv) where

> fun : f g → (a : A) → f a g a fun h = a i → h i a
>
> rightInv : section fun (funExt B f g) rightInv h = refl
>
> leftInv : retract fun (funExt B f g) leftInv h = refl

2.9.2 Justifying `J`

Work in progress.

2.10 Quest 5 - Dependent Paths

2.10.1 Part 0 - A motivating example

In *Quest 0 - Working with the Circle* we define the circle, which we work with here. We recommend also going through the definitions for `doubleCover`, `flipPath` and `Flip` in the same quest. They will be referred to here.

```
data S¹ : Type where
  base : S¹
  loop : base  base
```

In said quest we experience mapping out of `S¹` by casing on a point `x : S¹`; this was `doubleCover`, it was *not a dependent function*, in the sense that `doubleCover : (x : S¹) → Type` where `Type` does *not* depend on `x`. We give an example of having to construct a map out of `S¹` that *is* dependent on `x`:

```
example : (x : S¹) → doubleCover x → doubleCover x
example = {!!}
```

We intend for this map to flip each fiber `doubleCover x` just like `Flip : Bool → Bool` flips `Bool`.

- We could case on `x` like we did in the definition of `doubleCover`, resulting in

  ```
  example : (x : S¹) → doubleCover x → doubleCover x
  example base = {!!}
  example (loop i) = {!!}
  ```

- Check and fill the first goal for the `base` case.

  It asks for a map `Bool → Bool` since the fiber `doubleCover base` is by definition `Bool`. We give `Flip` since we want it to flip each fiber.

- In the second case we need to give a map `doubleCover (loop i) → doubleCover (loop i)`, which by definition reduces to `flipPath i → flipPath i`. It is not immediately obvious what we can do here. Case on things in `flipPath i` is not an option for instance.

We should take a step back and notice what we have. Firstly `i → doubleCover (loop i) → doubleCover (loop i)` defines a path in the space of spaces (it is a function space at each `i`). It starts and ends at `Bool → Bool`.

On the other hand, the goal requires is a "path" starting and ending at `Flip : Bool → Bool`, and being a point in `p i : doubleCover (loop i) → doubleCover (loop i)` at each point. It moves along *inside* the path of spaces `i → doubleCover (loop i) → doubleCover (loop i)`. However, this "path" `p` moving along inside the path of spaces is *not* a path in a single space, so we need to formalise this new notion.

Idea

What we need is a generalisation of paths : we need paths that can move between spaces, which we call "dependent paths".

2.10.2 Part 1 - Dependent Paths

In general

Recall that if we have two spaces `A0 A1 : Type` (e.g. both `Bool → Bool`) and a path `A : A0  A1` between them (e.g. `i → doubleCover (loop i) → doubleCover (loop i)`), then any point in `A0` can be transported along the path `A` to a point in `A1` using `pathToFun` a.k.a. `transport`. Since `A0` and `A1` are internally *equal*, one might wonder if we can even consider what it means for points `x : A0` and `y : A1` to be *equal*, perhaps keeping the path `A` in mind somehow (e.g. `x` and `y` are both `Flip`). We are asking whether the notion of a *dependent path* - in this case a path dependent on `A` - can be made precise. *Externally* `x` and `y` belong to different spaces so it doesn't make sense to ask for the path type `x  y`, but HoTT and `cubical agda` offer solutions to this.

In HoTT, a workaround is "say `x` and `y` are equal along `A` when we have a path from `pathToFun A x` to `y` in the space `A1`" (note that we made a choice of a path in `A1` here; we could have done the same with `A0`, with an inverted `pathToFun`). This is sensible, since `pathToFun A x` is meant to be the point in `A1` corresponding to `x` under the identification of the spaces `A0` and `A1` given by `A`. Try to define this in `1FundamentalGroup/Quest5.agda`.

```
PathD : {A0 A1 : Type} (A : A0  A1) (x : A0) (y : A1) → Type
PathD A x y = pathToFun A x  y
```

If you like, we can introduce suggestive notation for dependent paths, but may be harder to read than `PathD` :

```
syntax PathD A x y = x  y along A
```

So now we can write `x  y along A` to mean paths from `x` to `y` dependent on the path `A`.

There is a slightly different `cubical agda` way of going about this. Intuitively a path in `cubical agda` is a starting point, an ending point, and something in between that agrees on the boundary. Thus a path dependent on `A : A0  A1` from `x` to `y` can be introduced by giving at each arbitrary `i : I` on the "interval" a point `t : A i` such that `t` is *externally equal to* `x` at the start and `y` at the end.

```
PathP : (A : I → Type) → A i0 → A i1 → Type
```

`A` takes each `i : I` to a space `A i : Type`, so we can think of `A` as a path. Then `PathP` takes `A`, a point `x : A i0` in the starting space and a point `y : A i1` in the ending space, and gives the space of dependent paths along `A`.

We will try to mostly use the HoTT version of paths, since HoTT is the main discussion here. So we will assume that the two notions are the same using an isomorphism `PathPIsoPathD` from the library.

```
PathPIsoPath : (A : I → Type) (x : A i0) (y : A i1) →
   (PathP A x y)  (transport ( i → A i) x  y)
```

Let us continue with the example to understand how this works.

Using Dependent Paths

Going back to our example, we need to give a dependent path from Flip to Flip - dependent on the path i → flipPath i → flipPath i in the space of spaces. Let us extract this as a lemma :

```
example : (x : S¹) → doubleCover x → doubleCover x
example base = Flip
example (loop i) = p i where

  p : PathP ( i → flipPath i → flipPath i) Flip Flip
  p = {!!}
```

At point loop i on the loop, we give the point p i in flipPath i → flipPath i. Note that PathP needs to know which path we are depending on, and that is the first piece of data it takes in.

Now, instead of giving a PathP, as agda natively prefers, we will give a PathD, using PathPIsoPathD. PathPIsoPathD will give us an isomorphism, but we only want the map backwards - taking a PathD and giving us a PathP. To do so we write __.inv in the hole and refine. It knows that the goal is a PathP, so it should reduce to

```
p : PathP ( i → flipPath i → flipPath i) Flip Flip
p = __.inv {!!} {!!}
```

Check the goals, in the first it should now be asking for an isomorphism, which we give by refining with PathPIsoPathD, the second hole depends on the first, so it will make more sense when we can come back to it later.

```
p : PathP ( i → flipPath i → flipPath i) Flip Flip
p = __.inv (PathPIsoPathD {!!} {!!} {!!}) {!!}
```

Now try to give PathPIsoPathD the necessary inputs.

It just needs to know what path we want to be dependent over, the starting point, and the ending point.

```
p : PathP ( i → flipPath i → flipPath i) Flip Flip
p = __.inv (PathPIsoPathD ( i → flipPath i → flipPath i) Flip Flip) {!!}
```

Checking the final hole we see that we need a path from the function pathToFun (i_1 → flipPath i_1 → flipPath i_1) Flip to the function Flip. This is now just a normal path in Bool → Bool. We refrain from spoiling the rest of the proof.

To prove that two functions are the same we can use funExt to just check they are the same at each point. Naturally, we extract this as a lemma so that we can case on the point in Bool.

Reminding ourselves of what flipPath looks like, and what pathToFun does, we should be able to guess what the values on each side turn out to be.

```
example : (x : S¹) → doubleCover x → doubleCover x
example base = Flip
example (loop i) = p i where

  lem : (x : Bool) → pathToFun ( i → flipPath i → flipPath i) Flip x  Flip x
```

(continues on next page)

(continued from previous page)

```
  lem false = refl
  lem true = refl

  p : PathP ( i → flipPath i → flipPath i) Flip Flip
  p = __.inv (PathPIsoPathD ( i → flipPath i → flipPath i) Flip Flip) (funExt lem)
```

Mapping out of the circle

We might want to generalize the above process once and for all so that we can map out the circle with greater ease. We suggest that to map out of the circle into a bundle over the circle `B : S¹ → Type, it suffices to give a point b : B base to map base to, and to give a PathD dependent on B and loop which starts and ends at b.

Try to formalise and prove this in the quest.

You need not, but we found it is convenient to define one for each PathP and PathD. The first is of course trivial.

```
outOfS¹P : (B : S¹ → Type) → (b : B base) → PathP ( i → B (loop i)) b b → (x : S¹)␣
↪→ B x
outOfS¹P B b p base = b
outOfS¹P B b p (loop i) = p i

outOfS¹D : (B : S¹ → Type) → (b : B base) → b  b along ( i → B (loop i))
  → (x : S¹) → B x
outOfS¹D B b p x = {!!}
```

The next we can define using the first, using PathPIsoPathD.

```
outOfS¹D : (B : S¹ → Type) → (b : B base) → b  b along ( i → B (loop i))
  → (x : S¹) → B x
outOfS¹D B b p x = outOfS¹P B b (__.inv (PathPIsoPathD ( i → B (loop i)) b b) p) x
```

Cases / Induction / Recursors / Universal properties

In general, given a higher inductive type we will always have the above process, which can be interpreted in the following ways :

- It is casing on where the term came from or where the proof came from. For example to map out of the proposition "A or B" we can case on if the proof came from A or from B. To map out of "A and B" we can case on the proof and it must give us a pair, proving both.
- It is induction on the inductively defined type. This was exemplified in our *discussion on the naturals*.
- It is the mapping out property of the type, commonly called *the recursor*, and it just considers what recipes went into making the type. For example the only recipes that went into making the circle are base and loop.
- Often this can be seen as a universal property. For example the universal property of disjoint sums (a.k.a coproducts a.k.a "or") can be seen as saying "to map out of A B it suffices to give a map out of A and a map out of B"

We should verify that the this mapping out property does actually give us what we expect. For example, we gave it a point b : B base to map base. We therefore should expect that it does map base to b. Tracing through the definitions we have made, we should be able to see this is true *externally*.

More explicitly

```
outOfS¹DBase : (B : S¹ → Type) (b : B base)
  (p : b  b along ( i → B (loop i)))→ outOfS¹D B b p base  b
outOfS¹DBase B b p = refl
```

2.10.3 Part 2 - How pathToFun Interacts with Other Types

When we are coming up with dependent paths between points in equal spaces connected by some path A, we end up with needing some idea of what pathToFun looks like when it goes along the path. For example, if A were i → B i → C i, where B and C are respectfully paths between spaces, then we might guess that we can describe pathToFun B f more explicitly by checking what it does on points.

In this part we will consider different type constructions and how paths between them convert to functions between them via pathToFun. A detailed motivating example can be found *here*.

Function spaces

Suppose we have spaces A0 A1 B0 B1 : Type and paths A : A0 A1 and B : B0 B1. Then let pAB denote the path i → A i → B i : (A0 → B0) (A1 → B1). We want to figure out what pathToFun does when it follows a function f : A0 → B0 along the path pAB.

We know by functional extensionality that the function pathToFun pAB f : A1 → B1 should be determined by what it does to terms in A1, so we can assume a1 : A1. The idea is we "apply f by sending a1 back to A0". Since pathToFun (sym A) a1 is meant to give the point in A0 that "looks like a1", we try applying f to this point, then send it across again via the path B to the point f (pathToFun (sym A) a1) looks like in B1. We expect the outcome to be the same.

$$\begin{tikzcd}
A_1 \arrow[r, dashed, "\texttt{pathToFun}\ p_{AB}\ f"] & B_0 \\
A_0 \arrow[u, "\texttt{pathToFun}\ A"] \arrow[r, "f"'] & B_1 \arrow[u, "\texttt{pathToFun}\ B"']
\end{tikzcd}$$

Try to formalize and prove this in 0Trinitarianism/Quest5.agda.

```
pathToFun→ : {A0 A1 B0 B1 : Type} {A : A0  A1} {B : B0  B1} (f : A0 → B0) →
  pathToFun ( i → A i → B i) f   a1 → pathToFun B (f (pathToFun (sym A) a1))
pathToFun→ = {!!}
```

There are several ways to state the same idea. We didn't have to reverse the path `A` for example.

We can induct on both `A` and `B`.

```
 J ( A1 A → pathToFun ( i → A i → B i) f   a1 → pathToFun B (f (pathToFun (sym A) a1)))
(
  J ( B1 B → pathToFun ( i → A0 → B i) f   a → pathToFun B (f (pathToFun (sym refl) a)))
  (
      pathToFun refl f
     {!!}
      ( a → pathToFun refl (f (pathToFun (sym refl) a)))
  )
  B
)
A
```

There are many small equalities that are needed, for example, we need how `sym` and `refl` interact and what `pathToFun` does to `refl`. At some point it would be useful to just check that the functions are equal on terms.

```
pathToFun→ : {A0 A1 B0 B1 : Type} {A : A0  A1} {B : B0  B1} (f : A0 → B0) →
  pathToFun ( i → A i → B i) f   a1 → pathToFun B (f (pathToFun (sym A) a1))
pathToFun→ {A0} {A1} {B0} {B1} {A} {B} f =
  J ( A1 A → pathToFun ( i → A i → B i) f   a1 → pathToFun B (f (pathToFun (sym A) a1)))
  (
    J ( B1 B → pathToFun ( i → A0 → B i) f   a → pathToFun B (f (pathToFun (sym refl) a)))
    (
        pathToFun refl f
       pathToFunReflx f
        f
       funExt ( a →
           f a
          cong f (sym (pathToFunReflx a))
           f (pathToFun refl a)
          cong ( p → f (pathToFun p a)) (sym symRefl)
           f (pathToFun (sym refl) a)
          sym (pathToFunReflx (f (pathToFun (sym refl) a)))
           pathToFun refl (f (pathToFun (sym refl) a))
      )

        ( a → pathToFun refl (f (pathToFun (sym refl) a)))
    )
    B
  )
  A
```

More to come in the future

This quest is a work in progress.

CHAPTER

THREE

FUNDAMENTAL GROUP OF THE CIRCLE

3.1 Overview

One key attraction of HoTT (homotopy type theory) is for doing homotopy theory *synthetically* - like how one does Euclidean geometry from axioms without needing the existence of the real numbers. In this arc we will formalize what it means for the circle `S¹` to have fundamental group in this setting.

Applying the philosophy of trinitarianism, is strongly recommended in this arc. However, it is designed so that anyone eager to see familiar geometric results can also start here with no prerequisites. Hence, this arc mostly adopts a geometric (hence categorical) perspective on types.

3.2 Quest 0 - Working with the Circle

In this series of quests we will prove that the fundamental group of `S¹` is . In fact, our strategy will also show that the higher homotopy groups of `S¹` are all trivial. You don't need to know any prerequisites - in particular we will define the fundamental group and higher homotopy groups if you don't know what they are already.

Important: In your cloned copy of the HoTT Game locate the file `1FundamentalGroup/Quest0.agda`, and open this file in `emacs`. Before starting it is important to have a look at our *guide to emacs and list of emacs commands*.

3.2.1 Part 0 - The Circle

Theory - Definition of the Circle

We begin by formalising the problem statement.

A construction of "the circle" is :

- a point called `base`
- an edge from that point to itself called `loop`

Here is our definition of the circle in `agda`.

```
data S¹ : Type where
  base : S¹
  loop : base  base
```

This reads :

- S^1 is a point in Type, the *space of spaces*. In other words, S^1 is a space.
- base is a point in the space S^1
- loop is a *path* in S^1 from base to itself. This is phrased as saying loop is a point in base base the *space of paths from* base *to* base.

Path

We think of a path in a space A as consisting of its starting point, its end point, and some generic point in the middle agreeing on the boundary.

You can see this as defining the circle via a CW-complex.

Type theory notation

In general a : A is read as a is a point in the space A. Note that in the above definition S^1 is seen both as a point and a space depending on the context. In cubical agda, everything is a point in a "unique" space.

Type theory notation

In general when a b : A (a and b are points in a space A), we have a *path space* a b, whose points are *paths* from a to b in the space A.

Exercise - defining the constant path Refl

There are other paths in S^1, for example the *constant path at* base. In 1FundamentalGroup/Quest0.agda navigate to

```
Refl : base  base
Refl = {!!}
```

We will guide you through defining it. We are about to construct a path Refl : base base, a path from base to base.

Tip: The {!!} are called *holes*. These are blanks in the agda file that you can fill to complete the quest. You can write ? to make a new hole.

We will fill the hole Refl = {!!}.

- Make sure you are in *insert mode* by pressing i. To escape *insert mode* press ESC.

  Note: We have compiled a list of useful emacs and agda commands in *Emacs Commands*.

- Enter C-c C-l (this means Ctrl-c Ctrl-l).

Whenever you do this, agda will check the document is written correctly.

We say agda *compiles* or *loads* the file. This will open the *Agda Information* window looking like

```
?0 : base  base
?1 : (something)
?2 : (something)
...
```

This is the list of unfilled holes that are in your file currently. You should see that the holes in the file have changed in appearance, for example :

```
Refl : base  base
Refl = { }0
```

These are what holes look like when the file is compiled. The numbering is just for reference and may change upon reloading.

- *Navigate between holes* using `C-c C-f` (forward) or `C-c C-b` (backward).
- Navigate to the first hole, making sure your cursor is inside the hole. *Check the goal* using `C-c C-,` (this means `Ctrl-c Ctrl-comma`). Whenever you do `C-c C-,`, `agda` will tell you what kind of "point" it expects in the hole. The `*Agda Information*` window should be focused on this hole only :

```
Goal: base  base
```

This says `agda` is expecting a path from `base` to `base` in the hole. Making sure your cursor is still inside the hole, enter `C-c C-r`. The `r` stands for *refine*. Whenever you do this whilst having your cursor in a hole, `agda` will try to help you.

- You should now see ` i → {!!}`. This is the `agda` way of writing `i  {!!}`. Load the file again (using `C-c C-l`) and the `*Agda Information*` window should now look like :

```
?0 : S¹
...
?6 : (something)

---- Errors ---------------
Failed to solve the following constraints:
  ?0 (i = i1) = base : S¹ (blocked on _3)
  ?0 (i = i0) = base : S¹ (blocked on _3)
```

Do not worry about the errors, we will soon explain it.

- Navigate (`C-c C-f` and `C-c C-b`) to that new hole in ` i → {!!}` and enter `C-c C-,` to *check the goal*. The `*Agda information*` window should look like :

```
Goal: S¹
-------------------------
i : I
---- Constraints --------------
?0 (i = i1) = base : S¹ (blocked on _3, belongs to problem 4)
?0 (i = i0) = base : S¹ (blocked on _3, belongs to problem 4)
_4 :=  i → ?0 (i = i) (blocked on problem 4)
```

This says :

- `agda` is expecting a point in `S¹` for this hole.
- you have a point `i` in `I` available to you. You can think of `I` as the "unit interval" and `i` as a generic point in the interval.

  - The point in S¹ that you give has to satisfy the constraints that it is `base` when "`i = 1`" and "`i = 0`". In `agda`, `i0` and `i1` are the "start" and "end" point of `I`. Afterall, we are defining a path from `base` to itself.
  - Don't worry about the last line.

- Since `Refl` is meant to be the constant path at `base`, write `base` in the hole.
- Press `C-c C-SPC` to fill the hole with `base`. In general when you have some text (and your cursor) in a hole, doing `C-c C-SPC` will tell `agda` to replace the hole with that text. `agda` will give you an error if it can't make sense of your text.

Tip: Everytime you are filling a hole, it is recommended that you first write what you want to fill in the hole *then* do `C-c C-SPC`. You can do it in the reverse order, however the recommended order has other benefits down the line.

- Load the file again (`C-c C-l`). The `*Agda Information*` window should now look like this :

```
?0 : Bool
?1 : Bool  Bool
?2 : Bool  Bool
?3 : Type
?4 : doubleCover base
?5 :
```

The `?0 :  S¹` has disappeared. This means `agda` has accepted what you filled this hole with.

- If you want to play around with this you reset this question by replacing what you wrote with `?` and doing `C-c C-l`.

3.2.2 Part 1 - `Refl  loop` is empty

To get a better feel of `S¹`, we show that the space of paths (homotopies) between `Refl` and `loop`, written `Refl  loop`, is empty.

Paths between paths

In general if we have `p q :  a  b` in a space `A` then a path `Path :  p  q` in the path space `a  b` consists of

- the starting path `p`
- the end path `q`
- and some generic path in between `Path i :  a  b` that agrees on the boundary

In algebraic topology this is called a *path homotopy*.

First, we define the empty space and what it means for a space to be empty. Here is what this looks like in `agda` :

```
data  : Type where
```

This says "the empty space is a space with no points in it".

Here are three candidate definitions for a space `A` to be empty :

- there is a point `f :  A →` in the space of functions from `A` to the empty space
- there is a path `p :  A ` in the space of spaces `Type` from `A` to the empty space

- there is an isomorphism `i : A   of spaces`

These turn out to be "the same" (see *Different notions of "empty"*), however for our present purposes we will use the first definition. Our goal is therefore to produce a point in the function space

```
( Refl   loop ) →
```

The authors of this series have thought long and hard about how one would come up with the following argument. Unfortunately, sometimes mathematics is in need of a new trick and this was one of them.

The trick

We make a path `p :  true   false` from the assumed path (homotopy) `h :  Refl   loop` by constructing a non-trivial `Bool`-bundle over the circle, hence obtaining a map `( Refl   loop ) →` .

To elaborate : `Bool` here refers to the discrete space with two points `true, false`. We will create a map `doubleCover :  S`^{1} → `Type` that sends `base` to `Bool` and the path `loop` to a non-trivial path `flipPath :  Bool   Bool` in the space of spaces.

Viewing the picture vertically, for each point `x :  S`^{1}, we call `doubleCover x` the *fiber of* `doubleCover` *over* `x`. All the fibers look like `Bool`, hence our choice of the name `Bool`- *bundle*.

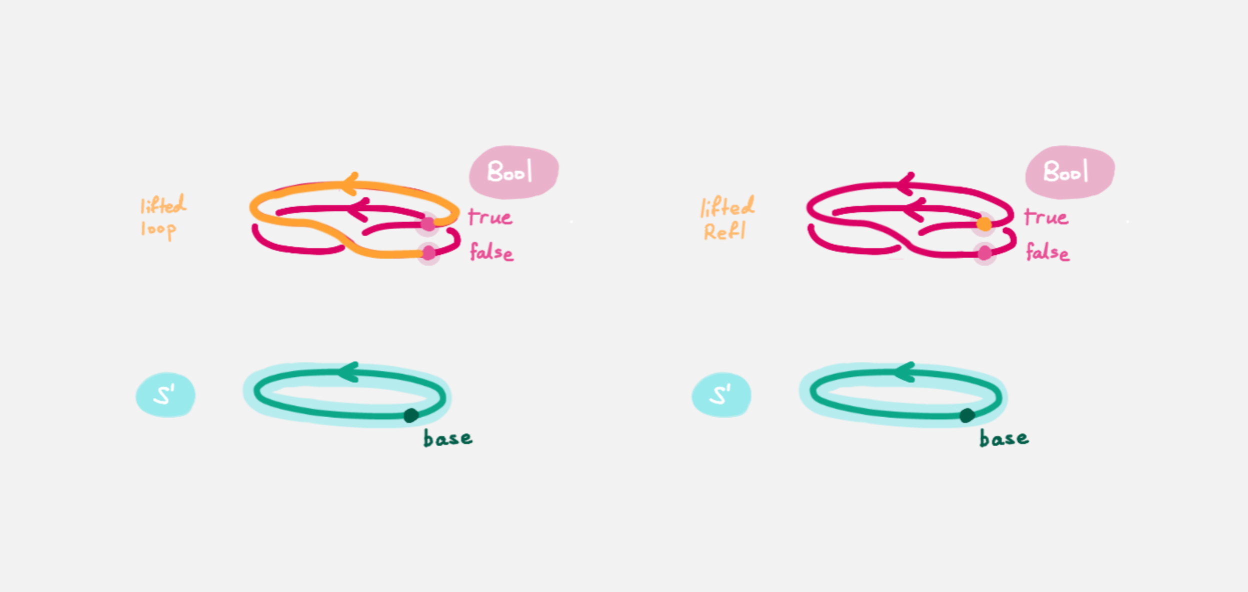
Homotopy type theory

A path `p :  X   Y` between two *spaces* `X Y : Type` (viewed as points in the space of spaces) can be visualised as follows :

- Two spaces `X` and `Y` as end points.
- For the generic point `i :  I` in the interval `p i :  Type` is a space that varies continuously with to respect to `i` such that `p 0` is `X` and `p 1` is `Y`.

The continuity guarantees that each point along the path looks "the same". In particular the end points look "the same".

We will get a path from `true` to `false` in the fiber of `doubleCover` over `base` by "lifting the homotopy" `h :  Refl loop` and considering the end points of the "lifted paths". `Refl` will "lift" to a "constant path" and `loop` will "lift" to



Let's assume for the moment that we have `flipPath` already and define `doubleCover`.

- Make sure you are in *insert mode*.
- Navigate to the definition of `doubleCover` and make sure you have *loaded the file* with `C-c C-l`.

  ```
  doubleCover : S¹ → Type
  doubleCover x = {!!}
  ```

- *Navigate to the hole* and *check the goal*. It should look like

  ```
  Goal: Type
  ————————————————————
  x : S¹
  ```

  This says it is expecting a point in `Type`, the space of spaces, i.e. it expects a space. We will first *case on x* by writing `x` in the hole and doing `C-c C-c` (`c` for cases). You should now see two new holes :

  ```
  doubleCover : S¹ → Type
  doubleCover base = {!!}
  doubleCover (loop i) = {!!}
  ```

  This means : `S¹` is made from a point `base` and an edge `loop`, so a map out of `S¹` to a space is the same as choosing a point to map `base` to, and an edge to map `loop` to respectively. Since `loop` is a path from `base` to itself, its image must also be a path from the image of `base` to itself.

- *Navigate* to the first new hole and *check the goal*. We want to map `base` to the space `Bool` so write `Bool` in the hole, then do `C-c C-SPC` to *fill* it.
- *Navigate* to the second new hole and *check the goal*. Here `loop i` is a generic point in the path `loop`, where `i : I` is a generic point of the "unit interval". We are assuming we have `flipPath` defined already and want to map `loop` to `flipPath`, so `loop i` should map to a generic point in the path `flipPath`.

  Note: We can use `flipPath` without completing the definition of `flipPath`.

  Try filling the hole.

- Once you think you are done, *reload* the `agda` file with `C-c C-l` and if it doesn't complain this means there are no problems with your definition. Compare your definition to that in `1FundamentalGroup/Quest0Solutions.agda` to check that your definition is the one we want. To navigate to solutions file escape *insert mode* using `ESC` and do `SPC f f` to find the file, see *Emacs and Unicode Commands*. Here is a definition that `agda` will accept, but is *not* what we need:

  ```
  doubleCover : S¹ → Type
  doubleCover base = Bool
  doubleCover (loop i) = Bool
  ```

Defining `flipPath` is quite involved and we will do so in the following part.

3.2.3 Part 2 - Defining `flipPath` via Univalence

In this part, we will define the path `flipPath :  Bool  Bool`. Recall the picture of `doubleCover`.

This means we need `flipPath` to correspond to the unique non-identity permutation of `Bool` that flips `true` and `false`.

The function

We proceed in steps :

1. Define the function `Flip :  Bool → Bool`.
2. Promote this to an isomorphism `flipIso :  Bool  Bool`.
3. We use *univalence* to turn `flipIso` into a path `flipPath :  Bool  Bool`. The univalence axiom asserts that paths in `Type` - the space of spaces - correspond to homotopy-equivalences of spaces. As a corollary, we can make paths in `Type` from isomorphisms in `Type`.

Isomorphism

One with a topological mindset might worry if isomorphism means homeomorphism, homotopy equivalence, bijection or something else; one might even wonder what *continuity* is here. The answer is that this is *synthetic homotopy theory*, where

- there is *no need for real numbers*
- every map is continuous in the sense that they respect paths
- an isomorphism `A  B` is given by the data of
  - `fun :  A → B`
  - `inv :  B → A`
  - `rightInv` that says (extensionally) `fun  inv` is homotopic to the identity, i.e. given any `b :  B` we have a path `fun inv b  b`.
  - `leftInv` that says (extensionally) `inv  fun` is homotopic to the identity.

  You might notice that the above looks like the classical definition of homotopy equivalence. (They turn out to be "the same".)

Univalence

We have described paths between points as giving a starting point, an ending point, and a generic point between that agrees on the boundary. Drawing a path between *spaces* in the *space of spaces*, we can see that such a path is the data of two spaces that "continuously look the same":

We already have a notion of "the same" for spaces, which is isomorphism. Hence we assume the following "univalence" axiom : Any isomorphism can be turned into a path between spaces.

- In `1FundamentalGroup/Quest0.agda`, navigate to :

```
Flip : Bool → Bool
Flip x = {!!}
```

- Make sure you are in *insert mode*.

- *Check the goal*. It should be asking for a point in `Bool`, since we have already given it an `x :  Bool` at the front.

Tip: Whenever you encounter a new hole, you should first *check the goal*.

- Write `x` inside the hole, and *case* on `x` using `C-c C-c` with your cursor still inside. You should now see :

  ```
  Flip : Bool → Bool
  Flip false = {!!}
  Flip true = {!!}
  ```

  This means : the space `Bool` is made of two points `false, true` and nothing else, so to map out of `Bool` it suffices to find images for `false` and `true` respectively.

- Since we want `Flip` to flip `true` and `false`, fill the first hole with `true` and the second with `false`.

- To check things have worked, try `C-c C-d` (`d` stands for *deduce its space*). Then `agda` will ask you to input an expression. Enter `Flip`. In the `*Agda Information*` window, you should see

  ```
  Bool → Bool
  ```

  This means `agda` recognises `Flip` as a well-formulated term and is a point in the space of maps from `Bool` to `Bool`.

- We can also ask `agda` to compute outputs of `Flip`. Try `C-c C-n` (`n` stands for *normalise*), `agda` should again be asking for an expression. Enter `Flip true`. In the `*Agda Information*` window, you should see `false`, as desired.

The isomorphism

- *Navigate* to

  ```
  flipIso : Bool  Bool
  flipIso = {!!}
  ```

- *Refine* with `C-c C-r`. You should now see

  ```
  flipIso : Bool  Bool
  flipIso = iso {!!} {!!} {!!} {!!}
  ```

- Given two spaces `A` and `B`, `iso` (with respect to `A` and `B`) belongs to the following space :

  ```
  iso : (fun : A → B) (inv : B → A)
        (rightInv : section fun inv) (leftInv : retract fun inv) →
        A  B
  ```

  which says that `iso` will produce an isomorphism from `A` to `B` given a map `fun` forwards and an inverse `inv` backwards, and points of the space `section fun inv` and `retract fun inv`. Try to find out what `section` and `retract` are by doing `C-c C-n` and entering their respective names. They should respectively say that `inv` is a right and left inverse of `fun`.

- *Check the first two holes*, `agda` should expect functions `Bool → Bool` to go in both of them. This is because it is expecting a function and its inverse, respectively, so fill them with `Flip` and its inverse `Flip`.

- Check the goal of the next two holes. They should be

```
section Flip Flip
```

and

```
retract Flip Flip
```

- Add the following to your code (make sure you copy it exactly) :

```
flipIso : Bool  Bool
flipIso = iso Flip Flip {!!} {!!} where

  rightInv : section Flip Flip
  rightInv x = {!!}

  leftInv : retract Flip Flip
  leftInv x = {!!}
```

Then *load* the file with `C-c C-l`. If `agda` gives an error it could be due to

1. missing spaces; `agda` is space sensitive
2. wrong indentation before `rightInv` and `leftInv`; `agda` is indentation sensitive
3. missing the `where` in the second line.
4. lower and upper case differences

where to use `where`

The `where` allows you to make definitions local to the current definition, in the sense that you will not be able to access `rightInv` and `leftInv` outside this proof. We will eventually fill the missing holes from before with `rightInv` and `leftInv`. If you like you can also place the definitions of `rightInv` and `leftInv` before `flipIso`.

- *Check the goal* of the hole `rightInv x = {!!}`. In the `*Agda Information*` window, you should see

```
Goal: Flip (Flip x)  x
---------------------------------
x : Bool
```

This says `rightInv` should give for each `x : Bool` a path `p : Flip (Flip x)  x`. We gave an `x : Bool` in front, so the goal is simply to give a path `p : Flip (Flip x)  x`. Try to give such a path.

You need to *case* on what `x` can be. Then for the case of `false`, `Flip (Flip false)` should just be `false` by design, so you need to give a path from `false` to `false`.

The benefit of having `x` before the = is that we can case on what `x` can be. This is called *pattern matching*.

- Do the same for `leftInv x = {!!}`.
- Fill in the missing goals from the original problem using `rightInv`, `leftInv`.
- If you got the definition right then `agda` should not have any errors when you load using `C-c C-l`.

The path

- *Navigate* to

```
flipPath : Bool  Bool
flipPath = {!!}
```

- In the hole, write in `isoToPath` and refine with `C-c C-r`. You should now have

```
flipPath : Bool  Bool
flipPath = isoToPath {!!}
```

  If you check the new hole, you should see that `agda` is expecting an isomorphism `Bool  Bool`.

  `isoToPath` is the function from the cubical library that converts isomorphisms between spaces into paths between the corresponding points in the space of spaces `Type`.

- Fill in the hole with `flipIso` and use `C-c C-d` to check `agda` is happy with `flipPath`.
- Try to *normalise* `pathToFun flipPath false`. You should get `true` in the `*Agda Information*` window.

  What `pathToFun` did is it took the path `flipPath` in the space of spaces `Type` and followed the point `false` as `Bool` is transformed along `flipPath`. The end result is of course `true`, since `flipPath` is the path obtained from `flip`! Try to follow what `pathToFun` does in the *animation*.

  pathToFun

  `pathToFun` is called `transport` in the cubical library.

3.2.4 Part 3 - Lifting paths using doubleCover

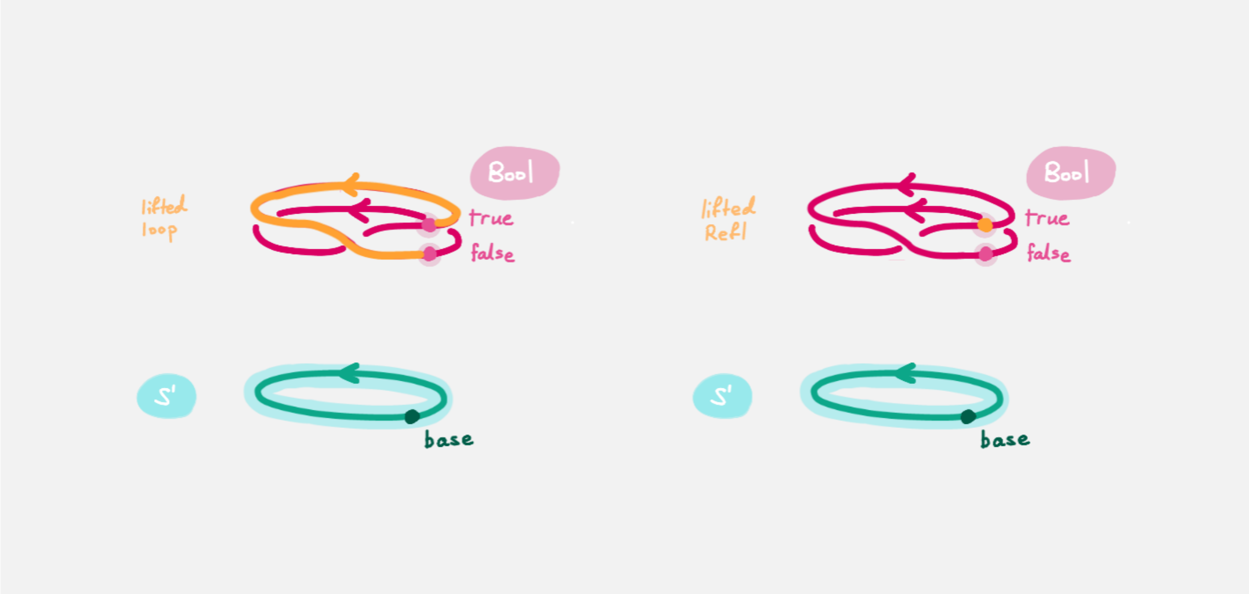
By the end of this page we will have shown that `refl  loop` is an empty space. In `1FundamentalGroup/Quest0.agda` locate

```
Reflloop : Refl  loop →
Reflloop h = ?
```

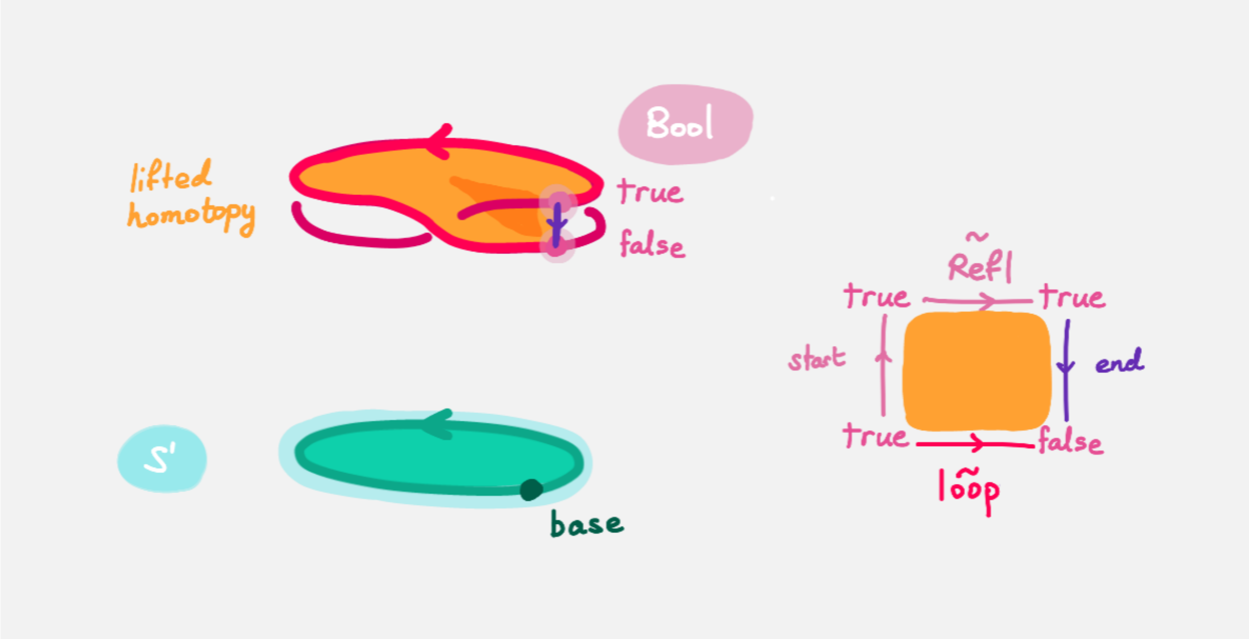
The cubical library has the result `truefalse :  true  false →` which says that the space of paths in `Bool` from `true` to `false` is empty. We will assume it here and leave the proof as a side quest, see *Proving truefalse*.

- Load the file with `C-c C-l` and navigate to the hole. Write `truefalse` (input `\==n` for ; see *Emacs and Unicode Commands*) in the hole and refine using `C-c C-r`, `agda` knows `truefalse` maps to so it automatically will make a new hole.
- Check the goal in the new hole using `C-c C-,` it should be asking for a path from `true` to `false`.

To give this path we need to visualise "lifting" `Refl`, `loop` and the homotopy `h :  Refl  loop` along the Bool-bundle `doubleCover`. When we "lift" `Refl` - starting at the point `true :  doubleCover base` - it will still be a constant path at `true`, drawn as a dot `true`. When we "lift" `loop` - starting at the point `true :  doubleCover base` - it will look like



The homotopy `h :  Refl  loop` is "lifted" (starting at "lifted `Refl`") to some kind of surface



According to the pictures the end point of the "lifted" `Refl` is `true` and the end point of the "lifted" `loop` is `false`. We are interested in the end points of each "lifted paths" in the "lifted homotopy", since this forms a path in the endpoint fiber `doubleCover base` from `true` to `false`.

We can evaluate the end points of both "lifted paths" by using something in the cubical library (called `subst`) which we call `endPt`.

```
endPt : (B : A → Type) (p : x  y) (bx : B x) → B y
```

Note: It says given a bundle `B` over space `A`, a path `p` from `x :  A` to `y :  A`, and a point `bx` above `x`, we can get the end point of "lifted `p` starting at `bx`". So let's make the function that takes a path from `base` to `base` and spits out the end point of the "lifted paths" starting at `true`.

```
endPtOfTrue : base  base → doubleCover base
endPtOfTrue p = {!!}
```

- *Check the goal*. It should be asking for

  ```
  Goal: Bool
  ————————————————————————————————————————————————————————————
  p : base  base
  ---- Constraints ---------------
  ?0 (p = loop) = false : Bool
    (blocked on _29, belongs to problem 90)
  ?0 (p = Refl) = true : Bool (blocked on _29, belongs to problem 90)
  _40 :=  p i → endPtOfTrue (p i) (blocked on problem 90)
  ```

- We want to use `endPt`, which can output something in the space `B y` (as described above). In this case we want `B y` to be `Bool`. `agda` is smart and can figure out how to use `endPt` :

  1. Type `endPt` into the hole and do `C-c C-r`.

  Tip: In general if the goal of the hole

  ```
  Goal: Y
  ————————————————————————————————————————————————————————————
  f : X → Y
  ...
  ```

  is to find a point in a space `Y` and you have a function `f :  X → Y` then you can write `f` in the hole and do `C-c C-r`.

  You should see

  ```
  endPtOfTrue : base  base → doubleCover base
  endPtOfTrue p = endPt {!!} {!!} {!!}
  ```

  2. *Check these new holes*.
  3. Try to fill in these holes.

- Once you think you are done, you can verify our expectation that `endPtOfTrue Refl` is `true` and `endPtOfTrue loop` is `false` using `C-c C-n`.

Lastly we need to make the function `endPtOfTrue` take the path `h :  Refl  loop` to a path from `true` to `false`. In general if `f :  A → B` is a function and `p` is a path between points `x y :  A` then we get a map `cong f p` from `f x` to `f y`. (Note that `p` here is actually a homotopy `h`.)

```
cong : (f : A → B) → (p : x  y) → f x  f y
```

We will define `cong` in a *side quest* Using `cong` and `endPtOfTrue` you should be able to complete `Quest0`. If you have done everything correctly you can reload `agda` and see that you have no remaining goals.

3.3 Quest 0 - Side Quests

3.3.1 Different notions of “empty”

The following are “the same”,

- there is a point `f :  A →  ` in the space of functions from `A` to the empty space
- there is a path `p :  A  ` in the space of spaces `Type` from `A` to the empty space
- there is an isomorphism `i :  A  ` (input `\cong` for) of spaces

Here we will take “the same” to mean there are maps from any one to another (they are “propositionally the same”). We will first define the three.

In `1FundamentalGroup/Quest0.agda`, uncomment this side quest and locate these three definitions :

```
_toEmpty : (A : Type) → Type
A toEmpty = {!!}

pathEmpty : (A : Type) → Type₁
pathEmpty A = {!!}

isoEmpty : (A : Type) → Type
isoEmpty A = {!!}
```

Note: You can use underscores when you name a function. `agda` will put the inputs in the underscores in order.

Tip: `agda` supports unicode symbols such as . See *here* for how to insert and other symbols.

Try to fill them in according to the above. You may have noticed we used `Type₁` in the second definition. To find out what `Type₁` is, see *Part 3 - Universes* in *Trinitarianism*.

Check that your definitions are the same as ours by comparing with the solutions in `1FundamentalGroup/Quest0Solutions.agda`. We will make maps from `toEmpty A` to `isoEmpty A` to `pathEmpty A` and back to `toEmpty A`.

First we show that the empty space maps into any other space. This is *very useful* when working with the empty space.

```
outOf : (A : Type) →  → A
outOf = {!!}
```

Try to fill in the definition without looking at the hint.

Recall the definition of the empty space being a CW-complex with nothing in it. We can always *case* on variable points from CW-complexes. What cases are there?

We fill in `toEmpty→isoEmpty`

```
toEmpty→isoEmpty : (A : Type) → toEmpty A → isoEmpty A
toEmpty→isoEmpty A = {!!}
```

Tip: You can use `where` to extract lemmas / make local definitions like we did in defining `flipIso`; see *here*.

- Check the goal to see what we have and what we need to give.
- Assume `f :  toEmpty A` by putting an `f` before the =.
- Refine the goal to see what `agda` suggests.
- We need to give an isomorphism, i.e. a map from `A` to , and a map from to `A`, and proofs that these satisfy `section` and `retract` respectively.
- If we have a point in then we can get a point in any space.

Try filling in

```
isoEmpty→pathEmpty : (A : Type) → isoEmpty A → pathEmpty A
isoEmpty→pathEmpty A = {!!}
```

We converted an isomorphism to a path in *quest 0*.

Lastly try filling in

```
pathEmpty→toEmpty : (A : Type) → pathEmpty A → toEmpty A
pathEmpty→toEmpty A = {!!}
```

- Check the goal
- We can assume a path `p :  pathEmpty A`
- Check the goal again
- Since `toEmpty A` as defined as `A →` we can assume a point `x :  A`
- We can follow the point `x` along the path `p` using `pathToFun`, as we did for `flipPath` in *Quest 0 - Working with the Circle*.

3.3.2 Proving `truefalse`

Locate `1FundamentalGroup/Quest0SideQuests/TrueNotFalse.agda` we will show

```
truefalse : true  false →
truefalse = {!!}
```

We do this by making a *subsingleton bundle* over `Bool` whose fiber over `true` is the singleton space and fiber over `false` is the empty space . The definition of is

```
data  : Type where
  tt :
```

- Assume a path `h :  true  false`
- Define a map from `Bool` to `Type` (as a lemma or using where), that takes `true` to and `false` to . This is a *subsingleton bundle* over `Bool`, since each *fiber* is and , having only a single or no points.
- We can follow how the point `tt :` changes along the path `h` using `pathToFun`, as we did for `flipPath` in *Quest 0 - Working with the Circle*. This should give you a point in the empty space .

Due to the previous side quest *Different notions of "empty"* this tells us that the space `true  false` is empty.

3.3.3 Defining cong

Under construction

3.4 Quest 1 - Loop Space of the Circle

3.4.1 Part 0 - Definition of the Loop Space

In this quest, we continue to formalise the problem statement.

The problem statement

The fundamental group of S^1 is .

Intuitively, the fundamental group of S^1 at base consists of loops based as base up to homotopy of paths. In homotopy type theory, we have a native description of loops based at base : it is the space base   base.

In general the *loop space* of a space A at a point a is defined as follows :

```
loopSpace : (A : Type) (a : A) → Type
loopSpace A a = a  a
```

For now, we will treat the loop space of S^1 as the fundamental group. Later we will understand why this is illegal in general (the fundamental group is *set truncated*) but legitimate in this special case (the loop space of S^1 turns out to be a *set* anyway).

Exercise - loop_times

Clearly for each integer n :   we have a path that is “loop around n times”. Locate loop_times in 1FundamentalGroup/Quest1.agda (note how agda treats underscores)

```
loop_times :  → loopSpace S¹ base
loop n times = {!!}
```

Note: You can use underscores when you name a function. agda will put the inputs in the underscores in order.

Try casing on n, you should see

```
loop_times :  → loopSpace S¹ base
loop pos n times = {!!}
loop negsuc n times = {!!}
```

It says to map out of it suffices to map the non-negative integers (pos) and the negative integers (negsuc). The definition of in agda is

```
data  : Type where
  pos    :  → 
  negsuc :  → 
```

It says is is two copies of where the first copy represents `0, 1, 2, ...`, and the second represents `-1, -2, ...` (`negsuc n` is meant to mean `- (n + 1)`). This definition of uses the naturals, so try casing on `n` again, you should see

```
loop_times :  →  S¹ base
loop pos zero times = {!!}
loop pos (suc n) times = {!!}
loop negsuc n times = {!!}
```

It says to map out of it suffices to map `zero` and map each successive integer `suc n` inductively. We can do the same with `negsuc n`, obtaining four cases.

```
loop_times :  →  S¹ base
loop pos zero times = {!!}
loop pos (suc n) times = {!!}
loop negsuc zero times = {!!}
loop negsuc (suc n) times = {!!}
```

These four cases represent :

- If you “loop 0 times” then you stay at `base`.
- If you “loop n + 1 times”, you “loop n times” then “loop once more”.
- If you “loop -1 times”, you “loop once in reverse”
- If you “loop -(n + 2) times”, you loop “loop -(n + 1) times” then “loop once more in reverse”

Individually

- Try filling the first hole with what we get when we loop `0` (`pos zero`) times.
- For looping `pos (suc n)` times we loop `n` times and loop once more. For this we need composition of paths.

  ```
  __ : x  y → y  z → x  z
  ```

  Try typing `__` or `?  ?` in the hole (input `\.`) and refining. Checking the new holes you should see that now you need to give two loops.

  ```
  loop pos (suc n) times = {!!}  {!!}
  ```

  Try giving it “`loop n times`” concatenated with `loop`.

- To “loop in reverse” we use

  ```
  sym : x  y → y  x
  ```

  Use this to define “loop -1 times”.

- For the last case “concatenate loop -(n + 1) times with loop in reverse”.

3.4.2 Part 1 - Making a Path From to Itself

In the previous part we have defined the map loop_times : → S^1 base. Creating the inverse map is difficult without access to the entire circle. Similarly to how we used doubleCover to distinguish refl (Refl is now refl which is more general) from loop, the idea is to replace Bool with , allowing us to distinguish between all loops on S^1. In this quest we will construct one of the two comparison maps across the whole circle, called windingNumber.

The plan is :

1. Define a function suc : → that increases every integer by one
2. Prove that suc is an isomorphism by constructing an inverse map pred : → .
3. Turn the isomorphism suc into a path sucPath :   using isoToPath
4. Define helix : S^1 → Type by mapping base to and a generic point loop i to sucPath i.
5. Use helix and endPt to define the map windingNumberBase : base base → . Intuitively it counts how many times a path loops around S^1. a generic point loop i to sucPath i.
6. Generalize this across the circle.

In this part, we focus on 1, 2 and 3.

Defining suc

- Set up the definition of suc so that it is of the form :

  ```
  Name : TypeOfSpace
  Name inputs = ?
  ```

  Just writing in the name and the type of the space is enough for now. Load the file and check that it is looks like:

  ```
  suc :  → 
  suc = ?
  ```

- We will define suc the same way we defined loop_times : by induction. Do cases on the input of suc. You should have something like :

  ```
  suc :  → 
  suc pos n = ?
  suc negsuc n = ?
  ```

- For the non-negative integers pos n we want to map to its successor. Recall that the n here is a point of the naturals whose definition is :

  ```
  data  : Type where
    zero : 
    suc :  → 
  ```

  Use suc to map pos n to its successor.

- The negative integers require a bit more care. Recall that annoyingly negsuc n means "- (n + 1)". We want to map - (n + 1) to - n. Try doing this. Then realise "you run out of negative integers at -(0 + 1)" so you must do cases on n and treat the -(0 + 1) case separately.

  Do C-c C-c on n. Then map negsuc zero to pos zero. For negsuc (suc n), map it to negsuc n.

- This completes the definition of suc. Use C-c C-n to check it computes correctly. E.g. check that suc (negsuc zero) computes to pos zero and suc (pos zero) computes to pos (suc zero).

suc is an Isomorphism

- The goal is to define pred : → which "takes n to its predecessor n - 1". This will act as the (homotopical) inverse of suc. Now that you have experience from defining suc, try defining pred.
- Imitating what we did with flipIso and give a point sucIso : by using pred as the inverse and proving section suc pred and retract suc pred.

suc is a Path

- Imitating what we did with flipPath, upgrade sucIso to sucPath.

3.4.3 Part 2 - Winding Number

The -bundle helix

We want to make a -bundle over S^1 by 'copying across the loop via sucPath'. In Quest1.agda locate

```
helix : S¹ → Type
helix = {!!}
```

Try to imitate the definition of doubleCover to define the bundle helix. You should compare your definition to ours in Quest1Solutions.agda. Note that we have called this helix, since the picture of this -bundle looks like

Counting Loops

Now we can do what was originally difficult - constructing an inverse map (over all points). Now we want to be able to count how many times a path base base loops around S^1, which we can do now using helix and finding end points of 'lifted' paths. For example the path loop should loop around once, counted by looking at the end point of 'lifted' loop, starting at 0. Hence try to define

```
windingNumberBase : base  base → helix base
windingNumberBase = {!!}
```

- endPt evaluates the end point of 'lifted paths'.

Try computing a few values using C-c C-n, you can try it on refl, loop, 'loop three times', 'loop negative one times' and so on.

Generalising

The function windingNumberBase can actually be improved without any extra work to a function on all of S^1.

```
windingNumber : (x : S¹) → base  x → helix x
windingNumber = {!!}
```

Try filling this in. We will show that this and a general version of loop_times are inverses of each other over S^1, in particular obtaining an isomorphism between base base and .

3.5 Quest 2 - is a Set

An overview of this quest :

- We want to show that the higher loop spaces of S^1 are trivial
- We note that it suffices to show that the loop space of is trivial, assuming the end result `loopSpace S¹ base` .
- Show that the loop space of any set is trivial, hence it suffices to show that is a set
- Show that looks like the sum of two disjoint copies of , and is a set; it then suffices to show the general result that the disjoint sum of two sets is a set.
- To show that the disjoint sum of sets is a set we find ourselves trying to classify the path space of disjoint sums.

The bulk of the work will be to classify the path space of disjoint sums, and showing that it actually corresponds to the path space. This is the content of the last three parts.

3.5.1 Part 0 - `loopSpace loopSpace`

We are interested in knowing what the higher homotopy groups of S^1 might be. Whilst the data of the fundamental group $_1$ S^1 is captured in `loopSpace S¹ base`, the data of $_2$ S^1 would be captured in `loopSpace (loopSpace S¹ base) refl`; loops in `loopSpace S¹ base` based at `refl`. Points in the second loop space are paths `h : refl refl`, i.e. `h` would be a homotopy from the constant path to itself.

The second loop space contains an obvious point `refl : refl refl` (this is of course not the same "refl" as the one before), and we could define the next loop space to be loops in `loopSpace (loopSpace S¹ base) refl` based at `refl` (the one from `loopSpace S¹ base` that is).

The important conclusion we will arrive at in this quest is that the loop space of - which will correspond to the second loop space of S^1 (this is the conclusion of this entire arc) - is trivial, in the sense that it just consists of a point (up to paths) :

```
loopSpace (loopSpace S¹ base) refl  loopSpace  0
```

Intuitively this is because the only loop (up to a path) in from `0` to itself is `refl`, so `loopSpace  0` is contractible - it looks just like the singleton space . This is more general : *any two paths in are homotopic*, which we formalise in the definition `isSet`.

`isSet`

`isSet`

The statement "any two paths in the space `A` are homotopic" is captured in the definition of `isSet` :

```
isSet : Type → Type
isSet A = (x y : A) → isProp (x  y)
```

In the above `isProp` captures the statement "any two points are (continuously) connected by a path" :

```
isProp : Type → Type
isProp A = (x y : A) → x  y
```

If a type satisfies `isSet` we say it *is a set*, and if it satisfies `isProp` we say it *is a proposition*.

Intuitively a “set” is meant to be a bunch of disjoint points. However in homotopy type theory we consider points up to paths, and paths up to homotopy, hence a “set” is a bunch of disjoint blobs, where each blob is contractible to a point. In other words a “set” is a type where any circle (that lands in a blob) can be filled (hence the blob is contractible).

We will justify the use of the word “proposition” once we have introduced the propositional perspective on types, see *trinitarianism* and *Part 5 - Using the Propositional Perspective*.

There is a subtlety in the definition `isProp`. Having `isProp A` is *stronger* than saying that the space `A` is path connected. Since `A` is equipped with a *continuous* map taking pairs `x y : A` to a path between them.

We will show in a later quest that `isProp S¹` is *empty* despite `S¹` being path connected.

We can justify “the loop space of a set is trivial” by showing that “if any two paths in a space `A` are homotopic then the loop space of `A` at any point in `A` looks like ”. So we show that

```
isSet→LoopSpace : {A : Type} (x : A) → isSet A → (x  x)
isSet→LoopSpace = {!!}
```

Locate this in `1FundamentalGroup/Quest2.agda` and try filling it in.

Imitating what we did with `flipPath` and `flipIso` reduce this to showing that for each `x : A` and `h : isSet A` we have

- `fun : x  x →`
- `inv :  → x  x`
- `rightInv : section fun inv`
- `leftInv : retract inv fun`

There is only one possible map from `x  x` to since is terminal (see *trinitarianism*).

To map out of one can do cases and see that you only need to map `tt`.

- `fun` can just be `( p → tt)`
- `inv` can be

  ```
  inv :  → x  x
  inv tt = refl
  ```

For `rightInv` by casing on the point in there should be nothing much to show.

For `leftInv` we need to use our assumption that “any two paths are homotopic”.

```
rightInv : section ( p → tt) inv
rightInv tt = refl

leftInv : retract ( p → tt) inv
leftInv p = h x x refl p
```

The goal

We have therefore reduced our goal to showing that is a set, i.e. only has trivial paths in it, which will tell us that the second loop (and in fact any higher loop space) of `S¹` is trivial.

3.5.2 Part 1 - as a disjoint sum

As a first step, we note that actually looks like two disjoint copies of , i.e. we have

```
 :
```

where we have the definition of the *disjoint sum of two spaces* as follows

```
data __ (A B : Type) : Type where

  inl : A → A  B
  inr : B → A  B
```

It says there are two ways of making points in the space, taking them from `A` and taking them from `B`. Try proving in `1FundamentalGroup/Quest2.agda`.

As in defining `flipPath` in *quest 0* we first make an isomorphism and then convert it to a path/proof of equality. To make the isomorphism note that the definition of is already as "two copies of ", as described in *quest 1*.

If you have made the function and inverse appropriately, you should only need constant paths in the proofs that they satisfy `section` and `retract` respectively.

```
 :
 = isoToPath (iso fun inv rightInv leftInv) where

fun :  →
fun (pos n) = inl n
fun (negsuc n) = inr n

inv :    →
inv (inl n) = pos n
inv (inr n) = negsuc n

rightInv : section fun inv
rightInv (inl n) = refl
rightInv (inr n) = refl

leftInv : retract fun inv
leftInv (pos n) = refl
leftInv (negsuc n) = refl
```

We want to show that is a set, by using the path . Intuitively if is a set then two disjoint copies of it should also be a set, (think about filling circles on the disjoint sum). Thus we can break down our goal into two :

Goal 1 : is a set

```
isSet : isSet
isSet = {!!}
```

Goal 2

If `A` and `B` are both sets then `A  B` is also a set.

Goal 1 will be handled in a side quest. We focus on Goal 2 from now on.

3.5.3 Part 2 - Disjoint Sum of Sets is a Set

Try formulating (but not proving) the result `isSet`, which should say "if spaces `A` and `B` are both sets then so is their disjoint sum `A   B`". This should be done in `1FundamentalGroup/Quest2.agda` where indicated.

```
isSet : {A B : Type} → isSet A → isSet B → isSet (A  B)
isSet = {!!}
```

Without proving this, we can use this to show `isSet (  )` using `isSet :  isSet` , which will be shown in a side quest. Then using either `pathToFun` or `endPt` you can show `isSet` from `isSet (  )`, using the path from to   we made earlier. Try to set up everything described in this paragraph where indicated in `1FundamentalGroup/Quest2.agda`.

```
isSet : isSet
```

To use `pathToFun` you must figure out what path you are following and what point you are following the path along.

To use `endPt` you must figure out what bundle you are making, what the path in the base space is, and what point you are starting at in the first fiber.

The point you need to follow in either case is the point in the space `isSet (  )` :

```
isSet : isSet
isSet = pathToFun {!!} (isSet isSet isSet)

isSet' : isSet
isSet' = endPt {!!} {!!} (isSet isSet isSet)
```

```
isSet : isSet
isSet = pathToFun (cong isSet (sym )) (isSet isSet isSet)

isSet' : isSet
isSet' = endPt ( A → isSet A) (sym ) (isSet isSet isSet)
```

If you tried refining using `endPt` you may have been given 2 holes instead of 3. This is because `agda` had too many possible options when trying to match up the output of `endPt` and the goal. To add an extra hole simply add a `?` afterwards and reload.

Once this is complete we can go back and work on `isSet`.

3.5.4 Part 3 - Path Space of Disjoint Sums

Motivation

- Locate your formulation of `isSet`.
- We assume `hA : isSet A`, `hB : isSet B`, and points `x y :  A  B`. Currently our code looks like

  ```
  isSet : {A B : Type} → isSet A → isSet B → isSet (A  B)
  isSet hA hB x y = {!!}
  ```

- Check the goal. It should be asking for a point in the space isProp (x   y).

  We need to consider how to get information on the path space of A   B when our hypotheses are about the path spaces of A and B respectively. We could try to case on x and y.

- If x and y are "both from A", i.e. of the form inl ax and inl ay for ax ay :   A, then we need to find a point in isProp (inl ax   inl ay). This *should* be due to hA, which gives us hA ax ay :   isProp (ax   ay). So somehow we need to identify the path spaces inl ax   inl ay and ax   ay (try to formalize this, though we are not expecting a solution here).

- If x and y are of the forms inl ax and inr by respectively for ax :   A and by :   B then intuitively the space inl ax   inr bx *should* be empty, since the sum is disjoint (again we are not expecting a solution here).

- The other two cases are similar.

The conclusion is that we need some kind of classification of the path space of disjoint sums.

Classifying the Path Space of Disjoint Sums

Path space of disjoint sums

A path in the the disjoint sum should just be a path in one of the two parts.

This says points from A cannot be confused with points from B or points in A that they were not already path connected to.

For now we leave isSet alone and define a function NoConfusion that takes two points in A   B and returns a space, which is meant to represent the path space in each case, as described in our motivation above. Try to formulate (but not fill in) this where indicated in Quest2.agda. It should look like:

```
NoConfusion : {A B : Type} → A   B → A   B → Type
NoConfusion = {!!}
```

Assume points x and y in the disjoint sum and try to case on them. There should be four cases.

- When both points are from A, i.e. they are inl ax and inl ay, then we should give the space ax   ay, which we expect to be isomorphic to inl ax   inl ay.

- (Two cases) When each is from a different space we expect the path space between them to be empty, so we should give .

- If both are from B then we should imitate what we did in the first case

```
NoConfusion : A   B → A   B → Type
NoConfusion (inl x) (inl y) = x   y -- Path A x y
NoConfusion (inl x) (inr y) =
NoConfusion (inr x) (inl y) =
NoConfusion (inr x) (inr y) = x   y -- Path B x y
```

Using the Classification

Now we have two of goals :

- `PathNoConfusion` : We need to show that for each `x y :  A  B` the path space looks like our classification, i.e. that `(x  y)  (NoConfusion x y)`
- `isSetNoConfusion` : For `isSet`, given `hA : isSet A`, `hB : isSet B` and `x y :  A  B` we needed to show `isProp (x  y)`. Hence we want to show that under the same assumptions `isProp (NoConfusion x y)`.

Formalise (but don't prove) both of these where indicated in `1FundamentalGroup/Quest2.agda`. They should look like

```
PathNoConfusion : (x y : A  B) → (x  y)  NoConfusion x y
PathNoConfusion = {!!}

isSetNoConfusion : isSet A → isSet B → (x y : A  B) → isProp (NoConfusion x y)
isSetNoConfusion = {!!}
```

Tip: If you are tired of writing `{A B : Type} →` each time you can stick

```
private
  variable
    A B : Type
```

at the beginning of your `agda` file, and it will assume `A` and `B` implicitly whenever they are mentioned. Make sure it is indented correctly.

Without showing either of these new definitions, try using them to complete `isSet`.

We can use `pathToFun` or `endPt` to follow how a point of "`isProp` applied to `NoConfusion`" changes into a point of "`isProp` on the path space `x  y`".

```
isSet : {A B : Type} → isSet A → isSet B → isSet (A  B)
isSet hA hB x y = pathToFun {!!} (isSetNoConfusion hA hB x y)

isSet' : {A B : Type} → isSet A → isSet B → isSet (A  B)
isSet' hA hB x y = endPt {!!} {!!} (isSetNoConfusion hA hB x y)
```

```
isSet : {A B : Type} → isSet A → isSet B → isSet (A  B)
isSet hA hB x y = pathToFun (cong isProp (sym (PathNoConfusion x y)))
                    (isSetNoConfusion hA hB x y)

isSet' : {A B : Type} → isSet A → isSet B → isSet (A  B)
isSet' hA hB x y = endPt ( A → isProp A) (sym (PathNoConfusion x y))
                    (isSetNoConfusion hA hB x y)
```

Proving isSetNoConfusion

We will now show that NoConfusion “is a set”. Locate your definition of isSetNoConfusion and try proving it.

We need to case on the points in A   B.

- If they are both “from A” then we need to show that the path spaces in A are propositions.
- (2 cases) If they are from different spaces then we must show that the path spaces in are propositions.
- If they are both “from B” then it is similar to the first case.

3.5.5 Part 4 - Proving PathNoConfusion

It suffices to make an isomorphism

Replicate our proof of flipPath in quest 0, it suffices to show an isomorphism instead of an equality. Make this precise in 1FundamentalGroup/Quest2.

So that you can follow, we will make a lemma (you don’t have to) :

```
PathNoConfusion : (x y : A   B) → (x   y)   NoConfusion x y
PathNoConfusion = {!!}
```

To prove the isomorphism (for each arbitrary x and y) we need four things, which we can extract as local definitions / lemmas using where.

```
fun : (x y : A   B) → (x   y) → NoConfusion x y
fun x y = {!!}

inv : (x y : A   B) → NoConfusion x y → x   y
inv x y = {!!}

rightInv : (x y : A   B) → section (fun x y) (inv x y)
rightInv {A} {B} = {!!}

leftInv : (x y : A   B) → retract (fun x y) (inv x y)
leftInv = {!!}
```

inv

First try defining inv : (x y : A B) → NoConfusion x y → x y.

Check the goal. You can assume points x y : A B and a point h : NoConfusion x y. If you case on x and y you might find there are fewer cases than you need. This is because NoConfusion (inl ax) (inr by) was defined to be empty, so agda automatically removes the case.

In the case that both points are from x we need to show that given a proof p : ax ay we get a proof of inl ax inr ay. We already have the result that if two points are equal then their images under a function are equal.

```
inv : (x y : A   B) → NoConfusion x y → x   y
inv (inl x) (inl y) p = cong inl p
inv (inr x) (inr y) p = cong inr p
```

Attempting fun

We try to define the map forward, which we called fun. If we assume and case on x and y in the disjoint sum then

- When x and y are both from A then they will be inl ax and inl ay, so checking the goal we should be required to give a point in inl x inl y → x y. Reading this carelessly one could call this "inl is injective".
- When x and y are from different spaces then checking the goal, we should be required to give a point in inl ax inr by → . This says there are no paths between the disjoint parts.
- The last case is similar to the first.

We can extract the second case as a lemma :

```
disjoint : (a : A) (b : B) → inl a  inr b →
disjoint a b p = {!!}
```

which we can prove by constructing a *subsingleton bundle* over A B, just like we did to prove that true false is empty, in the *side quest*. In fact this is a generalisation of that result, and the proof also generalises.

We make a bundle over the disjoint union with the starting fiber as and the ending fiber as .

```
disjoint : (a : A) (b : B) → inl a  inr b →
disjoint a b p = endPt bundle p tt where

  bundle : A  B → Type
  bundle (inl a) =
  bundle (inr b) =
```

The other case is quite problematic. This is what we want to show

```
inlInj : (x y : A) → (inl {A} {B} x  inl y) → x  y
inlInj x y p = {!!}
```

Here are the problems:

- If we had a map backwards that cancelled inl we would be done, but in general this doesn't exist. For example, if A were empty and B had a point then we cannot expect to have a map A B → A.
- There is nothing to induct on : we have no information about x y : A. More importantly :

  Important: We don't know how to induct on paths.

  Specifically we don't yet know how to map out of a path space in general.

To find out how to induct on paths, complete *quest 4* in trinitarianism, and return to this quest with a completely new perspective.

3.5.6 Part 5 - Using the Propositional Perspective

After learning about the propositional perspective on equality, we can review some of the things we showed in a new light :

- `a  b →` can be read as `a` is not equal to `b` since assuming a proof that `a` is equal to `b` we have a point in the empty space.
- In showing an isomorphism between spaces we must show that two functions satisfy `fun (inv x)  x` for each `x` in the domain. This can now be read as `fun` composed with `inv` is equal to the identity on points.
- `isoToPath` says that if two spaces are isomorphic then they are equal.
- `endPt` (`subst` for substitute in the library) takes a bundle and a proof that `x  y` in the base space and substitutes `x` for `y`, hence replacing a point in the fiber of `x` with a point in the fiber of `y`.
- `cong : (f : A → B) → (p : x  y) → f x  f y` says that if two points are equal then their images are equal.
- `true` is not equal to `false`
- `refl` is not equal to `loop`
- `flipPath : Bool  Bool` is a non-trivial equality between `Bool` and itself.
- `inl` is injective (we still have not shown this yet).
- The objective of this whole arc is to show that the fundamental group of the circle is *equal* to .
- `isProp` says there is at most one point in the space; at most one proof of the proposition. Classically propositions are meant to only have a single proof ("proof irrelevance"), because for propositions `A` and `B`, having implications `A → B` and `B → A` is enough to show `A  B`.
- `isSet` says between any two points there is at most one path between them, i.e. "there is only `refl`", i.e. the space is disjoint.

We shall apply this perspective to the problem at hand.

fun

Now that we know how to induct on paths, we need to pick a path to induct on. Continuing with trying to show that `inl` is injective we will notice that path induction does *not* actually work here, since we have

- a start point `ax : A`
- a variable end point `ay : A`
- but the path is in the disjoint union `inl ax  inl ay` not a path in `A`

We instead take a step back and look at `fun` itself. (You can now abandon `inlInj` if you like, this will become a corollary of the classification.) We also remove the cases so that we are back to just having

```
fun : (x y : A  B) → (x  y) → NoConfusion x y
fun x y = {!!}
```

You might have noticed by now that we are in the perfect position to induct on paths in `x  y`. Path induction - `J` - says that to make a function `(x y : A B) → (x  y) → NoConfusion x y`, it suffices just to give a point in `NoConfusion x x`. Formalise the above (without showing `NoConfusion x x` yet) :

```
fun : (x y : A  B) → (x  y) → NoConfusion x y
fun x y = J ( y' p → NoConfusion x y') {!!}
```

To prove NoConfusion x x it would be convenient to be able to case on x so we will extract it as a lemma. Once you extract and case on x this it should be quite easy to show.

```
NoConfusionSelf : (x : A  B) → NoConfusion x x
NoConfusionSelf (inl x) = refl
NoConfusionSelf (inr x) = refl
```

rightInv

Try to define rightInv : (x y : A B) → section (fun x y) (inv x y).

It is a good idea to case on x and y in the space A B, since inv is the first to take these inputs in here, and inv was defined by casing on x and y. This should reduce us to just two cases, like when defining inv. We will just describe the case when they are both from A.

We can use J to reduce to the case of when the path is refl. (No proof of the refl case yet.)

```
rightInv : (x y : A  B) → section (fun x y) (inv x y)
rightInv {A} {B} (inl x) (inl y) p =
   J ( y' p → fun {A} {B} (inl x) (inl y') (inv (inl x) (inl y') p)  p) {!!}
```

We added the implicit arguments {A} and {B} so we can actually access them here. The remaining hole is for showing that

```
fun (inl x) (inl x) (inv (inl x) (inl x) refl)  refl
```

It would help to make a chain of equalities. We defined inv (inl x) (inl x) refl to be refl, so we only need to show that

```
fun (inl x) (inl x) refl  refl
```

Since fun was defined using J we need to know how J computes when it is fed refl. We *described this before*, it is called JRefl.

```
rightInv : (x y : A  B) → section (fun x y) (inv x y)
rightInv {A} {B} (inl x) (inl y) p = J ( y' p → fun {A} {B} (inl x) (inl y') (inv (inl␣
↪x) (inl y') p)  p)
                     (
                       fun {A} {B} (inl x) (inl x) refl
                      JRefl {x = inl x} (( y' p → NoConfusion {A} {B} (inl x) y')) _
                     -- uses how J computes on refl
                       refl
                     ) p
rightInv {A} {B} (inr x) (inr y) p = {!!}
```

leftInv

Try to define leftInv.

We do this but each part of this proof will be relevant anywayuse J since fun "happens first". This should reduce the problem to showing

```
inv x x (fun x x refl)  refl
```

```
leftInv : (x y : A  B) → retract (fun x y) (inv x y)
leftInv x y = J ( y' p → inv x y' (fun x y' p)  p) {!!}
```

If you extract what is needed as a lemma you can case on the variable. Remember to use JRefl for the application of fun.

```
leftInv : (x y : A  B) → retract (fun x y) (inv x y)
leftInv x y = J ( y' p → inv x y' (fun x y' p)  p)
                (
                  (inv x x (fun x x refl))
                 cong (inv x x) (JRefl (( y' p → NoConfusion x y')) _)
                  inv x x (NoConfusionSelf x)
                 lem x
                  refl
                ) where

  lem : (x : A  B) → inv x x (NoConfusionSelf x)  refl
  lem (inl x) = refl
  lem (inr x) = refl
```

3.6 Quest 3 - The Loop Space is

In *Quest 1 - Loop Space of the Circle* we introduced our main method of proving that the fundamental group (which we take to be loopSpace S^1 base for now) is , and in *Quest 2 - is a Set* we decided that this means to show that they are equal spaces. .. admonition:: The Goal

```
loopSpace : loopSpace S¹ base
loopSpaceZ = {!!}
```

As usual we will show this via giving an isomorphism, so we must make comparison maps forward and back. However, we discovered we had to define the backwards map *over all of* S^1. We already have windingNumber, the forwards comparison map, which gives us a map loopSpace S^1 base → when applied to base.

```
windingNumber : (x : S¹) → base  x → helix x
```

In this quest our goal is to make a map backwards

Current Goal

```
rewind : (x : S¹) → helix x → base  x
```

Since `windingNumber` took a path and found how many times the path loops around, in general “an integer twisted around the helix a bit”, or “an integer plus a bit”. We want to make `rewind` do the reverse. So `rewind` should take “an integer `n` plus a bit”, loop around `n` times, then add that extra corresponding bit, the path from `base` to `x` to the end.

3.6.1 Part 0 - `rewind`

Dependent paths

We try making `rewind`. We can assume a point `x :  S¹`, then case on what it is.

```
rewind : (x : S¹) → helix x → base  x
rewind base = {!!}
rewind (loop i) = {!!}
```

In the case of `base` we want a map from `helix base` i.e. , to `base  base`. Try filling this in.

We want this to be the correct inverse, described as looping around `n` times and adding that extra bit on the end. However there is nothing to add at the end in this case, so it should just be `loop_times`, which we already defined in *Quest 1 - Loop Space of the Circle*.

The case of `loop i` will be a lot more work. Checking the goal we see that at each point `loop i` on the loop, it wants a point in the space `helix (loop i) → base  (loop i)`, which it might reduce to `sucPath i → base  (loop i)` according to the definition of `helix`.

Collecting these spaces together along this `i`, we obtain a loop in the space of spaces based at the space `→ base  base` given by

```
 i → helix (loop i) → base  (loop i) : ( → base  base)  ( → base  base).
```

Now collecting the points we need to give into a “path” as well, we obtain the notion of a *dependent path* : each point of this “path” belongs to a space along the path of spaces. We define dependent paths and design a way of mapping out of `S¹` in general in *Quest 5 - Dependent Paths* from *Trinitarianism*. We assume from now on knowledge of dependent paths.

Using `outOfS¹`

Now that we have a way of mapping out of `S¹` (using `PathD`), called `outOfS¹D`, try to use it to repackage the work we have to far.

Originally we have

```
rewind : (x : S¹) → helix x → base  x
rewind base = loop_times
rewind (loop i) = {!!}
```

Now we rearrange this to

```
rewind : (x : S¹) → helix x → base  x
rewind = outOfS¹D ( x → helix x → base  x) loop_times {!!}
```

since our bundle over `S¹` is `( x → helix x → base  x)` and our image for `base` is `loop_times`.

Checking the last goal, it remains to give a dependent path of type `PathD ( i → sucPath i → base  loop i) loop_times loop_times`. Remembering the definition of `PathD`, this should be exactly giving a path `pathToFun ( i → sucPath i → base  loop i) loop_times  loop_times`, since `PathD` reduces the issue of dependent paths to just paths in the end space, which is `→ base  base` in this case. Let’s make this a chain of equalities :

```
rewind : (x : S¹) → helix x → base  x
rewind = outOfS¹D ( x → helix x → base  x) loop_times
  (
    pathToFun ( i → sucPath i → base  loop i) loop_times
   {!!}
    loop_times
  )
```

Functions and `pathToFun`

The map `loop_times` takes an integer and does `loop` that many times. On the other hand `pathToFun` follows how `loop_times` changed along the path of spaces ` i → sucPath i → base  loop i`, and spits out the corresponding point at the end. This path of spaces is specifically a path of *function spaces*, so we need to find a more explicit way of describing what `pathToFun` does to spaces of functions.

To generalize, suppose we have spaces `A0 A1 B0 B1 :  Type` and paths `A : A0  A1` and `B : B0  B1`. Then let `pAB` denote the path ` i → A i → B i :  (A0 → B0)  (A1 → B1)`. We want to figure out what `pathToFun` does when it follows a function `f :  A0 → B0` along the path `pAB`.

We know by functional extensionality that the function `pathToFun pAB f :  A1 → B1` should be determined by what it does to terms in `A1`, so we can assume `a1 :  A1`. The idea is we “apply `f` by sending `a1` back to `A0`”. Since `pathToFun (sym A) a1` is meant to give the point in `A0` that “looks like `a1`”, we try applying `f` to this point, then send it across again via the path `B` to the point `f (pathToFun (sym A) a1)` looks like in `B1`. We expect the outcome to be the same.

```
pathToFun→ : {A0 A1 B0 B1 : Type} {A : A0  A1} {B : B0  B1} (f : A0 → B0) →
  pathToFun ( i → A i → B i) f  a1 → pathToFun B (f (pathToFun (sym A) a1))
```

$$\begin{tikzcd}
A_1 \arrow[r, dashed, "\texttt{pathToFun}\ p_{AB}\ f"] & B_0 \\
A_0 \arrow[u, "\texttt{pathToFun}\ A"] \arrow[r, "f"'] & B_1 \arrow[u, "\texttt{pathToFun}\ B"']
\end{tikzcd}$$

The proof of this in `cubical agda` is simply `refl`, so we need not even extract it as a lemma.

A cubical hack

Is actually one of the axioms asserted in `cubical agda` that `pathToFun ( i → A i → B i) f` is *externally equal to* `a1 → pathToFun B (f (pathToFun (sym A) a1))`. Here we are using the `cubical` definition of `pathToFun` so we can simply give `refl` for its proof.

However, according the definition of `pathToFun` we gave in *Trinitarianism*, they are not externally equal but can be shown to be internally equal using `J`. We prove this from *our own definitions here*.

We interpret what this result means in our specific case : We are making `pathToFun ( i → sucPath i → base loop i) loop_times` into another map in the space `→ base  base`, by following along the diagram

$$\begin{tikzcd}
\mathbb{Z} \arrow[r, dashed] & \texttt{base} \equiv \texttt{base} \\
\mathbb{Z} \arrow[u, "\texttt{pathToFun}\ \texttt{suc}\mathbb{Z}\texttt{Path}"] \arrow[r, "\texttt{loop\_times}"'] & \texttt{base} \equiv \texttt{base} \arrow[u, "\texttt{pathToFun}\ p_{\equiv}"']
\end{tikzcd}$$

Specifically, this map should take `n :`   and first send it backwards along `sucPath`, supposedly giving us `n - 1`. Then it applies `loop_times`, obtaining the loop `loop (n - 1) times`. Lastly it follows `loop (n - 1) times` along the path `i → base  loop i` (which itself is a loop starting and ending at `base  base` in the space of spaces), obtaining some path from `base  base`, which we expect to be internally equal to `loop n times`.

Try putting this together in our definition of `rewind`, as a new intermediate step in our chain of equalities.

```
rewind : (x : S¹) → helix x → base  x
rewind = outOfS¹D ( x → helix x → base  x) loop_times
  (
    pathToFun ( i → sucPath i → base  loop i) loop_times
   refl
    ( n → pathToFun ( i → base  loop i) (loop_times (pathToFun (sym sucPath) n)))
   {!!}
    loop_times
  )
```

We can simplify the above expression. We know that `pathToFun (sym sucPath) n` should follow `n` along `sucPath` backwards, so it should be `n - 1`. We can use this to move a step closer to the goal.

This equality is *definitional*.

```
rewind : (x : S¹) → helix x → base  x
rewind = outOfS¹D ( x → helix x → base  x) loop_times
```

(continues on next page)

(continued from previous page)

```
  (
    pathToFun ( i → sucPath i → base  loop i) loop_times
   refl
    ( n → pathToFun ( i → base  loop i) (loop_times (pathToFun (sym sucPath) n)))
   refl
    ( n → pathToFun ( i → base  loop i) (loop (pred n) times))
   {!!}
    loop_times
  )
```

The path fibration and pathToFun

It remains to find out how pathToFun interacts with the path of loops coming out of base. We call “the path of loops coming out of base” i → base loop i the *path fibration* at base. The animation tells us that we are gradually concatenating the input loop (n - 1) times with loop. Hence we *should* obtain loop (n - 1) times loop. We are a bit lucky here, and these are in fact *definitionally equal*, but to justify this in general, we can prove that “following along the path fibration is the same as concatenating”.

```
pathToFunPathFibration : {A : Type} {x y z : A} (q : x  y) (p : y  z) →
  pathToFun ( i → x  p i) q  q  p
```

This is in fact a quick exercise.

We take the propositional perspective - without loss of generality we can assume y and z are exactly the same.

Crucially : we know what pathToFun does to refl (recall pathToFunRefl from *the quest on paths*).

```
pathToFunPathFibration : {A : Type} {x y z : A} (q : x  y) (p : y  z) →
  pathToFun ( i → x  p i) q  q  p
pathToFunPathFibration {A} {x} {y} q = J ( z p → pathToFun ( i → x  p i) q  q  p)
  (
    pathToFun refl q
   pathToFunRefl q
    q
   Refl q
    q  refl
  )
```

To include this in rewind we have

```
rewind : (x : S¹) → helix x → base  x
rewind = outOfS¹D ( x → helix x → base  x) loop_times
  (
    pathToFun ( i → sucPath i → base  loop i) loop_times
   refl  -- how pathToFun interacts with →
    ( n → pathToFun ( i → base  loop i) (loop_times (pathToFun (sym sucPath) n)))
   refl  -- sucPath is just taking successor, and so its inverse is definitionally␣
↪taking predecessor
    ( n → pathToFun ( i → base  loop i) (loop_times (pred n)))
   funExt ( n → pathToFunPathFibration _ _)  -- how pathToFun interacts with the "path␣
↪fibration"
    ( n → (loop (pred n) times)  loop)
```

(continues on next page)

(continued from previous page)

```
   {!!}
    loop_times
  )
```

There are several ways to complete this final part. We will leave the rest in a hint.

Applying functional extensionality we just need to show that for each `n :  ` the outputs are equal, i.e. `loop pred n times  loop  loop n times`. By our design of `loop_times` we should have that `loop m times  loop` is equal to `loop (m + 1) times`. Then we are reduced to showing that `loop (suc pred n) times  loop n times`, or just `suc pred n  n`.

```
rewind : (x : S¹) → helix x → base  x
rewind = outOfS¹D ( x → helix x → base  x) loop_times
  (
    pathToFun ( i → sucPath i → base  loop i) loop_times
   refl  -- how pathToFun interacts with →
    ( n → pathToFun ( i → base  loop i) (loop_times (pathToFun (sym sucPath) n)))
   refl  -- sucPath is just taking successor, and so its inverse is definitionally␣
↪taking predecessor
    ( n → pathToFun ( i → base  loop i) (loop_times (pred n)))
   funExt ( n → pathToFunPathFibration _ _)  -- how pathToFun interacts with the "path␣
↪fibration"
    ( n → (loop (pred n) times)  loop)
   funExt ( n →
       loop pred n times  loop
       loopSuctimes (pred n)
        loop (suc (pred n)) times
       cong loop_times (sucPred n)
        loop n times )
    loop_times
  )
```

We can check that `rewind base` is indeed `loop_times` by using `C-c C-n`. This is to be expected as `outOfS¹` evaluated at `base` should back exactly what we fed it, as mentioned in the discussion on mapping out of the circle.

3.6.2 Part 1 - `rewind` is a right inverse

We are now in a position to approach the main goal :

```
loopSpaceS¹ : loopSpace S¹ base
loopSpaceS¹ = {!!}
```

We have reduced this to giving an isomorphism, which involves giving the map `windingNumber base` forward and `loop_times` backwards, and showing that they are inverses of each other.

Hence the next step is to show that "looping `n` times then taking the winding number gives back `n`". Try to state and prove this in `1FundamentalGroup/Quest3.agda`. In the hints we will use intuitive notation for integers that may not align exactly with `agda` code.

```
windingNumberRewindBase : (n :  ) → windingNumber base (rewind base n)  n
windingNumberRewindBase = {!!}
```

We identify `rewind base` with `loop_times`, since they are externally equal.

Since `loop_times` was defined by casing on `n` we case on `n` - it could be zero, a positive integer, negative one, or less than negative one.

Some of the cases are trivial - we know exactly what `loop 0 times` and `windingNumber base loop` are.

```
windingNumberRewindBase : (n : ) → windingNumber base (rewind base n)  n
windingNumberRewindBase (pos zero) = refl
windingNumberRewindBase (pos (suc n)) = {!!}
windingNumberRewindBase (negsuc zero) = refl
windingNumberRewindBase (negsuc (suc n)) = {!!}
```

We can identify `windingNumber base` with its definition, reducing the problem to showing that `endPt helix (loop n times) 0` is equal to `n`, in the separate cases.

For the first case, we can reduce `loop (n + 1) times` to just `loop n times  loop` since that was the definition. Hence we are interested in what `endPt helix (loop n times  loop) 0` is. Recalling our intuition behind `endPt`, this amounts to following the point `0` up the `helix` along the path `loop n times  loop`. This should just be going to `endPt helix (loop n times) 0` then adding 1.

You can also check what `agda` reduces the expression to by writing it in the hole and then doing `C-c C-n`. It should look something like `suc (transp ( i → helix (loop pos n times i)) i0 (pos 0))`. Clearly it has reduced the definition a bit too far, but the important idea is there, that it is `+ 1` of whatever data we have already.

Lastly we can just take `suc` on both sides of an equality we have from the induction hypothesis.

For one of the cases we detail the thought process going on above, and for the last case we extract only the important part of the proof.

```
windingNumberRewindBase : (n : ) → windingNumber base (rewind base n)  n
windingNumberRewindBase (pos zero) = refl
windingNumberRewindBase (pos (suc n)) =
    windingNumber base (rewind base (pos (suc n)))
   refl
    windingNumber base (loop (pos n) times  loop)
   refl
    endPt helix (loop (pos n) times  loop) (pos zero)
   refl
    suc (endPt helix (loop (pos n) times) (pos zero))
   cong suc (windingNumberRewindBase (pos n))
    suc (pos n)
   refl
    pos (suc n)
windingNumberRewindBase (negsuc zero) = refl
windingNumberRewindBase (negsuc (suc n)) = cong pred (windingNumberRewindBase (negsuc n))
```

You might wonder if it is possible to make the above map work across all of S^1, and the answer is yes. This is not really necessary for our goal, so feel free to skip to the next part if you are not interested. Try stating and proving the generalization of the above; which we call `windingNumberRewind`.

```
windingNumberRewind : (x : S¹) (n : helix x) → windingNumber x (rewind x n)  n
windingNumberRewind = {!!}
```

We defined `rewind` by casing on points in the circle and `rewind` is the first function being applied, so it would make sense to case on points in the circle. In the case when the point is `base` we can just give the map we wanted to generalize in the first place.

```
windingNumberRewind : (x : S¹) (n : helix x) → windingNumber x (rewind x n)  n
windingNumberRewind =
 outOfS¹D ( x → (n : helix x) → windingNumber x (rewind x n)  n)
  windingNumberRewindBase {!!}
```

Checking the last hole we see that we need to give a dependent path from `windingNumberRewindBase` to itself. According to the definition of a dependent path, this is just a path in the last fiber from `pathToFun` of `windingNumberRewindBase` to `windingNumberRewindBase` (the fiber is `(n :   )  → windingNumber base (rewind base n)  n`). Now this might seem very complicated : even after applying functional extensionality (this is equality of two functions) this would be "finding a path between paths in ". Try repeating that last bit in your head a couple of times.

We put a lot of effort into showing that is a set.

```
windingNumberRewind : (x : S¹) (n : helix x) → windingNumber x (rewind x n)  n
windingNumberRewind = -- must case on x / use recursor / outOfS¹ since that is def of␣
↪rewind
 outOfS¹D ( x → (n : helix x) → windingNumber x (rewind x n)  n)
  windingNumberRewindBase (
   pathToFun
    ( i → (n : helix (loop i)) → windingNumber (loop i) (rewind (loop i) n)  n)
    windingNumberRewindBase
  funExt ( x → isSet _ _ _ _ )
   windingNumberRewindBase )
```

3.6.3 Part 2 - `rewind` is a left inverse

Try to show that `rewind` is a left inverse.

Just like we struggled to only define `windingNumber base` without access to the entire circle, we make sure to include all the data we have access to. Note that this was not the case before.

```
rewindWindingNumber : (x : S¹) (p : base  x) → rewind x (windingNumber x p)  p
rewindWindingNumber x = {!!}
```

Remembering that `windingNumber x p` is externally equal to `endPt helix p 0`, and that `endPt` is defined by path induction - using `J` (this is not exactly true for `endPt` from the library for `cubical` reasons), the obvious thing to do here is to do path induction.

```
rewindWindingNumber : (x : S¹) (p : base  x) → rewind x (windingNumber x p)  p
rewindWindingNumber x = J ( x p → rewind x (windingNumber x p)  p) {!!}
```

It suffices to show that `rewind x (windingNumber x refl)  refl`, which by reducing the left side is the same as showing `loop_times (endPt helix refl 0)  refl`.

```
rewindWindingNumber : (x : S¹) (p : base  x) → rewind x (windingNumber x p)  p
rewindWindingNumber x = J ( x p → rewind x (windingNumber x p)  p)
   (rewind base (windingNumber base refl)
  refl
   loop_times (endPt helix (refl {x = base}) (pos zero))
  {!!}
   refl )
```

We know what `endPt` does to `refl`, which is given by the result `endPtRefl`. If you need to recall what `endPtRefl` proves you can type it into the hole and do `C-c C-.` for the goal and the type of `endPtRefl`.

```
rewindWindingNumber : (x : S¹) (p : base  x) → rewind x (windingNumber x p)  p
rewindWindingNumber x = J ( x p → rewind x (windingNumber x p)  p)
     (rewind base (windingNumber base refl)
    refl
     loop_times (endPt helix (refl {x = base}) (pos zero))
    cong loop_times (cong ( g → g (pos zero)) (endPtRefl {x = base} helix))
     loop (pos zero) times
    {!!}
     refl )
```

The last step is simply remembering how `loop_times` computes.

```
rewindWindingNumber : (x : S¹) (p : base  x) → rewind x (windingNumber x p)  p
rewindWindingNumber x = J ( x p → rewind x (windingNumber x p)  p)
     (rewind base (windingNumber base refl)
    refl
     loop_times (endPt helix (refl {x = base}) (pos zero)) -- reduce both definitions
    cong loop_times (cong ( g → g (pos zero)) (endPtRefl {x = base} helix))
     loop (pos zero) times
    refl
     refl )
```

3.6.4 Part 3 - The Loop Space is

We can conclude our main goal now, by collecting all of the components we have made above. We leave you the pleasure.

As usual we construct an isomorphism, but we can choose to do this over the entire circle or just between `loopSpace S¹ base` and . We do the former and have the latter as a corollary, but you could just do the latter directly as well.

```
pathFibrationhelix : (x : S¹) → (base  x)  helix x
pathFibrationhelix x =
  isoToPath (iso (windingNumber x) (rewind x) (windingNumberRewind x)␣
↪(rewindWindingNumber x))

loopSpaceS¹ : loopSpace S¹ base
loopSpaceS¹ = pathFibrationhelix base
```

What now?

We have mentioned already that we aren't *exactly* working with the fundamental group, but the loop space. In the final quest of this arc we discuss the definition of the fundamental group and show that the loop space in this case is the fundamental group already.